HARVARD UNIVERSITY

BACHELOR'S THESIS

Teach a Fish to Swim: Evaluating the Ability of Turing Learning to Infer Schooling Behavior

---

Author:
Katherine Binney

A thesis submitted in partial fulfillment of the honors requirements for the degree of Bachelor of Arts

in

Computer Science

March 29, 2019

HARVARD UNIVERSITY

Abstract

Teach a Fish to Swim: Evaluating the Ability of Turing Learning to Infer Schooling Behavior

by Katherine Binney

Turing Learning is a promising evolutionary design method for swarm robotics that uses observation of natural or artificial systems to infer controllers for agents in a swarm. However, Turing Learning has thus far only been used to infer very simple swarm behaviors. In this work, we expand Turing Learning to infer dispersion, a much more complex swarm behavior, by a simulated school of robotic fish. Turing Learning depends on the co-evolution of replicas and classifiers. Replicas mimic ideal behavior and classifiers distinguish between data samples from replica and ideal agents. We model replicas and classifiers with neural networks and investigate the architecture of each component independently in order to determine needed modifications to Turing Learning for it to infer fish schooling. We find that previously formulated data samples led to the inference of behaviors that locally mimicked the agent trajectories in dispersion, yet poorly mimicked dispersion of an entire swarm. We present three alternative data samples that consider the spatial arrangement of agents in a swarm. We also introduce three new classifier fitness functions that accelerate evolution of high-accuracy classifiers. We find in a preliminary trial that using one of our data samples (metrics) and classifier fitness functions ($f_{outputs}$) enables the successful inference of dispersion via Turing Learning.

Acknowledgements

I'd like to thank my faculty advisor Radhika Nagpal, for her support and insight throughout the process. Melvin Gauci served as an additional unofficial advisor, contributing incredibly valuable insights and subject-specific technical expertise. I'd further like to thank Florian Berlinger for discussion of ideas and for allowing me to use the simulator he had developed for his robotic fish. I'm also grateful for Magalay Gutierrez's collaboration during initial stages of the project. I appreciate the time being taken by my two thesis readers, Jim Waldo and Finale Doshi-Velez, to read and provide comments on this thesis. I'd like to give a special shout out to Professor Waldo for reading and providing comments on an initial partial draft. Finally, I'd like to thank my roommate, friends, and family for their enthusiastic support throughout this process.

Contents

1 Introduction

Self-organizing systems are abundant in nature. In these systems, many individual units work together in a decentralized manner to achieve a goal larger than any one individual could accomplish alone. Birds fly in large flocks, changing direction on a moment's notice. Termites build large mounds. Fish school in the hundreds. These natural systems have provided inspiration to the field of swarm robotics, which attempts to build robotic systems capable of such collective behavior and self-organization.

Though we observe great feats of self-organization and collective behavior abundantly in the natural world, it has proved difficult to design similar behavior in artificial systems. A grand challenge in the field of swarm robotics has been in the design of a global to local system that can synthesize local behavioral rules that achieve a macro-level goal. Progress has been made in developing such a system for particular behaviors in tightly constrained scenarios, but no universally applicable compiler or system yet exists.

Artificial evolution is one technique that may eventually be used for the automatic global-to-local transformation. Artificial evolution refers to an optimization procedure inspired by principles of natural evolution. In these procedures, a set (termed a *population*) of candidate solutions is proposed. A candidate solution is evaluated using a function called a *fitness function*. In the most simple case, artificial evolution is used to find the input that minimizes a function. In this case, the population of candidate solutions is a set of real numbers, and the fitness function is the function to be optimized. In a single *generation*, all members of a population are evaluated with this fitness function. Those with the highest fitness are used to influence the next set of candidate solutions via random *mutation* (creating more candidate solutions similar to the highest fitness candidate solutions) and *crossover* (combining two high fitness candidate solutions into a new candidate solution). Over many generations, this approach achieves a search across the parameter space of candidate solutions. Solutions generated with this approach are said to be *learned* or *evolved*, and the process of finding solutions over many generations *learning* or *evolution*. Mutation and crossover of candidate solutions, hallmarks of many evolutionary algorithms, may offer movement across the search space in a substantially different manner than other heuristic-based optimization algorithms (De Castro, 2006).

The design of swarm robotics systems provides an interesting search space. When applying

artificial evolution to robotics, candidate solutions generally are a component of a robot's *controller*. The controller is a hardware and/or software system that governs the interaction between a robot's sensors and its internal state and observable behavior. An example candidate solution in robotics is the weights of a neural network that maps a binary set of inputs from a robot's contact sensor to its wheel speed. The corresponding fitness function measures the quality of the observed behavior of a robot, rather than directly examining the generated controller parameters. Continuing with our example, to evolve a robot that moves away from obstacles, a fitness function could measure the time a robot spends with its contact sensor activated. When used to design controllers for or physical architecture of robots, artificial evolution is generally termed *Evolutionary Robotics*.

The design of robotic swarms provides a particularly interesting search space for artificial evolution. Not only are there many possible controllers for an individual robot, there are also often complex and poorly understood relationships between individual and group-level behavior. Using evolutionary robotics techniques, we can generate a population of agent-level controllers and evaluate the fitness of a controller based on the behavior of a swarm of agents using this controller. When candidate solutions are executed on agents in a collective, the relationship between controllers and group behavior is actualized, and the final group level behavior evaluated to drive forward evolution. As a result, artificial evolution has become a popular method for designing collective behavior.

1.1 Turing Learning

One evolutionary learning method, Turing Learning, is promising in its ability to infer controllers for agents from observation of an entire swarm (Li, Gauci, and Groß, 2016). Evolution of agent-level controllers for interesting collective behavior typically requires the use of hand-coded fitness functions. These fitness functions are based on predefined metrics researchers believe represent a macro-level behavior. While often successful, such an approach generally requires significant trial and error or pre-existing knowledge of the dynamics of the collective behavior. Further, the specificity of fitness functions generally limits the novelty of controllers achieving the desired behavior that can be learned. Turing Learning offers a metric-free approach, automatically inferring individual-level behavior that is important to achieving a collective goal. This technique could ultimately be used to develop controllers for robots in collectives such that the evolved swarms mimic a natural system, even if the system is not yet well understood. Turing Learning could also be used to design controllers for robots that mimic animal behavior despite having significantly different sensing and movement capabilities than their biological counterparts. Such an automatic design system will improve the development of effective *biomimics*: robots intended to

be integrated into biological systems and socially accepted by the biological agents in the system.

Turing Learning is an approach that simultaneously optimizes a population of *classifiers* and a population of *replicas*. The replicas are controllers for robots mimicking an unknown collective. The classifiers attempt to distinguish between replica and ideal agents. Competition between these populations drives the improvement of replicas. The simultaneous optimization of these populations via evolutionary algorithms has led Turing Learning to be categorized as a *co-evolutionary* approach. Turing Learning is inspired by Alan Turing's proposed test of machine intelligence. In the so-called "Turing Test," a person questions two hidden subjects and is asked at the conclusion of questioning to decide which of the two subjects is human (Turing, 1950). In Turing Learning, a classifier is tasked with distinguishing between data samples generated by an ideal system (a collective, or swarm, of agents executing a desired behavior) and counterfeit data samples generated by a replica system (a collective of agents we wish to execute the desired behavior). Classifiers and replicas that are successful in their respective tasks drive optimization.

Turing Learning has proved able to evolve near-perfect replicas in reactive agents executing collective aggregation and object clustering behavior (Li, Gauci, and Groß, 2016). In this work, we examine the ability of Turing Learning to infer more complex behaviors. In particular, we modify Turing Learning to infer dispersion behavior executed by simulated robotic schools of fish.

1.2 Fish Schooling

Craig Reynolds' seminal work on flocking in (1987) initiated the development of a large body of work that attempts to mathematically model the dynamics of movement of schools of fish. *Flocking* refers to the coordinated movement of a group of agents. In his work, Reynolds proposed three rules of behavior needed to model flocking: collision avoidance, velocity matching, and flock centering (1987). Reynolds demonstrated that these simple rules generated animations of flocks that appeared natural to human observers. Reynolds examined flocking in the abstract. He did not claim to model the behavior of a particular biological collective. Nevertheless, Reynold's work has proved relevant to fish schooling. Later work explicitly used experimental data from observation of fish to infer dynamics governing movement of groups of fish (Katz et al., 2011). Katz et. al. found that governing motion as a linear function of pairwise interactions between fish and their neighbors, a common flocking paradigm informed by Reynold's work, qualitatively approximates the spatial dynamics of larger fish collectives, even if it ignores substantial three-body contributions to dynamics. Recent work in flocking as a general behavior has formalized Reynolds' three rules into formal algorithms locally executable by agents in a swarm such that the swarm can achieve size-tunable mapping and environment-based and leader-based migration (Delight et al.,

2016).

Models of flocking behavior show promise for designing robotic collectives. Delight et al. (2016) motivated their behavioral models via a collective of robotic boats intended to execute ocean-surface mapping. Flocking behaviors in nature, however, are often a 3D phenomenon. Fish are thus a particularly intriguing natural inspiration for new robotic collectives. Recent work has demonstrated the possibility of creating aquatic robots for use in a robotic fish collective (Berlinger et al., 2018).

1.3 Evolving Schooling

The depth of understanding of fish as collectives combined with current novel work in the development of robotic test beds has led us to choose to expand Turing Learning to infer behavior characteristic to fish schooling. In particular, in this work, we identify and overcome some of the limitations of the current implementation of Turing Learning in its ability to infer the dispersive behavior of a simulated school of robotic fish. The contributions presented in this work are as follows:

- The implementation of a Turing Learning system to infer schooling behavior of robotic fish in simulation. This system is quite modular so that it can be used again in future work. Modifying the system to infer new behaviors will require only minimal changes. We also implement an ideal artificial system from which to infer behavior.

- Identification of a mechanism for failures encountered in attempts to directly apply previous implementations of Turing Learning to our behavior. This failure and the corresponding cause is likely to be common to many potential applications of Turing Learning in swarm robots. These limitations thus inform the rest of the work, which proposes methods to overcome them.

- Verification that the replica model controller architecture proposed in this learning system is sufficiently expressive. It can behavior reasonably similar to that of the ideal system. We also present limitations of this architecture and justify its continued use: Its use allows us to examine the use of Turing Learning in situations where replica and ideal agent capabilities diverge.

- A formulation of alternative data samples to be generated by ideal and replica systems. We propose three observational data samples on the basis of which classifiers operate. One data sample, **neighbor awareness**, is local to individual agents, and is likely to allow Turing

Learning to infer dispersion but may not be generalizable to other applications. Another data sample, **metrics**, is a set of macro-level observations about a swarm. Turing Learning may have more difficulty inferring dispersion behavior using this data sample, but it is more likely to generalize to new applications in swarm robots. The final data sample, **position**, appears most natural to the promise of Turing Learning's metric-free approach, but proves too large to be effective for learning.

- The design and comparison of three new functions for determining the quality of a proposed classifier. We examine the use of these functions to train classifiers to distinguish data samples from the ideal system executing *aggregation* and *dispersion*. All functions are an improvement over previous formulations and two, $f_{outputs}$ and $f_{diversity}$, are shown to evolve a population of classifiers with extremely high classification accuracy in this context.

- A demonstration that combining two modifications to Turing Learning results in successful inference of dispersion by a replica swarm in a preliminary trial. We use the **metrics** data sample and the classifier fitness function f_{output} to successfully evolve a high quality replica swarm executing dispersion.

The outline of this paper is as follows. Chapter 2 describes related work. Chapter 3 outlines the implementation of Turing Learning used in this work. Chapter 4 presents experimental results, including the direct application of previous implementations of Turing Learning to fish schooling (Section 4.1), an analysis of replica model representation (Section 4.2), an analysis of classifier capabilities for a range of fitness functions and data samples (Section 4.3), and preliminary results of a full trial of Turing Learning using modifications presented and validated in this work (Section 4.4). We conclude the paper and suggest avenues for future research in Chapter 5.

2 Related Works

This section is organized as follows: We first outline general research areas in collective robotics to situate this work within the field. We then examine previous works in evolutionary robotics and Turing Leaning. We conclude with seminal research on flocking and schooling and works examining the inclusion of replicas in biological schools of fish.

2.1 Taxonomies of Swarm Robotics

In 2013, Brambilla et al. proposed two taxonomies by which to classify swarm robotics research. In the first taxonomy, they divide works in swarm robotics into works dealing with "design", the creation of systems, and works dealing with "analysis", the evaluation of a proposed or implemented system. In the second taxonomy, they categorize the types of collective behaviors evaluated by a work into spatially-organizing behaviors, navigation behaviors, collective decision-making, and other collective behaviors (Brambilla et al., 2013). According to their taxonomy, our work examines automatic design methods: We work to expand Turing Learning from application to the spatially-organizing behaviors of aggregation and object clustering to the navigation behavior of coordinated motion.

A more recent review of swarm robotics surveys the research by task, analyzing design methods, important works, and mathematical models and metrics used in each specified task (Bayındır, 2016). By Bayındır's taxonomy, this work contributes to design of flocking algorithms while previous work on Turing Learning looked at aggregation and object clustering. Bayındır's work includes an important aggregation of performance metrics. These metrics vary widely by task and by paper, even within a single task. This lack of common performance metrics may make traditional artificial evolution methods, which require metrics to evaluate the success of a model in achieving its goal, more difficult to design.

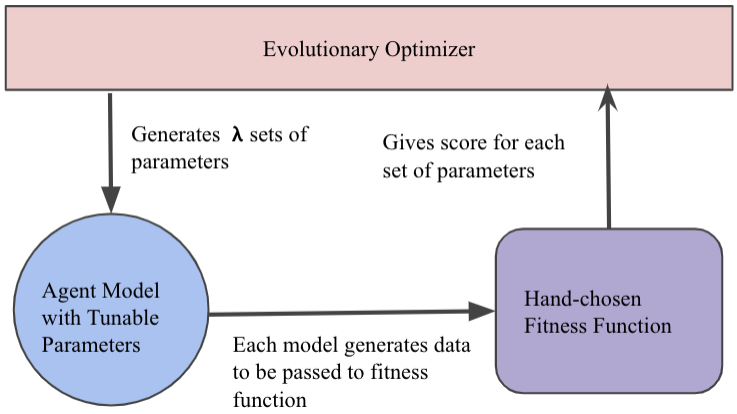


FIGURE 2.1: Traditional evolutionary robotics systems have three clear components: An optimizer, an agent model, and a fitness function.

2.2 Artificial Evolution

We first examine works in evolutionary robotics using traditional evolution systems. The general structure of these systems is given in Figure 2.1. A traditional evolution system begins with an evolutionary optimizer, which is the algorithm that generates a population of size λ of candidate solutions. Each of these candidate solutions is incorporated into an agent's controller. The agent's behavior is then scored using a fitness function. The fitness score for each agent is then communicated to the optimizer to inform the next generation of candidate solutions.

2.2.1 Evolution in Collectives

In 2004, Dorigo et al. demonstrated that evolutionary robotics could be effectively applied to swarm robotics by evolving swarms capable of aggregation and coordinated motion. We summarize the evolutionary system they used to evolve aggregation here. The system to evolve coordination motion was similar. The robots in their system, termed *s*-bots, had 8 proximity sensors and 3 sound sensors. The robots constantly emitted a noise and had two independently controlled bi-directional wheels. The authors modeled the controller of the robot as a single layer perceptron with 11 inputs (the sensor values) and 2 outputs, which controlled the speed of the two wheels. They designed an evolutionary optimizer for their system. In it, the 20 (of 100) highest-fitness controllers in a generation were selected. Each controller was mutated in 5 different ways to form 5 new candidate controllers for the subsequent generation, for a total of 100 candidate controllers in each generation. The final component of the system was a fitness function that computed the aggregation and movement quality of agents in a swarm.

Dorigo et al. (2004) were able to successfully evolve aggregation and, in a subsequent experiment, coordinated motion using this system. They further demonstrated that the evolved behavior was scalable: It could be used in different sized collectives. This work depends on the assumption that evolved behavior is scalable, an assumption validated by Dorigo et al. In addition, we use a similar model for replica robot controllers: a neural network mapping sensor values to movement. However, Dorigo et al. used pre-specified metrics in their fitness function. These metrics were tweaked after undesirable behavior was evolved in initial trials (2004, p. 228). Deciding on a particular metric or two inherently limits possible evolved behaviors. While this is often intentional, as researchers choose metrics to drive evolution of a *desired* behavior, it can be difficult to determine a metric that allows for both novel controllers and favorable outcomes. In contrast, this work analyzes the development of an approach that can be applied to multiple behaviors without spending significant time deciding on which metrics to use.

More recent work in evolutionary robotics applied to swarms has involved evolving robots for the purpose of real life applications. For instance, Duarte et al. (2016) evolved homing, dispersion, clustering, and monitoring behaviors for aquatic surface robots. They then analyzed the performance of the controllers, which were evolved in simulation, in a real, unconstrained environment and connected the independently evolved behaviors to allow the robots to execute a proof of concept environmental monitoring task. The robots in this work were small boats intended to operate in an uncontrolled environment. A waypoint (a known point of interest) and a geo-fence were placed in the environment. For the purpose of sensing, robots divided the world into equally sized quadrants. Each robot had sensors that detected (1) the relative angle from a robot to the waypoint, (2) the distance from a robot to the waypoint, (3) the closest distance from the robot to another robot in each quadrant, and (4) the distance from the robot to the geo-fence in each quadrant. The robot controller was a neural network mapping inputs from these sensors to the robot's linear and angular speed. The authors optimized candidate controllers using NEAT, an algorithm that simultaneously optimizes the architecture and weights of a neural network. The authors created fitness functions for each of the four behaviors learned: homing, dispersion, clustering, and area monitoring. The homing fitness function calculated the average distance from robots in the swarm to the waypoint. The dispersion fitness function measured the average difference between the distance between a robot and its closest neighbor and a target value for this distance. The clustering fitness function measured the number of clusters (closely spaced groups of robots) formed by robots over the course of a trial, penalizing swarms for having more than one cluster. Finally, the area monitoring fitness function divided the arena in which the robots operated into cells, then measured the extent to which each of these cells was visited by a robot in the swarm over the course of the trial.

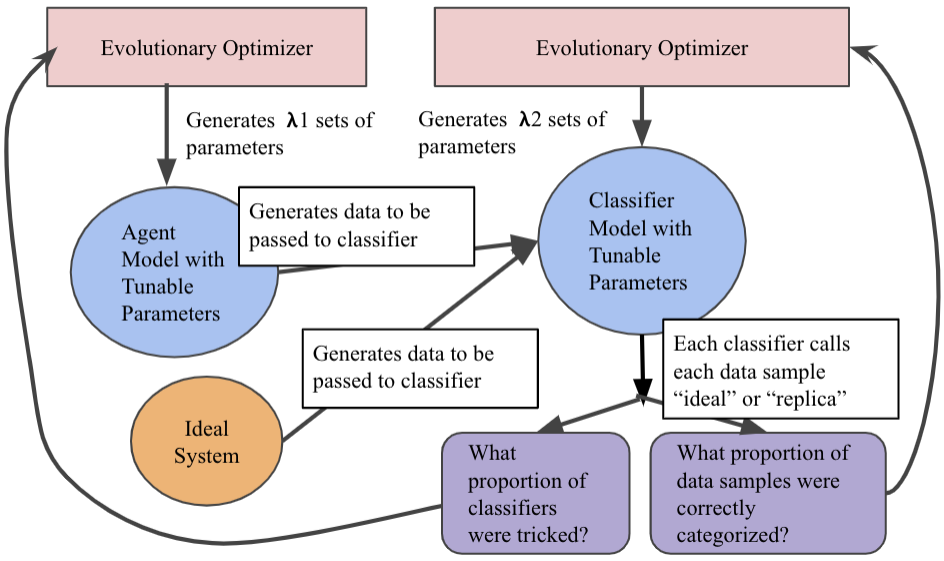


FIGURE 2.2: **Turing Learning**. Turing Learning is a co-evolutionary algorithm for evolving behavior that contains many more components than traditional evolutionary robotics systems.

In this paper, we work to evolve a dispersion behavior similar to the one specified in (Duarte et al., 2016). We imagine a similar eventual real-life application, and we use robots with similar sensing and movement capabilities, though we used a different optimization algorithm and different controller architecture. However, like in (Dorigo et al., 2004), Duarte et al. (2016) use directly specified metrics in fitness calculations. These metrics directly characterize the goal behavior in terms of the relationship between robots, including the distance between robots and the number of robot clusters (Duarte et al., 2016, pp. 9-10). Our work instead analyzes an evolutionary algorithm that does not require fully specifying metrics for a behavior. It thus is hypothetically useful even in situations where behavior is not well understood and one does not know which metrics may be appropriate.

2.2.2 Turing Learning

Turing Learning is a new approach that has thus far only been verified in a few scenarios. The original developers of the method proposed Turing Learning as "a metric-free approach to inferring behavior" (Li, Gauci, and Groß, 2016). In Turing Learning, users have an ideal system that executes a desired behavior. They use Turing Learning to evolve a model that mimics the ideal system. Turing Learning does not hand code a similarity measure between replica and ideal system.

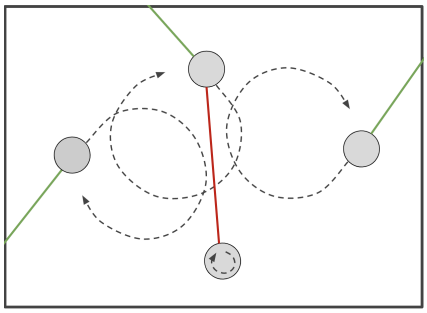


FIGURE 2.3: Ideal aggregation behavior inferred via Turing Learning in (Li, Gauci, and Groß, 2016). Agents have a single line of sight sensor through which they can detect obstacles. Agents move backwards in a clockwise circle unless there sensor is blocked, in which case they turn in place in a clockwise direction.

Instead, replica and ideal agents generate *data samples*, such as their trajectories in simulations. Classifiers attempt to distinguish between *genuine* data samples (those created by the ideal system) and *counterfeit* data samples (those created by replica systems). The replicas learn to generate better and better data samples over the course of an evolutionary run as classifiers evolve better and better ways to distinguish between the samples. "Metric-free" refers to the lack of hand-coded similarity metrics.

Turing Learning was originally verified on swarms: in their work, Li, Gauci, and Groß validated Turing Learning by inferring aggregation and object clustering behavior. This work builds upon the framework for Turing Learning proposed by Li, Gauci, and Groß. As in their paper, we simultaneously optimize a population of classifiers and a population of replica models, using the classifications provided by classifiers to calculate the fitness of candidate solutions in each population. The high level architecture of Turing Learning can be seen in Figure 2.2. In this work, we also use Turing Learning to infer swarm behavior; However, we examine a collective behavior that is significantly more complex than previously studied. Li, Gauci, and Groß inferred aggregation and object clustering behavior from agents executing computation-free behavior. Aggregation and object clustering had previously been shown to be swarm-level behaviors that could be executed without requiring agents to have memory or computation ability (Gauci et al., 2014b; Gauci et al., 2014a). In their computation-free manifestation, agents executing aggregation and object clustering have minimal sensing capabilities. We describe in detail here the algorithm for computation-free aggregation. The algorithm for object clustering is similar. Each agent has a single line of sight sensor with possible inputs `{BLOCKED, UNBLOCKED}`. Aggregation is a *reactive* behavior in which each of the two possible inputs maps directly to an agent's left, v_l, and right, v_r wheel speeds. These speeds lie in the range $[-1, 1]$. The controller for an agent can be represented as a list of wheel speeds corresponding to the inputs 0 for `UNBLOCKED` and 1 for `BLOCKED`. This is

denoted in (Li, Gauci, and Groß, 2016) as:

$$\mathbf{p} = (v_{l0}, v_{r0}, v_{l1}, v_{l1}) \tag{2.1}$$

Work by these authors had found the optimal values for $\mathbf{p}$ to be $(-0.7, -1.0, 1.0, -1.0)$. With these values, when a robot's sensor is unblocked, the robot moves backward along a clockwise circular trajectory. When the sensor is blocked, the robot turns in place in a clockwise direction, as shown in Figure 2.3. Robots executing aggregation thus will always be moving at one of two possible velocities.

The methodology for Turing Learning proposed by Li, Gauci, and Groß provided the classifier with data consisting of the linear speed, s, and angular speed, ω, of a single robot over a simulation. The authors used a recurrent neural network with 2 inputs, 5 hidden neurons, and 1 output neuron for their classifier. They optimized the replica and classifier populations via a $(\mu + \lambda)$ evolution strategy with self-adaptive mutation strengths. In most trials, the authors ran a simulation containing a swarm of 10 ideal agents and one replica agent. They also, however, demonstrated that aggregation could be inferred using swarms consisting only of ideal agents and only of replicas. Each classifier i considered the kth data sample from replica j or the lth ideal data sample and gave a classification output m_{ijk} or m_{il}. A classifier output of 1 meant the classifier believed the data sample to be genuine. Otherwise, the output was 0 and the data thought to be counterfeit. A replica's fitness was given as the proportion of classifiers it tricked into classifying its data samples as genuine. Classifier fitness averaged the proportion of genuine data samples correctly categorized as such and the proportion of counterfeit data samples correctly categorized as such. This system can be seen in Figure 2.4. We note that the authors also inferred behavior for replicas with an alternative controller. In this alternative, the controller was modeled as a recurrent neural network with 1 input (the sensor value), 1, 3 or 5 hidden nodes, and 2 outputs (the wheel speeds). Optimizing the replica in these trials consisted of optimizing weights for this neural network. These trials were all successful.

With Li, Gauci, and Groß's chosen artificial system from which to infer behavior, there was an implicit relationship between sensor input, wheel speed, and data provided to the classifier. The behavior to be learned in this work has not been shown to have the same implicit relationship between controller inputs and outputs and robot trajectories. We thus partially examine the extent to which this implicit relationship played a role in the prior success of Turing Learning in inferring swarm behavior and the ways in which the Turing Learning methodology may need to be modified for increased generalizability. In our work we vary components of the Turing Learning system to investigate what pieces of the system implemented in Li, Gauci, and Groß's work are generalizable, and which pieces need to be modified given an intended application.

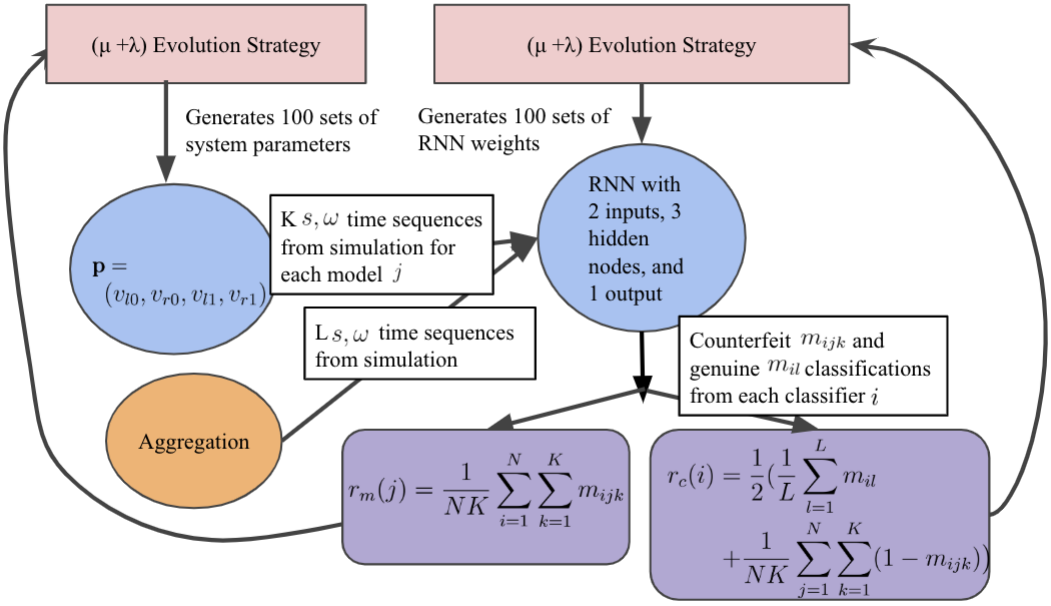


FIGURE 2.4: Implementation of Turing Learning used to infer aggregation in (Li, Gauci, and Groß, 2016)

To our knowledge, only one other work has attempted to apply Turing Learning to a new scenario. In a late November 2018 work, Zonta et al. applied Turing Learning to the problem of modelling human trajectories. They used Turing Learning to develop a generative model of possible human trajectories in crowded locations. In their system, the population of replica models was a set of generative networks that could predict the future location of a person given a limited sequence of that person's previous locations. Input to the classifier was modified only slightly from Li, Gauci, and Groß's initial paper. Zonta et al. fed the classifier bearing in addition to linear and angular speed. They ultimately found the original structure for Turing Learning failed to infer the desired behavior. Modifying the classifier architecture and fitness computation was required for the effective evolution of good generators (Zonta et al., 2018).

In this work, we examine Turing Learning in the field of swarm robotics. A replica model is executed on a local agent, but evaluated on the basis of the behavior of a group of robots. In (Zonta et al., 2018), the authors attempt to evolve a generator of human trajectories. Each replica generator operates in isolation, rather than in a collective. Small changes to agents in a collective can lead to very large changes in group behaviors, so replicas with similar parameters may actually have very different collective dynamics and thus different fitness values. In contrast,

small changes to replica generators in (Zonta et al., 2018) are more likely to lead to small changes in generator fitness. Thus, the evolution of collectives via Turing Learning in our work differs from the evolution of generators via Turing Learning in (Zonta et al., 2018). Despite the distinct areas of application, we encountered similar challenges in evolving high-quality classifiers and also found that modifying classifier fitness improved learning speed, suggesting these findings may be generally applicable.

2.3 Fish Schooling

We apply Turing Learning specifically to the inference of behavior of schools of fish. Fish schooling is particularly interesting as there exists a large body of work around it. Schooling is a particular form of flocking (coordinated group motion) behavior, and flocking has been studied from a variety of perspectives, including observational studies of real fish. In addition, recent work has investigated the possibility of inserting biomimics into real life schools of fish. Turing Learning provides a novel way to develop models of flocking and may eventually allow for the development of more perfect mimics than those currently in existence.

2.3.1 Flocking

Flocking has been investigated from a variety of perspectives, including graphics (Reynolds, 1987), particle dynamics (Vicsek et al., 1995), and biological systems (Couzin et al., 2005; Katz et al., 2011). Reynold's initial work on flocking demonstrated the success of local models of behavior in causing interesting global effects. The "boids" (abstract animated objects) in Reynold's simulations demonstrated realistic coordinated motion in large groups simply through the execution of local collision avoidance, velocity matching, and flock centering.

Subsequent work in (Vicsek et al., 1995) contributed a physics-based approach to flocking, finding that particles moving at constant velocity demonstrate a kinetic phase transition when they begin to assume the approximate average direction of their neighbors. Coordinated motion thus does not necessarily require intentional movement around neighbors: simply matching orientation is sufficient for the emergence of directed movement.

Coordination in biological systems requires more intentional movement than possible in the above models. It is not sufficient for fish to coordinate their movement in a *random* direction. Schools must escape predators and move to good locations for feeding. Environment cues influence the behavior of fish schools. Couzin et al. (2005) find that consensus in motion can still emerge even when individuals consider information from the environment in addition to the location of their neighbors. Further, leaders with more environmental information can influence a

school to move in a desirable direction without explicitly signaling other school members. The biological imperative was carried further in (Katz et al., 2011). Couzin et al. (2005) and Reynolds (1987) demonstrated models that executed realistic flocking behaviors but did not require that they be the models animals actually execute. Katz et al. explicitly inferred behavioral rules from experimental data, finding that the averaging of responses from neighbors, a common model, may not be entirely accurate, though it models the spatial interaction of agents in groups reasonably well. Instead, Katz et al found that three-way interactions may contribute to schooling dynamics and may be notable when schools rapidly change direction.

Our work advances a new technique to infer schooling behavior through observation of natural systems. In the future, this may lead to the development of new understandings of natural systems or novel mathematical models for artificial schooling. While we do not yet examine non-pairwise interactions, we anticipate that Turing Learning will be able to do so in the future, allowing for more systematic analysis of the three-body contribution posited in (Katz et al., 2011).

Models of schooling have recently been applied to robotic systems operating in real life conditions (Delight et al., 2016). Delight et al. formalize the models proposed in prior works into locally executable behaviors for a swarm of robotic boats intended to map the ocean surface. Their work includes the leader-follower and environmental dynamics ideas introduced in (Couzin et al., 2005) to create size-tunable mapping, environment-responsive migration, and leader-responsive migration behaviors in aquatic surface robots. Delight's model is particularly well suited to our intended robotic application. We have chosen to work to advance Turing Learning to flocking behavior by inferring some aspects of the model laid out in (Delight et al., 2016)

2.3.2 Imposter Fish

Turing Learning generates not simply a desired behavior, but rather a behavior that appears a perfect mimic of some ideal system. Many biologists are interested in developing robots that are socially accepted by a biological system. Such robots may allow scientists to determine causality of animal behaviors through the intentional actions of a robotic agent (Kim et al., 2018). Recent work examines the impact of biomimetic closed-loop control on zebrafish behavior (Kim et al., 2018; Cazenille et al., 2018). Biomimetic refers to situations in which synthetic processes mimic biological ones. Closed-loop control occurs when a system compares its desired outcome with its actual outcome to inform future behavior. Biomimetic closed-loop control thus refers to processes that attempt to mimic a biological system and use real-time feedback from the biological system to improve their mimicry. In (Kim et al., 2018) and (Cazenille et al., 2018), a robot is designed to visually approximate a live zebrafish and is placed in a tank containing both the robot and biological zebrafish. This robot is able to move throughout the tank with the assistance of an external

system. This external system tracks the biological fish in real time and uses this information to inform robot behavior. Both works currently depend on pre-specified metrics that determine how well the robot is being socially accepted by the biological fish. As a result, the robots will never become better mimics than the current measures of similarity can detect. Cazenille et al. (2018) in fact notes this as a limitation of the research. By analyzing current limitations of Turing Learning applied to schooling behavior, we advance its promise of metric-free system inference. In the future, this system could be used to achieve the promise of fully socially accepted robots in zebrafish groups.

3 Implementation

In this chapter, I present my implementation of Turing Learning for use in learning dispersion in artificial schools of fish. Turing Learning operates through the simultaneous optimization of a population of replicas and a population of classifiers. The classifiers are tasked with distinguishing between data samples created by genuine agents and data samples created by replicas. The replicas are tasked with "tricking" the classifiers into categorizing the counterfeit data samples as genuine. The ultimate goal is that, from an observer's perspective, the replicas replicate the behavior of the genuine agents. Algorithm 1 provides a description of Turing Learning implemented in this paper, modified from the algorithm proposed by Li, Gauci, and Groß (2016). Figure 3.1 contains the implementation in diagram form, and indicates the specific section where each component of the system is defined.

The first choice in this paper's expansion of Turing Learning was in the system from which to infer behavior. Ultimately, we hope to infer behavior for robotic fish from biological schools of fish. However, in this paper, we begin progress to this goal by inferring behavior of simulated schools of fish executing a known flocking algorithm. We further constrain our research to inferring behavior in two dimensions. This represents an important step towards the ultimate goal of learning from real fish: the behavior learned is much more complex than previous behavior inferred via Turing Learning. We thus are able to examine limitations of the current implementation of Turing Learning using a system from which we can obtain clear data samples with a much lower cost than a natural, three dimensional system. The exact specification of our ideal system can be found in Section 3.1.

Turing Learning evolves two populations: replicas and classifiers. The architecture for replicas and classifiers both must contain parameters that can be optimized through evolution. We specify the architecture of replicas in 3.2.1 and the architecture of classifiers in 3.2.2. The classifications given by this model provide the basis for fitness computation. We describe the general framework of the fitness functions used to optimize each population in 3.2.3. Finally, in 3.2.4 we provide a brief description of the algorithm that optimizes these parameters and provides candidate solutions in each generation.

Algorithm 1 Turing Learning Procedure

Initialize population of M replicas and population of N classifiers
while termination criterion not met **do**
  obtain data samples from ideal agents
  for all replicas $i \in \{1, 2, ..., M\}$ **do**
    obtain counterfeit data samples from replica i
  end for
  for all classifiers $j \in \{1, 2, ..., N\}$ **do**
    for each data sample, obtain and store output of classifier j
  end for
  reward replicas (r_m) for misleading classifiers (classifier outputs)
  reward classifiers (r_c) for making correct judgments (classifier outputs)
  improve replica and classifier populations based on r_m and r_c
end while

3.1 Artificial System from Which to Infer Behavior

Our long-term goal is the creation of a robotic school of fish that mimics real schools of fish. Such a robotic school could be released into a marine environment for monitoring purposes with minimal disruptive effect on the natural ecosystem. We have focused on designing the behavior of this collective after recent progress in the design of the physical robot for this task (Berlinger et al., 2018). In this work, we infer dispersion, a behavior characteristic of fish schooling. Although dispersion is only a small component of the overall dynamics of schooling fish, it is a significantly more complex behavior than those previously inferred with Turing Learning. Thus, inferring dispersion is an important step towards our ultimate goal of inferring schooling behavior from biological fish. Although we work entirely in simulation in this paper, our intended robotic test bed informs our simulation design.

3.1.1 Algorithm for Swarm Behavior: Dispersion

Coordinated motion depends on a collective's ability to maintain internal coherency. Individual agents can neither approach too close to each other, which would lead to crashing, nor drift too far apart, which would lead to splitting of the group. *Dispersion* is the behavior in which agents in a group move away from each other. *Aggregation* is when agents move toward each other. Dispersion and aggregation are important not only in biologically-inspired mimicry, but also in robotic collectives executing purely artificial tasks. For example, in an aquatic collective, robotic fish intended to monitor an area could use dispersion to expand a sampling radius and then at conclusion of a task aggregate to a single pick-up point.

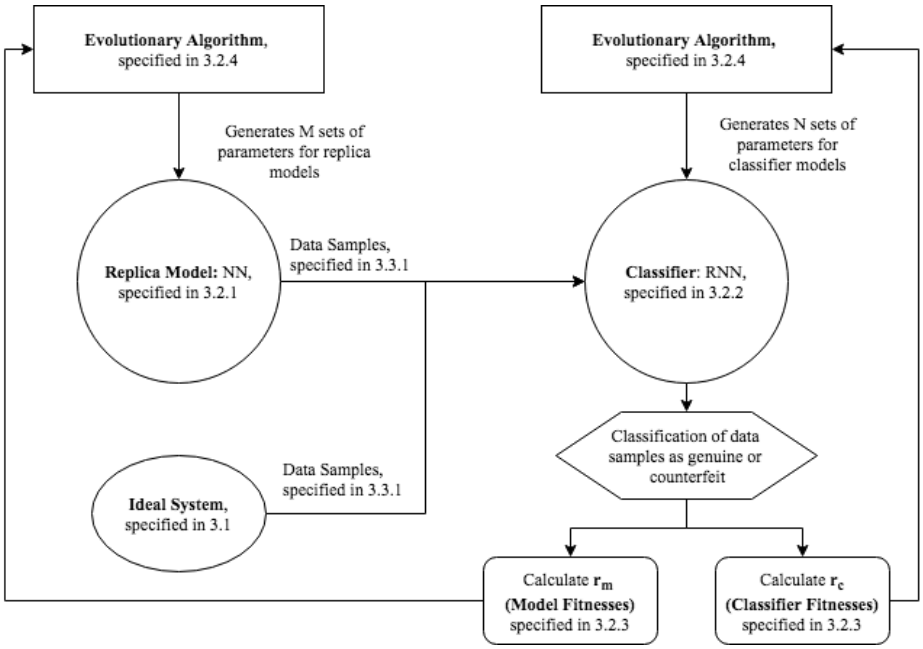


FIGURE 3.1: Turing Learning Procedure

This paper focuses particularly on dispersion to a pre-specified distance. In this behavior, agents move away from each other until the distance between agents is consistent with the user-specified distance. In light of our intended test bed, we use the term *fish* (pl. fishes) interchangeably with the term *agent*. Delight et al. (2016) recently developed a model for a boat collective that included a single mathematical model of behavior for both aggregation and dispersion. The initial spread of agents determines whether the agents aggregate or disperse. Our ideal system is based off this model.

In this system, fish are modeled as points in space. Each fish has a limited sensing radius and can only detect other agents within this radius. We decide on an equilibrium distance σ. Fishes move away from neighbors closer than σ and towards neighbors farther than σ. When fishes are initially randomly distributed in a radius $r < \sigma$, the fishes will have more neighbors within σ than outside of σ, and so will move apart from each other until this is no longer the case. A visualization of this behavior can be seen in Figure 3.2. Below, we formally define the functions governing this behavior. All functions were originally defined in (Delight et al., 2016). We modify the notation and formulation of the functions only slightly for our particular application.

A fish i is capable of determining the relative position $\vec{r}_{ij}$ of a neighbor j. That is, a fish can calculate a vector from itself to its neighbor. A fish j is termed a *neighbor* of i if j is within the sensing radius r_{comm} of i. The *neighborhood*, Ω_i, of i is the set of all neighbors of i. A fish i's position

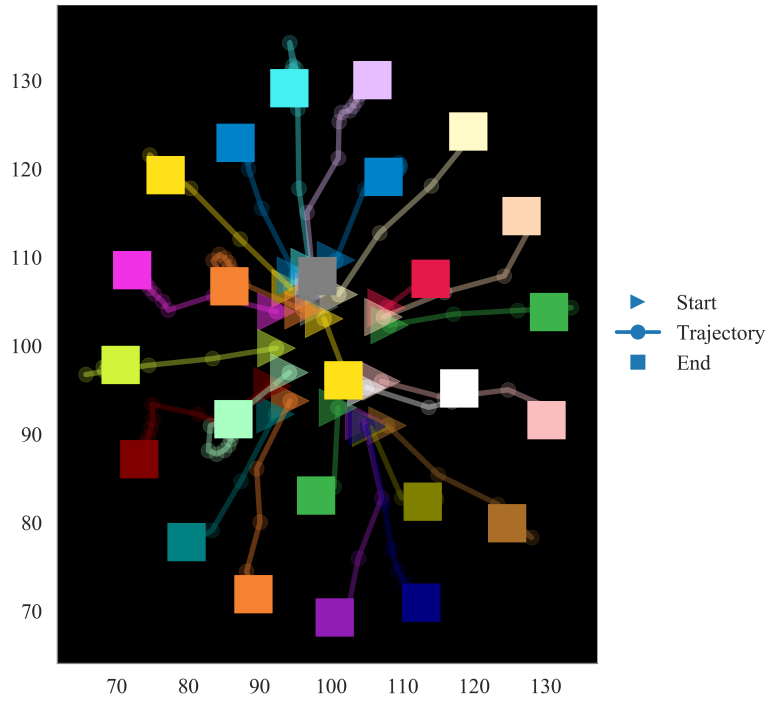


FIGURE 3.2: Here we present an image of *dispersion*. Each color represents a single agent, which we term a *fish*. A fish begins at the point marked with a triangle, and ends at the point marked with a square. Each point on the line connecting triangle and square is a single time step. The triangles are much more tightly packed than the squares, hence the term dispersion.

in the XY plane at time t is denoted as the vector $\vec{x}_i(t)$. We thus define relative position and the neighborhood of i as follows:

$$\begin{aligned} \vec{r}_{ij}(t) &= \vec{x}_j(t) - \vec{x}_i(t) \\ \Omega_i(t) &= \{j \mid \|\vec{r}_{ij}(t)\| \leq r_{comm}\} \end{aligned} \tag{3.1}$$

We model the position and velocity of a fish as a discrete time-process. That is, the velocity of a fish is updated at a specific interval, and the fish moves at a constant velocity between updates. This models the capabilities of robots well. They have limited computation ability and so limiting the processing of inputs to specific intervals aids in maximizing use of that computation ability. We model the interaction between a fish i and each of its neighbors independently, and sum the corresponding vectors to determine the final velocity for fish i. Formally, a fish's position ($\vec{x}_i(t)$)

and velocity ($\vec{v}_i(t)$) over time are:

$$\begin{aligned} \vec{v}_i(t) &= \sum_{j \in \Omega_i(t)} \vec{f}(\vec{r}_{ij}(t)) \\ \vec{x}_i(t + \Delta t) &= \vec{x}_i(t) + \vec{v}_i(t)\Delta t \end{aligned} \tag{3.2}$$

Previous work with this model used three components to compute the time evolution function f: cohesive attraction, separation-based attraction-repulsion, and environment-based migration (Delight et al., 2016). We were interested only in aggregation and dispersion, and so only considered the attraction-repulsion force. The strength of the attraction-repulsion force is mediated by a user-determined constant, k_{ar}. The time evolution function f is thus formally given as follows:

$$\vec{f}(\vec{r}_{ij}(t)) = \vec{f}_{ij}(t) = k_{ar} sgn(\|\vec{r}_{ij}\| - \sigma)(\|\vec{r}_{ij}\| - \sigma)^2 \frac{\vec{r}_{ij}(t)}{\|\vec{r}_{ij}(t)\|} \tag{3.3}$$

3.1.2 Selection of Constants Based on Physical Robot Capabilities

This model makes many simplifying assumptions. Robots are assumed to be points in space that cannot crash. They do not have an orientation. The relative position of a neighbor is calculated as if all robots face the same direction. We also assume robots can move in any direction. We assume there is no noise in sensing or movement. Robots see exactly the true positions of their neighbors and move exactly where they intend. Robots "sense" a list of the positions of their neighbors, rather than a camera image they must transform into neighbor position. Further, we examine this behavior in only two dimensions. These assumptions are not realistic and would prevent the direct translation of learned behavior into actual robots. However, the assumptions enormously simplify computation, and so have been deemed acceptable for this work. After validating Turing Learning in this simple scenario, future work can be done in three dimensions with more realistic physics and robot capabilities.

The chosen behavioral model requires consideration of a series of constants: an update period Δt, a communication radius r_{comm}, a defined equilibrium radius σ, at which distance neighboring agents execute no force, and a magnitude constant k_{ar}. Further, the model naively assumes no maximum speed, yet robots have a maximum speed v_{max}.

This work decided on constants through experimental parameter optimization and consideration of the intended test bed. The intended test bed is a collective comprised of a small, low-cost aquatic, fish-like robot being developed in the Self-Organizing Systems Research Group at Harvard (Berlinger et al., 2018). This robot is approximately 100 mm long x 25 mm wide, can achieve

a velocity of 0.6 BL/s (body lengths/s), and is operated in a 1.78 m x 1.78 m test bed (considering the XY plane only). Ideally, robots in this collective could be dropped in a tank close together, then the robots could independently disperse to avoid collisions prior to beginning a more complex task. The robot in (Berlinger et al., 2018) was operated alone in a tank and had only a very simple sensor. It would not have been able to detect the position of neighbors if it had had any. We have thus made assumptions regarding capabilities that will be added in order for this robot to operate in a school. In particular, we have assumed the ability to detect and determine the position of neighbors up to 1 m away, approximately equivalent to 10 BL. We also assumed the robot's max speed will increase to 0.9 BL/s. It is further important to note that the robot's small size partially limits its ability to carry significant computing power, making less frequent velocity updating preferable.

Given this intended future test bed, we decided on the following constants, implicitly equating our abstract units to centimeters:

$$\begin{aligned} \Delta t &= 1 \\ \sigma &= 40 \\ r_{comm} &= 100 \\ \mathrm{v}_{max} &= 9 \end{aligned}$$

We decided to execute behavior on a swarm of 25 agents in all trials presented in this work. This is approximately the size of the largest swarms evaluated in (Delight et al., 2016). We decided on a value of 0.003 for k_{ar} through experimental validation. This value was chosen such that when executing dispersion, agents dispersed quickly until reaching an equilibrium at which point agents were stable and movement ceased. We characterize swarm behavior via the use of two metrics: *mean neighbor distance* and *mean swarm speed*. The mean neighbor distance refers to the average distance, across all pairs, between a robot and one of its detected neighbors. Mean swarm speed averages the speed of all robots in a swarm. We decided on k_{ar} by surveying these metrics for robots initially randomly placed in a circle with a diameter of 20 units and an equilibrium distance $\sigma = 40$. We ran trials for 15 s. A good value of k_{ar} is one in which the robots move away from each other initially, then the distance remains constant and the speed reaches 0, as forces on each robot are stabilized. The behavioral model is such that an equilibrium neighbor distance is not necessarily equal to σ: if a robot has more distant neighbors than close ones, attractive forces may initially outweigh repulsive forces, increasing the density of the collectives until the proximity of neighbors less than σ away outweighs forces created by neighbors more than σ away. We tested 5 values of k_{ar}. The results of these trials can be an be found in Figure 3.3.

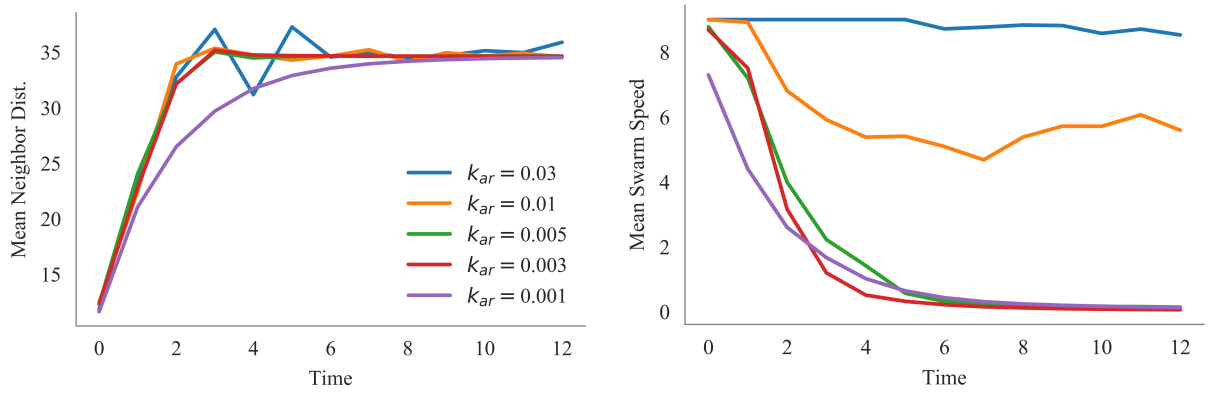


FIGURE 3.3: We chose the largest value of k_{ar} that reached a stable neighbor spacing. That value was 0.003, in red in the diagrams above. With this value, the swarm quickly reaches equilibrium, where the average swarm speed is 0. Trials ran for $t = 15$s.

3.2 Design of Component Parts of Turing Learning

3.2.1 Architecture of Replica Model

We assume replicas have the same movement and sensing capabilities as ideal agents. A replica fish can thus detect the relative positions of neighbors inside its sensing radius $r_{comm} = 100$. Further, a replica agent i's position and velocity is also given by the discrete time-processes defined in Equations 3.1 and 3.2. That is, the replica's velocity is assumed to be a summation across all neighbors of a function operating on the relative position of a single neighbor.

The replica's time evolution function f is **not** given by Equation 3.3. Instead, the time evolution function $f_{replica}$ is modelled using a neural network (NN). We decided on a neural network architecture of a multi-layer perceptron with 2 inputs, 3 hidden nodes, and 3 output nodes, a logistic sigmoid nonlinear activation function, and a bias for each hidden and output node. The logistic sigmoid activation function has the range $(0, 1)$ and is defined as:

$$\text{sig}(x) = \frac{1}{1 + e^{-x}} \tag{3.4}$$

The connection weights of this perceptron are optimized through the evolution process. The input to this perceptron for an agent i given a neighbor j are $\vec{r}_{ij} = (r_{ijx}, r_{ijy})$: the relative x and y location of neighbor j according to agent i. The inputs r_{ijx} and r_{ijy} are normalized to lie in the range $[0, 1]$ by dividing raw sensor position data by $r_{comm} = 100$. The output of this perceptron is j's contribution to i's current velocity, decomposed into x direction, y direction and speed components, such that

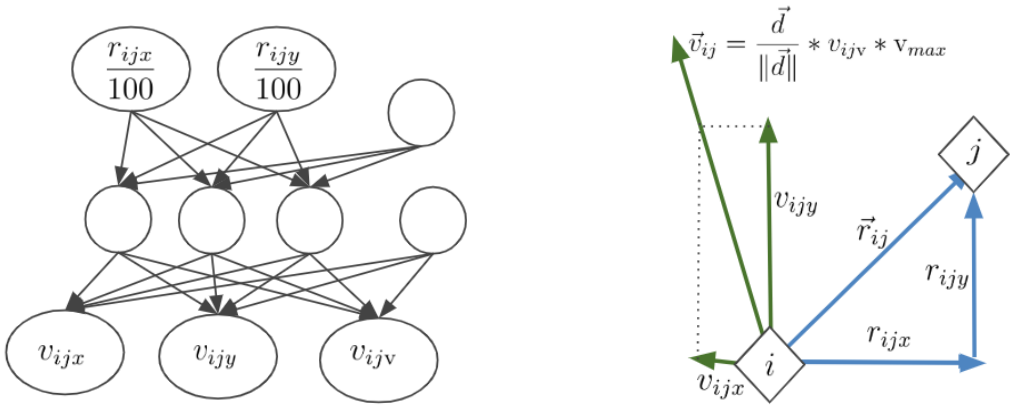


FIGURE 3.4: (a) presents $f_{replica}$, a multi-layer perception with 2 inputs, 3 hidden nodes, 3 output nodes, and a bias for each layer. The perceptron maps neighbor j's relative position to a contribution to i's velocity. In (b), we diagram the input and output values of the perceptron in an imagined scenario.

$\vec{v}_{ij} = (v_{ijx}, v_{ijy}, v_{ijv})$. We constrain all outputs to the range (0, 1) through appropriate use of our nonlinear activation function. We can then recombine the outputs to form a velocity vector such that the fish's speed must lie in the range $(0, v_{max})$. Formally, this recombination is as below and is shown in Figure 3.4.

$$\begin{aligned} x &= v_{ijx} - 0.5 \\ y &= v_{ijy} - 0.5 \\ \vec{d} &= (x, y) \\ \vec{v}_{ij} &= \frac{\vec{d}}{\|\vec{d}\|} * v_{ijv} * v_{max} \end{aligned} \tag{3.5}$$

3.2.2 Architecture of Classifier

Our classifier needs to be capable of discriminating between genuine and counterfeit data samples, those created by ideal agents and replica agents, respectively. Data samples are created by tracking information about agents over the course of a simulation. These samples are time sequences. Each step in the time sequence contains a number of measurements of some aspect of the system at that time step.

The classifier is an Elman neural network, a particular type of recurrent neural network (RNN) (Elman, 1990). In this network, the hidden layer is connected to a context layer which stores

information from previous inputs, making this network well suited for our time sequence data. The classifier examines data from a single time step simultaneously and can hold "memory" in its context layer about past time steps. We thus feed in an entire data sample from a single trial and then reset the context layer. The classifier has i inputs (the number of measurements taken regarding a system at a single time step) and h hidden layers. We have varied both i and h in our research as we formulate new data samples to be used as input to the classifier. In all cases, the classifier has a single output neuron. We add a bias to every non-input neuron and use the same logistic sigmoid nonlinear activation function for hidden and output neurons (function given in Equation 3.4). A data sample from a trial of T time steps is a T x i matrix D. Each row of the matrix contains data from one time step. The classifier considers each time step sequentially, and the hidden layer is reset for each full data sample D.

We denote the output of a classifier c after t time steps of data sample D as $O_c(D_t)$. The sequence of outputs $\{O_c(D_{t_1}), ...O_c(D_{t_f})\}$ from a data sample D is denoted $\vec{O}_c(D)$. We also define the *judgement* of a classifier as the final decision of the classifier on a data sample being genuine or formal. Formally, the judgement J of classifier c on a data sample D is given as follows:

$$J_c(D) = \begin{cases} 1 & \text{if } O_c(D_{t_f}) >= 0.5 \\ 0 & \text{otherwise} \end{cases} \tag{3.6}$$

A judgement of 1 means the classifier believes the data sample to be a genuine data sample from an ideal agent or ideal swarm and a judgement of 0 means the classifier believes the data sample to be a counterfeit data sample from a replica agent or replica swarm.

3.2.3 Computation of Fitness for Candidate Solutions in both Replicas and Classifiers

The quality of candidate replicas and candidate classifiers is determined by their relative performance against the other population. This competitive manner of fitness calculation is termed subjective fitness. The functions below are formulated according to a population of M replicas and a population of N classifiers in evolution.

Replica Fitness

Intuitively, we wish to give higher fitness scores to replicas that successfully trick classifiers into categorizing their counterfeit data samples as genuine data samples. Each replica creates K data samples. In its most general formulation, the fitness of a replica is the mean over all data samples and all classifiers of some function f_{model} of the outputs of a single classifier. We denote the kth data sample generated by a replica m as D_{mk}. The fitness r_m of replica m can thus be given generally as

follows:

$$r_m = \frac{1}{NK} \sum_{c=1}^{N} \sum_{k=1}^{K} f_{model}(\vec{O}_c(D_{mk})) \tag{3.7}$$

In this work, we consider only the final judgement of classifier when calculating replica fitness, as opposed to the entire set of outputs. As a judgement of 1 indicates the classifier believes a data sample to be genuine, we can simply average the judgement of all classifiers on all samples created by a replica model to calculate its quality, r_m:

$$r_m = \frac{1}{NK} \sum_{c=1}^{N} \sum_{k=1}^{K} J_c(D_{mk}) \tag{3.8}$$

Scoring classifiers is slightly more complex. We wish classifiers to correctly mark genuine data samples as genuine, and counterfeit data samples as counterfeit. We thus separately score a classifier on its ability to recognize genuine samples and on its ability to recognize counterfeit samples, and combine these scores via a scoring function g. We have L total genuine data samples, and denote the lth genuine data sample as D_l. To calculate the counterfeit score and to calculate the genuine score, we average a scoring function of the output of a classifier over all counterfeit or genuine data samples. Formally, we define the fitness r_c of classifier c:

$$\text{counterfeit}_c = \frac{1}{MK} \sum_{m=1}^{M} \sum_{k=1}^{K} f_{counterfeit}(\vec{O}_c(D_{mk}) \tag{3.9}$$

$$\text{genuine}_c = \frac{1}{L} \sum_{l=1}^{L} f_{genuine}(\vec{O}_c(D_l)) \tag{3.10}$$

$$r_c = g(\text{counterfeit}_c, \text{genuine}_c) \tag{3.11}$$

We evaluated multiple possibilities for $f_{counterfeit}$, $f_{genuine}$ and g. The evaluation of these functions can be found in 4.3.4.

3.2.4 Algorithm for Optimization of Replica and Classifier Parameters

We use the covariance matrix adaption evolution strategy (CMA-ES) algorithm to optimize our replicas and classifiers (Hansen, Müller, and Koumoutsakos, 2003). This algorithm assumes a dependent relation between values optimized by the algorithm and attempts to learn this relationship to more quickly optimize the learned values. Evolutionary algorithms such as CMA-ES determine the manner in which the fitnesses of a population of candidate solutions inform the population in the next generation. Our optimization processes for classifiers and replicas are run independently. A set of candidate replicas is generated by one evolutionary process. A set of

candidate classifiers generated by a second process. However, as the fitness of each candidate solution is determined by the fitness functions given previously, this subjective fitness implicitly links the two processes.

This work uses an existing Python implementation of CMA-ES (Hansen, Akimoto, and Baudis, 2019). This implementation allows the user to request the current set of candidate solutions, and then provide the fitness of each candidate solution. A candidate solution from CMA-ES is a list of numbers which we translate into weights for our replica neural network or classifier recurrent neural network. This implementation assumes the use of an objective function, rather than a fitness function: objective functions are generally minimized, while fitness functions are generally maximized. To utilize CMA-ES to maximize candidate solution fitness, we simply provide to the optimizer the negation of the fitness of a candidate solution, ie $-r_c$ or $-r_m$. CMA-ES internally determines parameters specific to the optimization process. Users are only asked to supply λ, the size of the population of candidate solutions in each generation, $\vec{x}_0$, the initial mean of each value in the set of values comprising a candidate solution, and σ_o, the initial variance of these values. We varied each of these values during the course of our work. Each experimental trial specifies the evolution parameters used in that trial.

It is useful to note here that the co-evolutionary strategy inherent in Turing Learning has led to symmetrical optimization. Attempting to *minimize* the fitness of models and classifiers *also* drives competition. However, if we attempt to minimize the fitness instead of the negation of the fitness, we implicitly switch the meaning of classification values. When minimizing fitness, $J(D) = 1$ would imply recognizing a counterfeit data sample, the opposite meaning of that which we define.

3.3 Simulation Platform

We use the `BlueSim` platform, a simulator designed explicitly to work with the robotic platform to which we plan to transfer learned behavior (Berlinger and Lekschas, 2018). It models communication, neighbor sensing, environmental influence, and motion of robotic fish in 2 dimensions. Robots are modelled as points in space with maximum speed and sensing distance. Robot's orientations are not tracked nor is turning speed considered in this simulator. Robots can thus be considered as having a constant orientation and the ability to move in all directions. The robots move in an unbounded arena. There is no noise added to sensor values or robot movement. The robot's control cycle is updated every 1 s, and locations of all robots are tracked every update cycle. The simulator uses a multi-threaded approach, with each fish operating on its own thread. This aspect of the simulator significantly increases the computational power required by a single

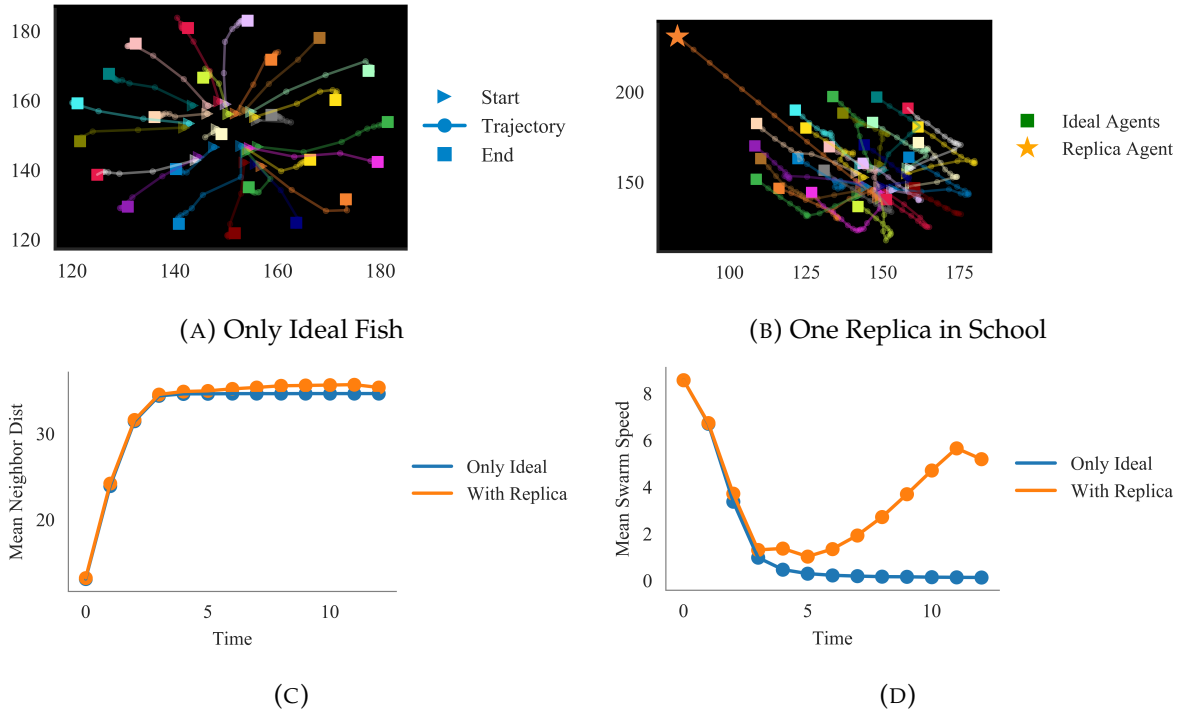


FIGURE 3.5: (A) shows a simulation of dispersion using 25 ideal fish, while (B) shows a simulation with 24 ideal fish and 1 untrained replica. We also compare the mean neighbor distance (C) and mean swarm speed (D) of schools with and without a replica. The school containing a replica executes qualitatively different behavior than the ideal-only school and has a notably different mean swarm speed at the conclusion of the simulation.

simulation, limiting the number of possible concurrent simulations a given computer can handle.

3.3.1 Data Collection: Homogeneous Swarms

We adapted the simulator in (Berlinger and Lekschas, 2018) to allow for the interaction of non-homogeneous agents. Li, Gauci, and Groß (2016) primarily simulated and collected data samples from agents in mixed swarms of ideal and replica agents. However, initial testing revealed that the behavior of ideal agents was significantly influenced by the inclusion of a replica as is shown in Figure 3.5. We thus chose to run simulations in which the school being simulated consisted entirely of ideal agents or entirely of copies of a single replica.

Running simulations comprised of a single replica also allowed for the generation of swarm-based data samples. Previous work in Turing Learning had had each agent in a swarm generate

an independent data sample consisting of its linear and angular speeds over the course of a simulation (Li, Gauci, and Groß, 2016). Classifiers had to determine the origin of this data sample without evaluating data samples created by other agents. Often in swarm robotics, however, it is important to be aware of swarm level characteristics. When all agents in a swarm execute the same behavioral model, it is reasonable to attribute some macro-level swarm behavior to that model.

4 Results and Discussion

In this chapter we present and discuss our results. The chapter is organized as follows:

Section 4.1 naively applies the previous implementation of Turing Learning to fish schooling. This implementation fails to infer dispersion, but the behavior that is learned reveals a limitation of Turing Learning as previously implemented: it cannot infer spatial relationships between agents. This finding motivates subsequent sections in which we decompose Turing Learning into an analysis of replicas and an analysis of classifiers. These components are the most important pieces of Turing Learning. If the architecture of either is insufficiently expressive to achieve its designated task, Turing Learning will fail.

Section 4.2 analyzes the architecture of replicas outside of the context of Turing Learning. Examining replica architecture independently ensures that the replica, and not another component of Turing Learning, leads to a given set of results. We use traditional evolution with objective fitness functions and deep learning with backprogation to assess the ability of the replica to achieve dispersion. The results from this section motivate the formulation of three new data samples to be used in Turing Learning. These data samples are defined at the conclusion of the section (4.2.4).

We continue in Section 4.3 with an analysis of classifiers. Classifiers must be able to distinguish between genuine and counterfeit data samples. To independently evaluate classifiers, we task classifiers with distinguishing between aggregation and dispersion data samples from the ideal system. In this section, we evaluate the infrastructure around the classifier in addition to the architecture of the classifier itself, examining how the choice of data sample, choice of evolution parameters, and classifier fitness function impact evolution of high-quality classifiers on the dispersion vs. aggregation task. We validate the data samples presented in the previous section, and propose and validate alternative classifier fitness functions (4.3.4). The data samples and fitness functions are novel modifications to the original implementation of Turing Learning.

We conclude the chapter in Section 4.4 by demonstrating preliminary successful results of a full Turing Learning trial to infer dispersion. This trial uses one of our novel data samples and one of our classifier fitness functions. Experiments validating component pieces of Turing Learning comprised the majority of research work. At the conclusion of that work, there were little time left to analyze the whole system. As a result, while these results are encouraging, the impacts of our proposed modifications on the whole system have not been rigorously analyzed.

4.1 Direct Application of Previous Implementation of Turing Learning to Dispersion in Schools of Fish

The success of Turing Learning in (Li, Gauci, and Groß, 2016) led us to expect that directly applying Turing Learning to dispersion behavior would be relatively straightforward. Turing Learning had previously been used to infer aggregation, a behavior that on the surface appears similar to dispersion. However, we discovered that the aggregation behavior of the ideal system in (Li, Gauci, and Groß, 2016) was actually significantly simpler than the dispersion behavior studied here. We begin by laying out the details of a direct implementation of Turing Learning and provide an explanation for why this implementation fails to learn the desired behavior.

4.1.1 Definition of Data Sample and Fitness Computation

Following the original work on Turing Learning, we tracked the angular and linear speeds of an agent at each time step of a simulation. We ran simulations for 15s and collected data beginning at time $t = 2$. Each agent thus created a 14 x 2 data sample D for a total of $K = 25$ data samples generated by each replica. For a swarm size of 25 and a population of M models we thus had $25 * M = 25M$ counterfeit data samples. We ran a single simulation of ideal agents, which generated of total of $L = 25$ genuine data samples. This gave us a total of $25(M + 1)$ unique data samples to be independently classified by the population of classifiers. Because each time step of the data sample contains two measurements, the current linear and angular speed of an agent, classifiers had $i = 2$ inputs and 1 output node. We used the same number of hidden nodes, $h = 5$, as in (Li, Gauci, and Groß, 2016), giving us a total of 46 parameters to optimize.

We fully define here the functions $f_{genuine}$, $f_{counterfeit}$ and g used in this trial. $f_{counterfeit}$ and $f_{genuine}$ specify a score for the classifier on its ability to identify counterfeit and genuine data samples, respectively, while g describes how to combine these scores into a single overall score. In this section, we simply averaged the two scores to determine the final quality of the classifier, r_c:

$$\begin{aligned} r_c &= g(\text{genuine}_c, \text{counterfeit}_c) \\ &= \frac{1}{2}(\text{genuine}_c + \text{counterfeit}_c) \end{aligned} \tag{4.1}$$

Following again from (Li, Gauci, and Groß, 2016), we used the specificity and sensitivity of the classifier to calculate $f_{genuine}$ and $f_{counterfeit}$, respectively. The *specificity* of a classifier refers to the fraction of genuine data samples it correctly categorizes, while the *sensitivity* of the a classifier refers to the fraction of counterfeit data samples it correctly categorizes. Formally, we define

$f_{genuine}$ to be the *specificity* of classifier c as follows:

$$\begin{aligned} \text{genuine}_c &= \frac{1}{L}\sum_{l=1}^{L} f_{genuine}(\vec{O}_c(D_l)) \\ &= \text{specificity}_c \\ &= \frac{1}{L}\sum_{l=1}^{L} J_c(D_l) \end{aligned} \tag{4.2}$$

We define $f_{counterfeit}$ to be the *sensitivity* of the classifiers. The classifier receives a higher score for a judgement of 0, which signifies correct classification of the data sample as counterfeit. Formally, this is defined as:

$$\begin{aligned} \text{counterfeit}_c =& \frac{1}{MK}\sum_{m=1}^{M}\sum_{k=1}^{K} f_{counterfeit}(\vec{O}_c(D_{mk}) \\ &= \text{sensitivity}_c \\ &= \frac{1}{MK}\sum_{m=1}^{M}\sum_{k=1}^{K}(1 - J_c(D_{mk})) \end{aligned} \tag{4.3}$$

Evolution Parameters

The CMA-ES package used works best when σ_0 is chosen such that solutions lie within $3\sigma_0$ of $\vec{x}_0$. As our parameters correspond to the weights of neural networks and recurrent neural networks, we selected $\vec{x}_0 = \vec{0}$ (all values were centered around 0) and $\sigma_0 = 1$ for both the optimization of classifiers and the optimization of replica models.

In preliminary trials, the system learned replica neural network weights that caused integer overflow. We thus constrained parameters for both replicas and classifiers to $(-10, 10)$ using a bounding method provided by the CMA-ES implementation. Documentation for this library implies the boundary constraints are implemented by mapping unbounded learned weights evenly throughout the designated range (Hansen, 2011).

Experimental Set Up

We worked with a population of $M = 100$ candidate replicas and $N = 100$ candidate classifiers. We terminated evolution at 200 generations. This was chosen due to computation limits. 200 generations of evolution took approximately 24 hours even when using a 16 virtual CPU AWS EC2 server and parallelizing the simulations required by each generation. We analyze the replica model that had the highest fitness score in the 200th generation.

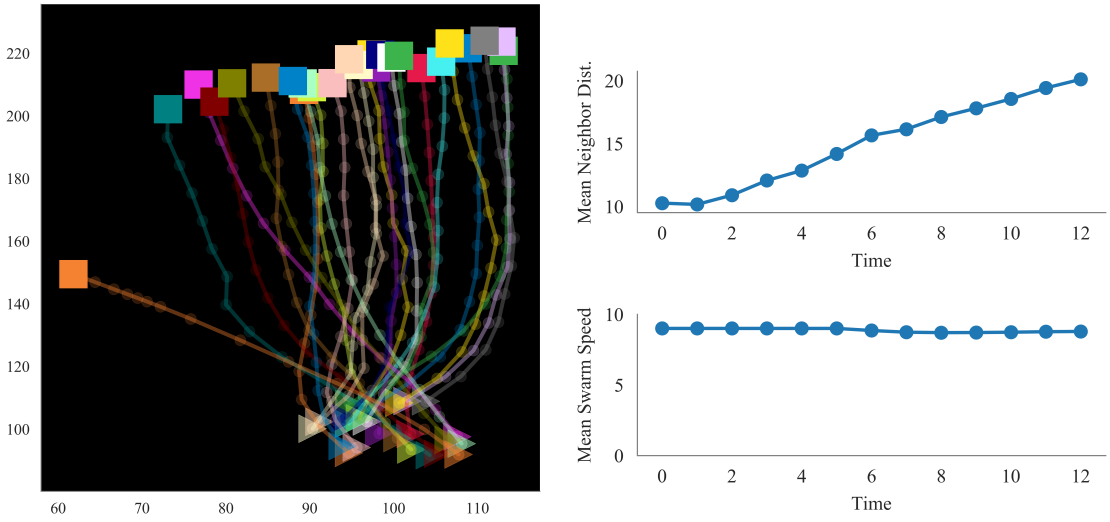


FIGURE 4.1: Directly applying Turing Learning does not lead to the inference of dispersion.

4.1.2 Analysis of Swarm-Level Learned Behavior from Direct Application

We found that although the replica system showed evidence of learning, it did not learn dispersion. Instead, in a school consisting of copies of the highest-quality replica evolved, all fish move side-by-side towards the top of the screen. Figure 4.1 visualizes the behavior of a swarm of replica fish with the highest subjective controller after 200 generations. We can see both visually and through the neighbor distance and swarm speed metrics that the correct behavior was not learned. In an ideal swarm, the fish reach an approximate mean neighbor distance of 35 and mean swarm speed of 0 by the 5th time step. These fish have a slow growing neighbor distance and a consistent average swarm speed of 9, which is the maximum agent speed. One fish (in orange) appears to follow a notably different path than the other fish. The fish all are running the same controller so we are not sure why this occurs.

The similarity of fish trajectories in the swarm in Figure 4.1 in fact explains a likely reason for the failure of direct application. In this trial, a classifier is given the trajectory information of a single fish in the form of a time series of linear and angular speeds. In dispersion, all fish have similar straight line trajectories. However, these trajectories are directed symmetrically outward from the center of the swarm. In contrast, the replica fish in Figure 4.1 execute straight line trajectories side-by-side, and do not consider the spatial arrangement of their travels. We thus need to expand the data passed to the classifier such that the classifier is granted some measure of the spatial context of an agent in the swarm.

We further found that the classifier population after 200 generations had a mean fitness score of 0.72, while the replica models had a mean fitness score of only 0.18 (note all fitness scores will be between 0 and 1, with 1 being the maximum possible fitness). This vast disparity suggests the coevolutionary algorithm has decoupled: it is relatively too easy for the classifier to detect the fish, and too hard for the fish to evolve behavior that can trick the classifier. We thus examine the replica model and classifier model independently to determine their capability of evolving an effective solution to the given task.

4.2 Decomposition of Component Pieces: Replica Model Analysis

We begin our system decomposition by examining replica model architecture. We wish to ensure that the architecture is sufficiently complex to achieve the desired behavior. If the behavior can be evolved using an objective fitness function, it follows that the replica model is sufficiently expressive, and the problem instead lies in the expressivity of the classifiers, the data passed to classifiers, or in the competition dynamics between the classifiers and the models. Further, we analyze these objective fitness functions to motivate new formulations for data samples to pass to the classifiers.

4.2.1 Objective Approaches to Developing High-Quality Replicas

In this section, we attempt to independently evolve replicas capable of dispersion. This evolution is much simpler than the co-evolution approach of Turing Learning. We need only a single optimization process, a singe set of candidate solutions, and an objective fitness function. An *objective* fitness function calculates the quality of a replica independently, rather than in comparison with a second population. This objective fitness function will measure how well a replica achieves a pre-specified goal. In formulating this function, we can directly state the factors we believe to be important to dispersion.

We found that Turing Learning implemented naively did not work because it did not consider the spatial arrangement of fish in a school. Using a direct evolution approach allows to ensure the objective fitness functions *do* consider the relation between agents in a swarm. This allows us to investigate if the desired behavior can be evolved using our replica architecture when the spatial organization of a swarm is considered.

In formulating these functions, we diverge from the terminology of *fitness functions*, and instead discuss *cost functions*. Fitness is most intuitively understood when it is given on a bounded and increasing scale. This is not a requirement, but simply an easy way to conceptualize it, with low quality solutions having fitnesses near 0, and high-quality solutions having fitnesses near 1. When directly evolving replicas, we find it easiest to specify a goal, and measure the distance of a

replica from that goal. Replicas can be arbitrarily far from the goal and the highest quality replica is the replica with the minimum distance from the goal. We thus term the distance from the goal *cost* and the fitness functions here, intended to be minimized rather than maximized, *cost functions*. We formulate four cost functions for the direct evolution of replicas.

Formulation of Cost Functions for Direct Replica Evolution

Our first two functions, $f_{outcome}$ and $f_{outcome\_time}$, are naive optimizers of our internal metrics for evaluating the quality of behavior inferred by Turing Learning. We have characterized our dispersion behavior in terms of the mean neighbor distance and mean swarm speed over the course of a simulation, noting that a swarm of ideal agents quickly reaches and remains at an approximate mean neighbor distance of 35 and mean swarm speed of 0. We thus formulate 2 cost functions from these metrics. The first considers these values only at the end of a simulation. The second considers the metrics over the entire course of a simulation. Consider a school consisting of copies of a single replica m. Let $d_t(m)$ equal the mean neighbor distance of this school at time t and $\mathrm{v}_t(m)$ equal the mean swarm speed of this school at time t. We thus formally define these functions as:

$$f_{outcome}(m) = \frac{|d_{t_f}(m) - 35| + \mathrm{v}_{t_f}(m)}{2} \tag{4.4}$$

$$f_{outcome\_time}(m) = \frac{1}{t_f}\sum_{t=1}^{t_f}\frac{|d_t(m) - 35| + \mathrm{v}_t(m)}{2} \tag{4.5}$$

Our next cost function, f_{ideal_model}, is inspired by the ideal behavioral model. We convert the implementation of the ideal controller into a cost function. In this function, we define the cost contributed by a single agent i at time t to be the norm of the velocity fish i would have had at time t if it were running the ideal controller. We average this cost across all fish in the swarm at each time step, then average the final cost over all time steps. Formally, we calculate the cost of replica model m to be:

$$\begin{aligned} \vec{r}_{ij}(t) &= \vec{x}_i(t) - \vec{x}_j(t) \\ f_{ideal\_model}(m) &= \frac{1}{t_f}\sum_{t=1}^{t_f}\frac{1}{25}\sum_{i=1}^{25}\left\| \sum_{j\in\Omega_i} sgn(\|\vec{r}_{ij}\| - \sigma)(\|\vec{r}_{ij}\| - \sigma)^2\frac{\vec{r}_{ij}(t)}{\|\vec{r}_{ij}(t)\|}\right\| \\ \sigma &= 40 \end{aligned} \tag{4.6}$$

We finally validated our controller by replicating the fitness function from (Duarte et al., 2016), treating it as a cost function f_{min_dist}. By evaluating a verified fitness function, we can validate our assumption that the controller is expressive enough to learn the behavior using our chosen replica

architecture (the multi-layer perceptron) and evolutionary optimizer (CMA-ES). This function assesses the distance of a robot from its closest neighbor at every time step in a simulation. The cost associated with a school of copies of replica fish m is the mean difference between this distance and a prespecified distance. We set that prespecified distance to 30: Our ideal model has an *mean* neighbor distance of 35, suggesting the minimum distance is lower. Formally, this function is given by:

$$\begin{aligned} min\_dist_i &= \min_{\vec{r}_{ij}} \|\vec{r}_{ij}\| \text{s.t.} j \in \Omega_i, \vec{r}_{ij} = \vec{x}_i - \vec{x}_j \\ f_{min\_dist}(m) &= \frac{1}{t_f} \sum_{t=1}^{t_f} \frac{1}{25} \sum_{i=0}^{25} |min\_dist_i - 30| \end{aligned} \tag{4.7}$$

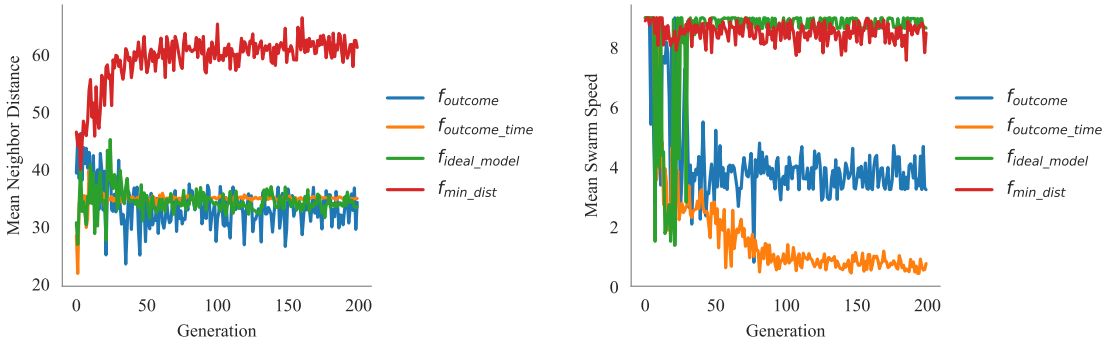
Experimental Set Up: Objective Fitness Approach to Replica Design

We worked with a population of $N = 200$ candidate replica models. We again ran trials for 200 generations. We bounded the possible weights learned during evolution to the range $(-10, 10)$. We analyze the replica with the lowest cost, and thus best objective fitness, at the conclusion of learning. In each epoch, we ran a 15s simulation of a school of 25 fish for each candidate model. The agents were randomly initialized within a circle of diameter 20.

We conducted only 1 evolutionary run for analysis: informal testing suggested very similar outcomes across evolutionary runs. Further, we aim to create a method that can create mimics better than current metrics can classify. As such, we rely heavily on qualitative comparisons between visualizations of ideal and replica swarms. We also examine graphs of the mean swarm speed and mean neighbor distance in simulations. While this comparison method proved more than sufficient to compare the various implementations considered in this work, it meant we did not have good metrics to compare models across evolutionary runs. This did not pose a major problem, however, as the final behavior of replicas evolved using each cost function was sufficiently distinct to assume behavior resulted from the differing cost functions rather than randomness in the evolutionary process.

Comparison of Cost Functions in Evolving High Quality Replicas

We found that the replicas evolved using each cost functions demonstrated notably different behavior. The outcome based metrics, perhaps unsurprisingly, did the best at generating controllers such that at the final time step of a simulation, the mean neighbor speed and mean swarm speed were similar to our desired behavior. A comparison of the mean neighbor distance and mean swarm speed at the final step of the simulation of the highest quality replica in a generation over



(A) (B)

FIGURE 4.2: We graph the mean neighbor distance and mean swarm speed at the final time step of a simulation of the highest quality replica of the generation across generations. We find that the outcome based functions (in blue and yellow) do the best at developing replicas that match our ideal system on these metrics.

all generations can be found in Figure 4.2 and demonstrates this phenomenon. However, f_{min_dist}, the function adapted from (Duarte et al., 2016) led to behavior that was visually most similar to our goal behavior, despite being dissimilar on the mean neighbor and average swarm speed metrics. A simulation of the behavior run by the highest quality model from each experiment can be found in Figure 4.3.

Interestingly we found that the use of $f_{outcome}$, $f_{outcome\_time}$, and f_{ideal_model} led to the inference of behavior where the fish reached approximately the goal neighbor distance and, for $f_{outcome}$ and $f_{outcome\_time}$, goal swarm speed, but did so via notably asymmetrical behavior, with many fish on top of each other. The difference in learned behavior is likely due to the difference in metrics used. $f_{outcome}$, $f_{outcome\_time}$, and f_{ideal_model} all use a mean-based metric for determining the cost of a particular configuration of fish. Thus, neighbors extremely close could be offset by far away neighbors. In comparison f_{min_dist} uses a single spatially dependent metric that will enforce spread between fish.

In addition to qualitative evaluation of the best replica model, we plotted the cost of behavior executed by our ideal model against the cost of the behavior executed by the highest-quality replica over a simulation. We can see the results from these trials in Figure 4.4. These results demonstrate that although the use of objective cost functions generates interesting behavior that is somewhat similar to the ideal behavior, even using these functions, we are unable to evolve behavior as good as the ideal behavior. This may be a result of the chosen architecture of the replica model controller. The limitation may also result from the evolutionary optimization algorithm. However, the learned behavior was sufficiently similar to dispersion that we choose not to modify the replica architecture.

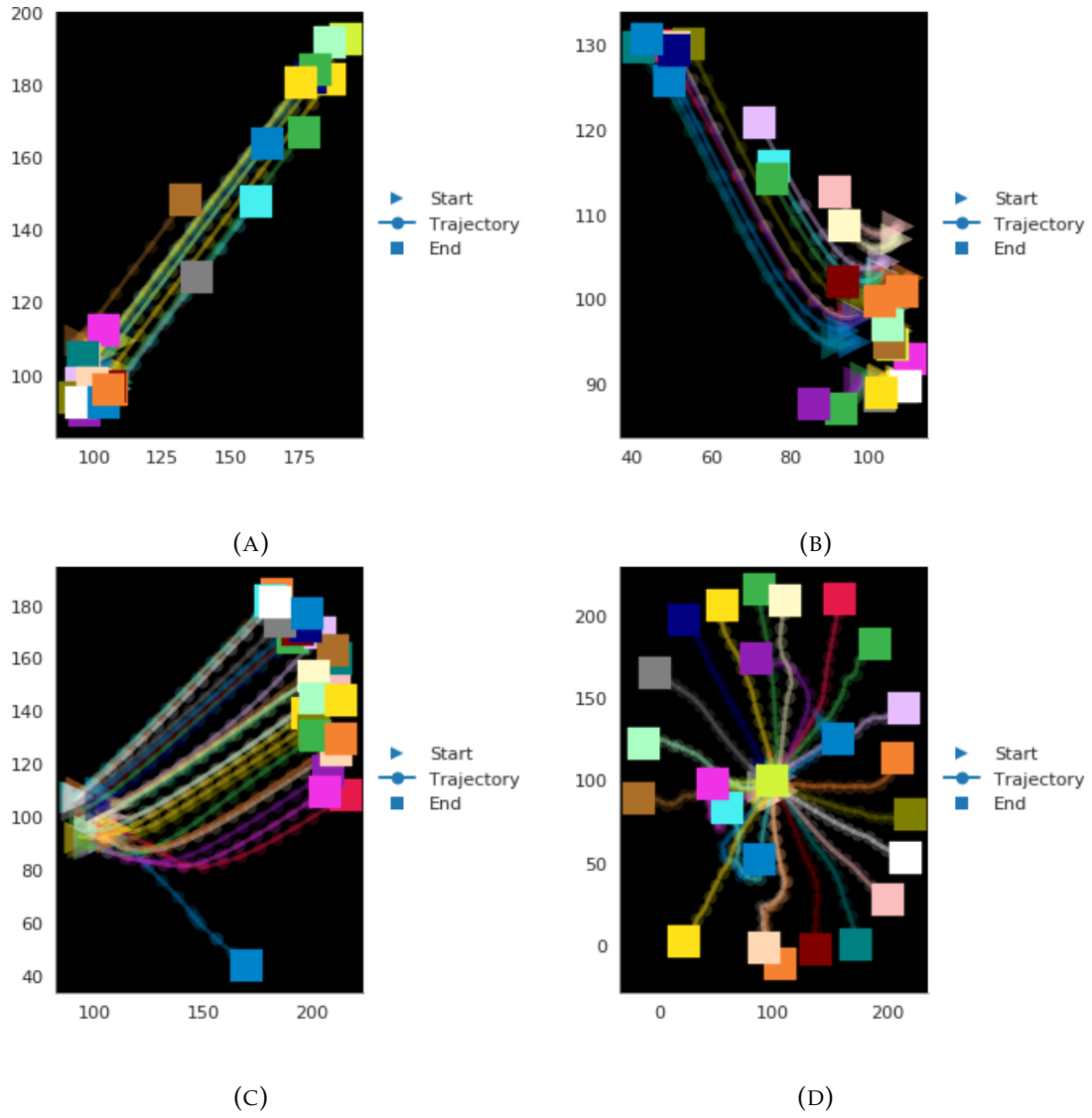
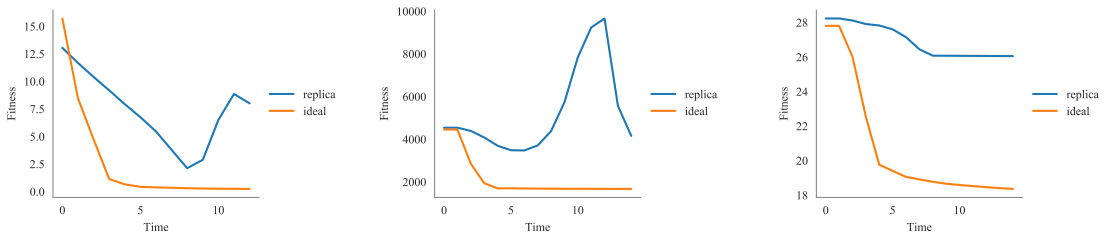


FIGURE 4.3: Simulations of behavior for highest quality model at conclusion of learning. Functions used for determining model quality in each image were: (A) $f_{outcome}$ (B) $f_{outcome\_time}$ (C) f_{ideal_model} (D) f_{min_dist}

4.2.2 Training Replica Controllers with Traditional Machine Learning

To decouple the expressivity of the model from the evolutionary algorithm used, we attempted to fit the neural network controller of the replica exactly to the ideal controller.



(A) (B) (C)

FIGURE 4.4: We plot the cost against time step for (A) $f_{outcome\_time}$, (B) f_{ideal_model} and (C) f_{min_dist}. In each plot the cost of the highest quality replica model after 200 generations is in blue, and the cost of the ideal model in orange. The lower value of the orange line means the ideal model is considered higher quality by all metrics.

Generating a Data Set for Training

The replica was implemented as an agent with the same capabilities as the ideal agent, and with a similar controller. However, an ideal agent i used a hand coded function $\vec{f}_{ij}(t)$ to determine the impact of some neighbor j's position at time t on i's velocity. In contrast, the replica model used an artificial neural network to relate a neighbor's position to its influence on velocity. This neural network was a multi-layer perceptron with 2 inputs, 3 hidden nodes, and 3 outputs. Each node in the hidden and output layer had a bias, and we used the logistic sigmoid function as our activation function and to bound outputs to the range (0, 1). These outputs were then converted to a fish velocity as specified in Equation 3.5.

We desired to train our neural network representing the replica model to fit the function $\vec{f}_{ij}(t)$ executed by the ideal model. To do this, we needed to generate data representing actual values of the function. While this function works across all real numbers, to ensure a practical good fit, we generated data representative of real input values seen by fish in dispersion. We ran 500 dispersion simulations with the ideal fish. We collected the inputs and corresponding value of $\vec{f}_{ij}(t)$ across all fish and all trials, generating a total of 3,891,098 observations. $\vec{f}_{ij}(t)$ considers only $\vec{r}_{ij}(t)$ in its computation, which is the relative position of fish j from the perspective of fish i. The fish have a sensing radius of 100 so in training and executing the replica controller, we divided all inputs $\vec{r}_{ij}$ by 100 so the data lay in the range (-1, 1). To train the replica controller, we needed our training data output values to be equivalent to the outputs of the replica controller. We thus needed to transform the velocity generated by $\vec{f}_{ij}(t)$ to a vector of outputs, essentially the reverse

of equation 3.5. Formally, we converted $\vec{f}_{ij}(t)$ to target values t_1, t_2, and t_3 as follows:

$$\begin{aligned} \vec{v} &= \vec{f}_{ij}(t) \\ (t_1, t_2) &= \frac{\vec{v}}{\|\vec{v}\|} * 0.5 \\ t_3 &= \|\vec{v}\| * \frac{1}{\mathrm{v}_{max}} \end{aligned} \tag{4.8}$$

Experimental Design: Training With Traditional Machine Learning

We used this data set to train the replica controller. We trained the network via stochastic gradient descent (a common machine learning algorithm to optimize neural network weights) on our original network architecture (2 inputs, 3 hidden nodes, 3 output nodes, logistic sigmoid nonlinear activation function, and a bias for all hidden and output nodes). We measured loss during training via Mean Squared Error (MSE). MSE looks at each data sample, and takes the square of the difference between the output predicted by the network using the current network weights, and the actual outputs, then finds the mean of this value over all data. We trained the data for 100 epochs. Each epoch is one pass over the training data, such that all data samples have been able to update the model weights. We also trained an alternative architecture for the replica model containing 50 hidden nodes rather than the original 3. The number of hidden nodes is one factor in the expressivity of a neural network. Increasing the number of nodes generally increases the expressive power of the network.

Results of Training via Traditional Machine Learning

We report the Root Mean Squared Error (RMSE) of the trained model on a subset of the generated data set reserved for testing that was not used in training the model. The test set was 10% of the original data set. The RMSE is simply the root of the MSE, so lower values represent better fitting models. We find that our model as is when trained on generated data had a RMSE of 0.293. The model with 50 hidden nodes actually performed worse, with an RMSE of 0.294. As our target values were in the range (0, 1), a RSME of $\approx$0.3 represents a *significant* error margin. Further, simply improving the number of hidden nodes did not demonstrate an ability to increase the expressivity of the model.

4.2.3 Implications for Turing Learning

Our replica model is unable to perfectly replicate the function executed in the ideal model. However, the behavior learned in the f_{min_dist} trial suggests our replica model architecture is sufficiently

expressive for *dispersion-like* behavior. We do not require that the replica model operates in exactly the same manner as the ideal model: in fact, a promise of Turing Learning is that it may be used to develop similar behavior even when replicas have capabilities different than the ideal agents. Our choice of replica model has created such a situation, as we cannot train the replica to exactly model the ideal controller. However, when training the replicas via traditional evolution, and evaluating behavior of candidate solutions via cost functions judging the quality of the swarm as a whole, we are able to generate controllers that locally act differently than the ideal controllers but cause reasonably similar swarm level behaviors. Although none of the learned behaviors execute dispersion as well as the ideal model according to the defined cost functions, they do demonstrate a set of possible approximations such that learning one of these behaviors would be a successful outcome for Turing Learning.

4.2.4 Formulation of Novel Data Samples to be Used in Turing Learning

Our use of direct evolution of replicas also motivates the formulation of data samples to pass to the classifier such that learning behaviors similar to those above via Turing Learning is possible. Our original data sample consisted of the linear and angular speeds of a single agent over the course of a simulation. This was insufficient information to detect spatial relationships between agents. All the cost functions we used consider in some manner the relationship between agents.

Turing Learning is intended to be a metric-free system inference architecture. We balanced this intention with the realities of learning schooling behavior in our three proposed data samples: **position**, **neighbor awareness**, and **metrics**. The **position** and **metrics** data samples confer *global* information by utilizing data inherent to a swarm rather than a single agent. The **neighbor awareness** sample, like the original implementation, only considers agent-specific information, so is a *local* data gathering metric, but it considers in that local information the context of a single agent.

Position

In the spirit of the metric-free approach, we first consider a fully observational data sample. Rather than passing the classifier the trajectory data of a single agent, we created a sequence of the positions of every member of the swarm. Our intuition is that this will allow the classifier to compute any needed metric of similarity. As we always operate with 25 fish in a swarm, this leads to a classifier with 50 inputs: the (x, y) position of each fish at a given time step. The classifier is run over the time sequence of the simulation before being reset.

A concern here lay in the potential for overfitting. With position information, classifiers may

use location-specific information to distinguish between counterfeit and genuine data samples, but we would like our replicas and classifiers to be generalizable beyond the location in which they are trained. However, we note that replica and ideal agents are randomly placed in an arena using the same algorithm at the start of a trial, so the starting position does not provide distinguishable information. We also note that a generally applicable classifier is not the goal of Turing Learning, but rather a nice side product. The true product is the replica, which cannot detect exact locations. Thus, even if the classifier utilizes specific position information to distinguish data samples, replicas able to fool a classifier will have developed a location-independent function. Though this function may lead to a particular configuration, it is reasonable to assume that configuration is, to the replica controller, generalizable beyond the original arena.

To account for the large input size, we increased the number of hidden nodes in classifiers operating on this data sample to $h = 10$. This leads to a total of 621 weights that need to be optimized via the evolutionary process.

Metrics

We next partially depart from the metric free approach. We note that we used metrics that appear superficially similar in our four cost functions. Further, each seemed like an intuitively reasonable metric for achieving the desired behavior. We found, however, that the learned behaviors were strikingly different. Thus, it may be that the downside of metrics is not inherently in their use, but rather in the difficulty of having to select exactly the correct metric. We thus propose a data sample containing a variety of computed swarm-level metrics. Turing Learning will hypothetically learn which metrics convey useful information regarding a specific behavior and rely on these metrics to distinguish between counterfeit and genuine data. The metrics we selected were drawn from a 2016 review by Bayındır of swarm robotics tasks and their associated performance metrics (Bayındır, 2016). The metrics are fully specified in Table 4.1.

These metrics are relevant to swarm robotics tasks beyond dispersion. As a result, we anticipate that this data sampling method will be generalizable to future applications of Turing Learning within swarm robotics. Accounting for elements of vector metrics, we have a total of 14 elements in our data sample. Thus, this sample leads to $i = 14$ input nodes in the classifier. We again expand the number of hidden nodes to account for the larger input to $h = 10$ hidden nodes for a total of 261 weights to be optimized during the evolutionary process.

Metric Name	Brief Description	Formula	Normalizer	Number
Min Inter-robot Distance	Finds the minimum distance between any 2 robots	$\min_{i,j} \|\vec{x}_t(i) - \vec{x}_t(j)\|$ s.t. $j, i \in M, j \neq i$	50	1
Mean Inter-robot Distance	Finds the mean distance between any 2 robots	$\frac{1}{300} \sum_{i=1}^{25} \sum_{j=i+1}^{25} \|\vec{x}_t(i) - \vec{x}_t(i)\|$	100	2
# Collisions	Collisions determined by interrobot distance < 5	$\|\{(i,j) \mid \|\vec{x}_t(i) - \vec{x}_t(j)\| < 5\}\|$	50	3
Mean Robot COM Dist.	Considers the mean distance of robots from the swarm center of mass (COM)	$COM_t = \frac{1}{25} \sum_{i=1}^{25} \vec{x}_t(i)$, $\sum_{i=1}^{25} \|\vec{x}_t(i) - COM_t\|$	1000	4
Second Moment	Sums the square of all robots' distance from the swarm COM	$\sum_{i=1}^{25} \|\vec{x}_t(i) - COM_t\|^2$	100000	5
Mean Speed	Determine the mean speed of robot movement	$\frac{1}{25} \sum_{i=1}^{25} \|\vec{v}_t(i)\|$	10	6
Bounding Box Area	Determine the size of the smallest possible bounding box around swarm	$\vec{x}_t(i) = (x_t(i), y_t(i))$, $(\max_i x_t(i) - \min_i x_t(i))$ $*(\max_i y_t(i) - \min_i y_t(i))$	10000	7
Mean Dist. Start	Norms the mean displacement of robots from their start position	$\|\frac{1}{25} \sum_{i=1}^{25} \vec{x}_t(i) - \vec{x}_0(i)\|$	75	8
Mean Velocity*	Mean velocity of all robots	$\frac{1}{25} \sum_{i=1}^{25} \vec{v}_t(i)$	1	9, 10
COM Displacement*	Displacement of the swarm center of mass from its start position	$COM_t - COM_0$	60	10, 11
Mean Robot Displacement*	Measure of robots' displacement from their starting position	$\frac{1}{25} \sum_{i=1}^{25} \vec{x}_t(i) - \vec{x}_0(i)$	75	13, 14

TABLE 4.1: Metrics included in the hybrid metrics data sample. Those metrics with a * are metrics that are vectors. Each element of the vector is considered an independent item in the data sample. The normalizer is determined from the maximum possible value of each metric. It is used to convert metrics to lie in the range (-10, 10).

Neighbor Awareness

Our final formulation of a data sample is a local sample inspired by the metrics used specifically for dispersion behavior. This follows from the original approach in that the data sample is collected at an individual agent level. However, failure in the initial approach resulted from a lack of agent contextualization in data samples passed to the classifier. Intuitively, to infer an agent is executing dispersion we need to know something about its position relative to other agents. We thus propose a data sample that includes this context. We term this data sample **neighbor awareness**.

We consider two radii, r_1 and r_2. For each agent i, let $n_1(t)$ be the number of other agents inside a circle of radius r_1 centered at i at time step t, and $n_2(t)$ be the number of other agents inside a circle of radius r_2 centered at i at time step t. For appropriately sized r_1 and r_2, we would anticipate n_1 and n_2 would decrease over the course of a simulation of dispersion. Our data sample consists of a time series of n_1 and n_2.

Our data sample has only 2 inputs ($n_1(t)$ and $n_2(t)$) in each time step, so we have $i = 2$ input nodes. This is the same size input as in the original implementation, so we hold the number of hidden nodes constant at $h = 5$, for a total of 46 weights to be optimized. However, we also need a method of determining r_1 and r_2. We chose to have these values also learned. Work leading up

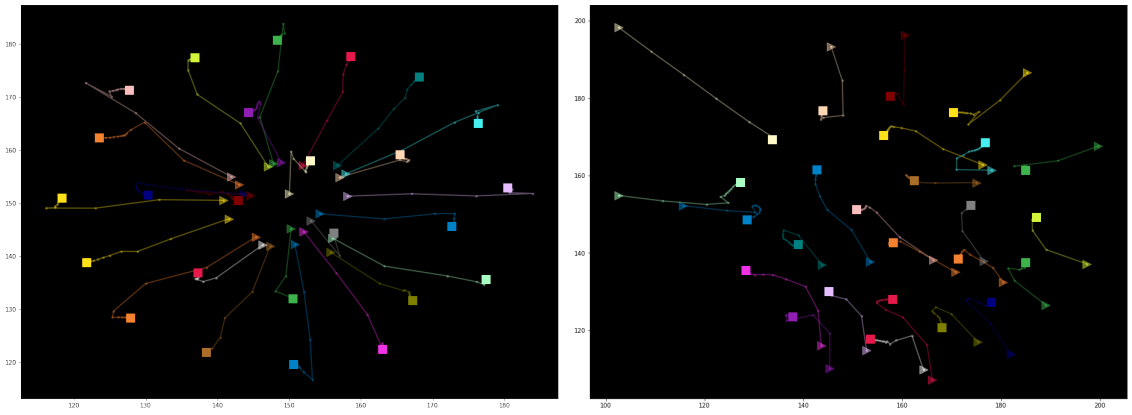
to the development and naming of Turing Learning as a technique examined a co-evolutionary paradigm of model and classifier in the inference of the behavior of a single agent (Li, Gauci, and Groß, 2013). In this work, the classifiers had the ability to "interrogate" the behavior of the agent by outputting both a judgement on a data sample and a stimulus for interaction with the agent. Our decision to have the radii learned follows from this idea that classifiers can "interrogate" the agents they observe. We consider the radii to be a fixed property of a single classifier. Thus a candidate solution for a classifier now includes both the weights of a recurrent neural network, and 2 weights which are appropriately scaled to be r_1 and r_2 then used to generate the data sample fed to this classifier. The addition of these two values leads to a total of 48 weights learned for this data sampling method through the evolutionary process.

4.3 Decomposition of Component Pieces: Classification Analysis

We continue our analysis of Turing Learning by investigating the properties of the classifiers. The classifiers utilize and thus investigate the data samples defined in the preceding section. We also investigate the impact of new formulations of $f_{genuine}$, $f_{counterfeit}$, and g (the function combining $f_{genuine}$ and $f_{counterfeit}$ into a single score) on the classifier's ability to distinguish data samples. To test the classifiers' expressivity, we attempt to evolve classifiers able to distinguish between ideal agents executing aggregation and ideal agents executing dispersion. This section is organized as follows. We first lay out the classification task in 4.3.1. We then analyze failures that occur using a naive fitness function in 4.3.2. This motivates modifying evolution parameters to increase initial diversity of classifiers in 4.3.3. We also propose alternative fitness functions for classifiers in 4.3.4. We conclude with the results of evolving classifiers using the various fitness functions and data samples.

4.3.1 Classifier Task for Direct Analysis

We attempted to evolve classifiers capable of distinguishing between ideal fish executing aggregation and dispersion. In both behaviors, the equilibrium neighbor distance σ is set to 40. However, in dispersion, the fish are initialized within a circle of diameter 20 and in aggregation they are initialized in a circle of diameter 100, leading to a difference in observed behavior. A visualization of each behavior can be found in Figure 4.5. All simulations were run for 15 s using swarms of 25 robots.



(A) Dispersion (B) Aggregation

FIGURE 4.5: (A) Ideal fish models executing dispersion behavior. Swarm begins (marked with triangles) randomly dispersed in a diameter of 20, fish have $\sigma = 40$. Final position is marked with a square. (B) Ideal fish models executing aggregation. Starts with initial spread of 100, $\sigma = 40$. Both behaviors were run for 15 seconds.

4.3.2 Preliminary Evolution of Classifiers on Novel Data Samples

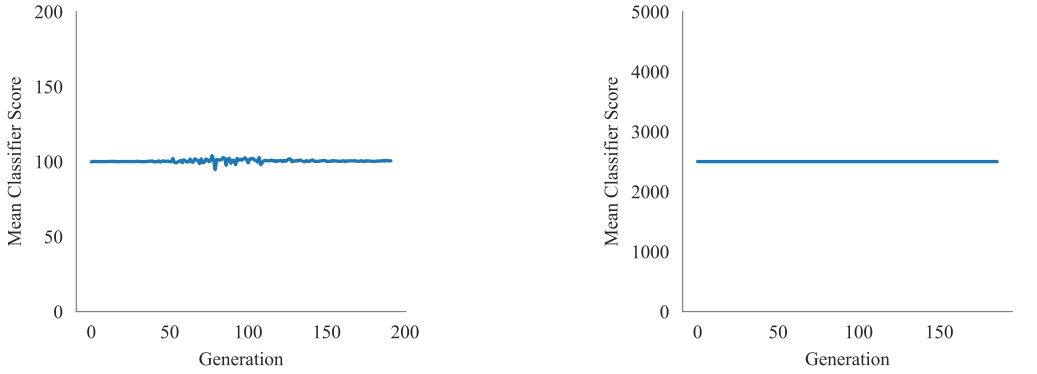
We initially validated classifiers using a slight modification of the fitness function from prior work. We directly evolve classifiers using data samples from 100 swarms executing aggregation and 100 swarms executing dispersion. The fitness of a classifier, r_c was the total number of samples correctly categorized by classifier c. We used the same evolution parameters as in our initial implementation: $\vec{x}_0 = \vec{0}$ and $\sigma_0 = 1$. We again bounded the parameters learned during evolution to the range $(-10, 10)$. We chose to consider initially the metrics and neighbor awareness data sampling methods, as they require the optimization of fewer weights so were better suited for early exploration.

Metrics Trial

Our first trial used 8 of the metrics in Table 4.1. We optimized a population population of 5000 candidate classifiers. The metrics data sampling method creates one data sample per swarm, for a total of 200 data samples.

Neighbor Awareness Trial

Our second trial used the neighbor awareness data sampling method. In this trial, we had a smaller population of candidate classifiers, 500. Classifiers using the neighbor awareness data sample have a smaller number of input and hidden nodes than classifiers using the metrics data



(A) Metrics (B) Neighbor Awareness

FIGURE 4.6: Early trials suggested the classifiers did not evolve well given the evolution parameters and fitness computation. (A) Metrics based classification (B) Neighbor context based classification.

sample. As a result, evolving classifiers for use on this data sample requires optimizing only 48 parameters, much lower than the 261 parameters to be optimized in the metrics trial. We chose the candidate solution population size in light of the number of parameters being optimized. In using the neighbor awareness data sample, we need to convert learned values into radii. Learned values are intended to be in the range $(-3\sigma_0, 3\sigma_0)$, but we needed radii in the range (0, 300). We thus converted the raw learned value to an appropriate radius value by utilizing the bounds on parameters. Given a weight w_1, we formulated r_1 via:

$$r_1 = \frac{(w_1 + 10)}{20} * 300 \tag{4.9}$$

This data sampling method creates one data sample per agent. With 200 total swarms and 25 fish per swarm, this was a total of 5000 data samples.

Results

The resulting mean classifier scores for each trial can be found in Figure 4.6 and demonstrate a dramatic lack of improvement over an evolutionary run. We hypothesized that r_c calculated in this manner did not provide a sufficiently informative gradient for reasonably fast learning. With this fitness function, classifiers can do quite well (half of max fitness) by simply categorizing all data samples as aggregation or all as dispersion. When exploring alternative solutions, classifiers generally receive a *lower score* when they discover a solution that doesn't simply guess in this manner. It thus is unlikely that classifiers will consistently do better than a uniform random guess if they begin with that guess. These results led us to ask *why* our classifiers always began with a

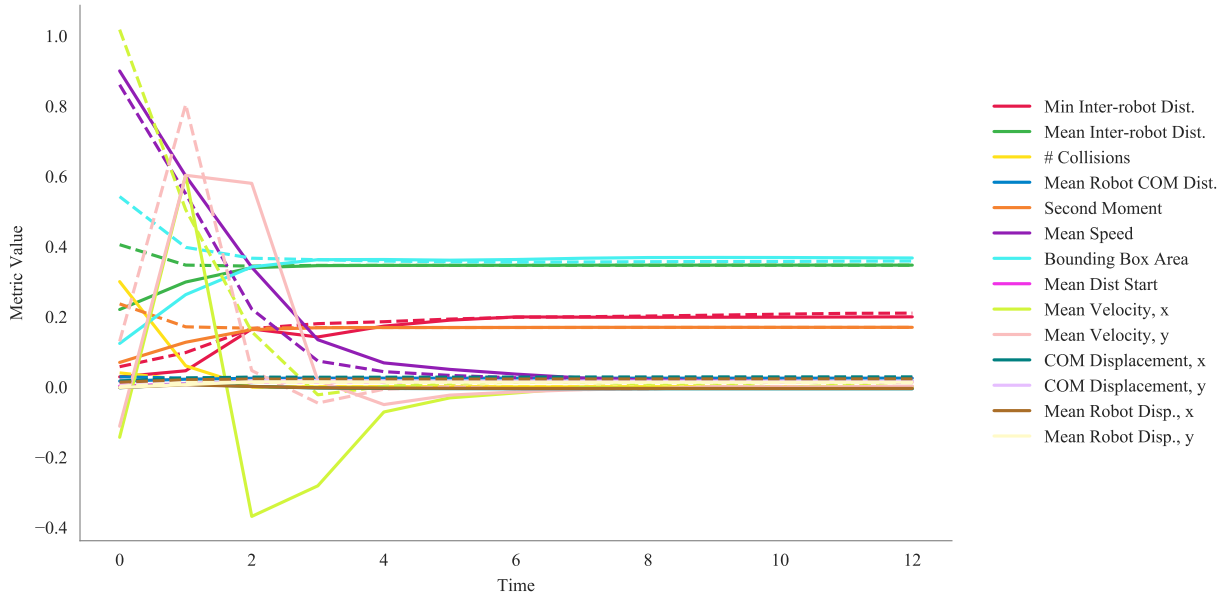


FIGURE 4.7: We compare the values of metrics over a simulation for dispersion (solid) and aggregation (dashed). Mean velocity, bounding box area, second moment, and mean inter-robot distance all begin with notably different values across behavior type, though values converge across behaviors as the simulation continues.

single random guess. In addition, they motivated the formulation of a series of alternative fitness functions for classifiers.

4.3.3 Diversity of Classifiers: Dependence on Evolution Parameters

An important question posed by the results from section 4.3.2 is why all classifiers initially make only a single guess (all data samples classified as aggregation or all as dispersion). We thus set up a series of trials aimed to examine classifiers from the first generation. We again examined only metrics and neighbor awareness data samples.

Metrics

We created a data sample for aggregation and dispersion using the full suite of metrics expressed in Table 4.1. We can see the values, as scaled for classifier input, of all metrics over the course of an aggregation simulation and a dispersion simulation in Figure 4.7. We find that aggregation and dispersion swarms generate clearly different data over a simulation. The metrics differ more at the start of a simulation than at its conclusion, at which point the data from dispersion and

aggregation converge. This comes from the design of the ideal behavior. Each fish is drawn to agents outside the user given radius σ and repelled from agents inside σ. In aggregation trials, fish are distributed inside a circle of diameter much larger than σ, while in dispersion, they are distributed inside a circle of diameter smaller than σ. However, σ is the same across both trials, so the spread of fish at the final time step of a simulation should appear similar across behaviors. The converging metrics reflect this.

Neighbor Awareness

The neighbor awareness data sample is created using aspects of a classifier. We thus are unable to plot a representative data sample. However, the absolute lack of variance in classifier score in Figure 4.6b suggests a problem with the data sample in addition to a problem with the classifier fitness calculation.

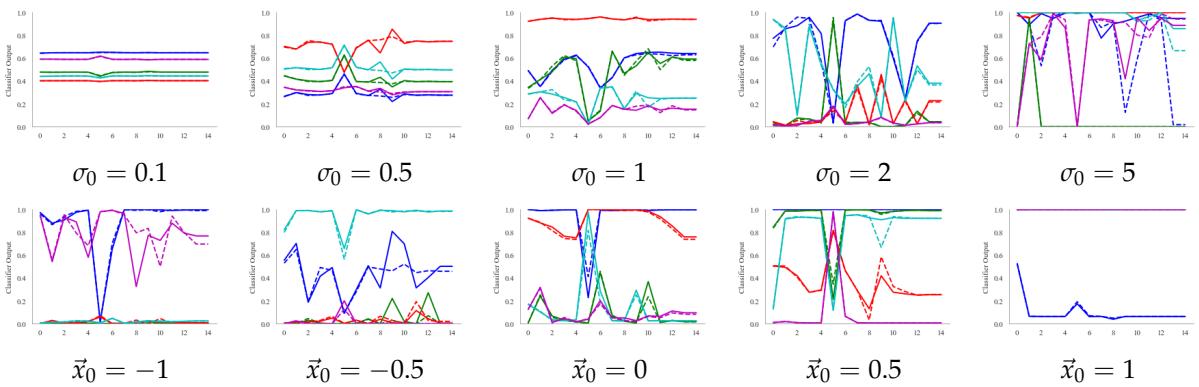
We hypothesize that the function from w to r is a limiting factor. Larger radii will not occur until weights are pushed towards the boundary. However, radii need to be sized sufficiently well that they can distinguish between aggregation and dispersion. Too small radii will never detect neighbors, and too large radii will detect all neighbors, causing data samples for aggregation and dispersion to be the same.

We thus move away from our linear function that depends on bounded values to a quadratic equation that depends on the evolution σ_0. CMA-ES works best when solutions are within $3\sigma_0$ of $\vec{x}_0$. We thus define a function to map values learned via evolution within $3\sigma_0$ of 0 to radii lying in the range $(0, 225)$. The conversion is given as follows:

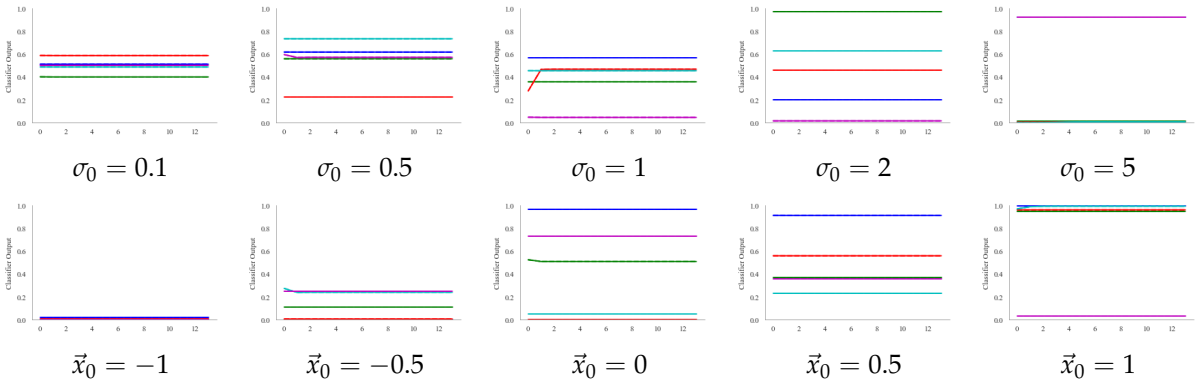
$$r = (\frac{25}{3\sigma_0} w)^2 \tag{4.10}$$

Results

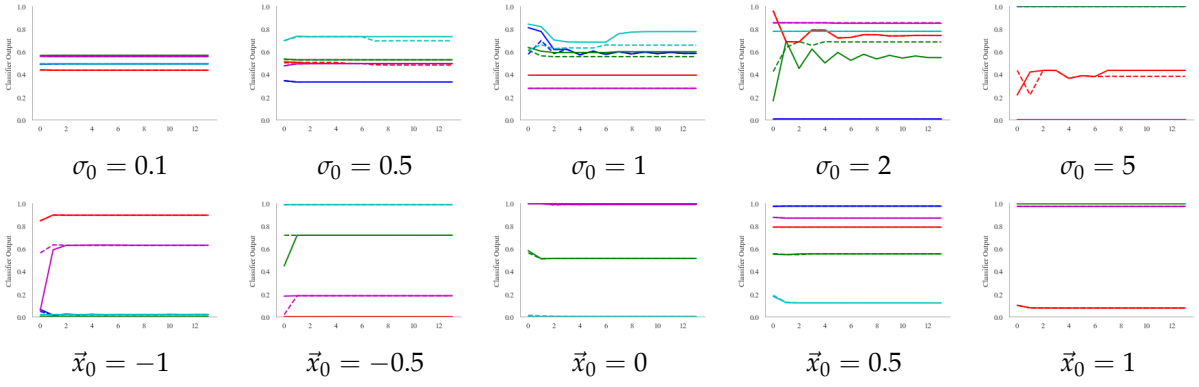
We determine the impact of varying our evolution parameters $\vec{x}_0$ and σ_0 on the classifiers' ability to distinguish between dispersion and aggregation data samples. To do this, we consider the output of a classifier over the course of a trial. We wish to select values of $\vec{x}_0$ and σ_0 such that classifiers generated by an evolutionary process using these parameters demonstrate sensitivity (give distinct classifier output) to different types of data samples. Figure 4.8 contains the outputs of 5 randomly selected classifiers from the first generation of evolution on aggregation and dispersion data samples for a variety of $\vec{x}_0$ and σ_0. We find that $\sigma_0 = 2$ and $\vec{x}_0 = \vec{0}$, and the squaring neighbor awareness scaling method, give the most consistently diverse outputs. We have thus used these parameters in all subsequent experimentation.



(A) Metrics



(B) Neighbor Awareness, scale via bounds



(C) Neighbor Awareness, scale via squaring

FIGURE 4.8: This figure considers how the range of classifiers initialized depends on evolution parameters. We plot the output of 5 randomly selected classifiers evaluating dispersion (solid) and aggregation (dashed, same color for same classifier) data samples over a simulation. Trials varying σ_0 set $\vec{x}_0$ to 0. Trials varying $\vec{x}_0$ use $\sigma_0 = 2$. (B) and (C) also evaluate two methods of scaling radii from learned weights. In this figure, $\vec{x}_0 = 1$ means $\vec{x}_0$ is a uniform vector of 1s.

4.3.4 Alternative Functions to Compute the Fitness of Classifiers

We propose a series of fitness functions designed to provide more informative gradients for optimizing classifier fitness. The first two, $f_{diverse}$ and f_{min}, are premised on the idea that guessing a uniform classification is a poor strategy and should be given low fitness. Our final computation, $f_{outputs}$, considers more information from the classifiers than their final judgement in an effort to extract more precisely the quality of a classifier. We term the original classifier fitness function that averages specificity and sensitivity f_{naive}.

In this section, we examine data samples from swarms executing dispersion and swarms executing aggregation, rather than from replica and ideal swarms. For simplicity, in this section we will equate genuine data samples with aggregation data samples, and counterfeit data samples with dispersion data samples. Thus, the judgement of a classifier on a data sample D, $J(D)$, being 1 represents aggregation and $J(D) = 0$ represents dispersion. We assume we have K data samples from dispersion trials and L data samples from aggregation trials. The lth aggregation data sample is denoted D_l. The kth dispersion data sample is denoted D_k.

Diverse Guesses

A close examination of classifier judgement revealed the existence in early generations of non uniform guessing classifiers. However, these classifiers nearly always received scores lower than classifiers executing uniform guesses, as the increased range of guesses led to more errors overall. We thus decided to add a diversity component to the final score. This component gives a uniform boost to classifiers that correctly identify at least one data sample in each category. Our genuine_c and counterfeit_c scores remained unchanged from initial formulation. They calculate the proportion of genuine and counterfeit data samples the classifier identifies. The complete formulation of $f_{diversity}$ calculate the fitness r_c of classifier c is thus given as follows:

$$\begin{aligned}
\text{genuine}_c = \text{aggregation}_c &= \frac{1}{L}\sum_{l=1}^{L} J_c(D_l) \\
\text{counterfeit}_c = \text{dispersion}_c &= \frac{1}{K}\sum_{k=1}^{K}(1 - J_c(D_k)) \\
\text{diversity}_c &= \begin{cases} 1 & \text{if genuine}_c > 0 \text{ and counterfeit}_c > 0 \\ 0 & o.w. \end{cases} \\
r_c = f_{diversity}(c) &= g(\text{genuine}_c, \text{counterfeit}_c, \text{diversity}_c) \\
&= \frac{1}{3}(\text{genuine}_c + \text{counterfeit}_c + \text{diversity}_c)
\end{aligned} \tag{4.11}$$

Minimum: Do as well as worst classification group

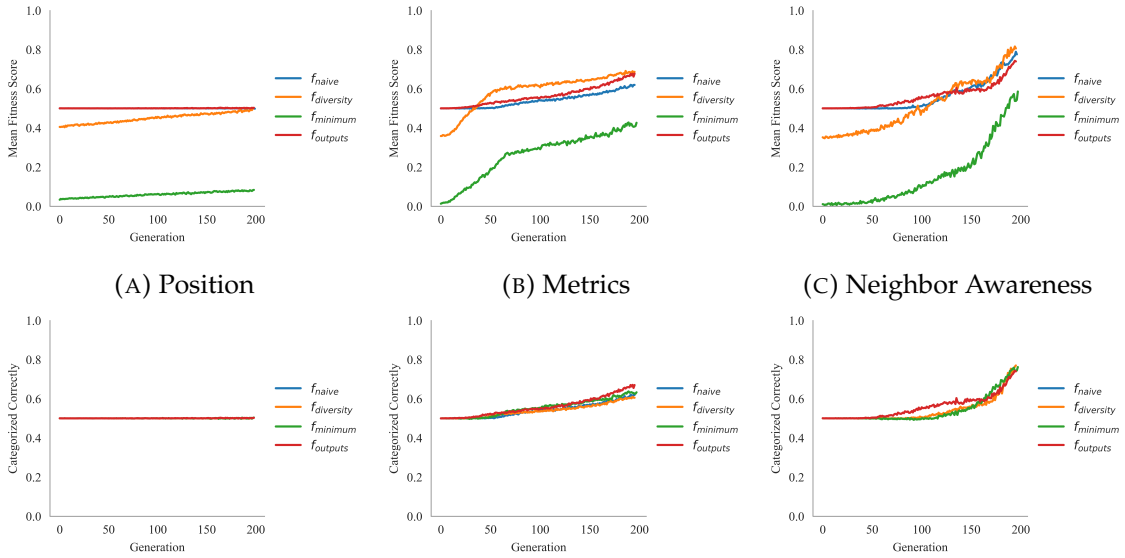
While the above formulation overcomes the challenges we faced with diverse scores being considered lower quality, granting the "diversity" metric a third of the overall score was somewhat arbitrary. We thus also consider a formulation which considers only the group the classifier scores worst on. This formulation uses the same calculations for genuine and counterfeit as above, but combines the score in a different way. We formally define the fitness r_c of classifier c as calculated by $f_{minimum}$ as follows:

$$r_c = f_{minimum}(c) = \min(\text{genuine}_c, \text{counterfeit}_c) \tag{4.12}$$

Outputs: Consider classifier predictions Over time

Thus far, all classification fitness functions we have considered examine only the final judgement of a classifier. However, the classifier produces an output for all time steps of a data sample. Our final formulation uses this sequence of outputs to select for classifiers that offer more distinct output values from the beginning of a time sequence on differing types of data. This additional precision would reward classifiers for getting closer to correctly categorizing a data sample, and so seems to overcome the problem we identified where more diverse guesses led to lower scores. We use this formulation to calculate $f_{genuine}$ and $f_{counterfeit}$ (which equate to $f_{aggregation}$ and $f_{dispersion}$ for this task) and then average these scores. However, it is also possible to combine the full output-based subscores into a single final scores with either of the combination methods described above. The notation $O_c(D_{lt})$ refers to the output of a classifier c after t time steps on the lth aggregation data sample. Using this notation, we formally define $f_{outputs}$ below:

$$\begin{aligned} S &= \sum_{t=1}^{t_f} t^2 \\ \text{genuine}_c &= \frac{1}{SL} \sum_{l=1}^{L} \sum_{t=1}^{t_f} t^2 O_c(D_{lt}) \\ \text{counterfeit}_c &= \frac{1}{SK} \sum_{k=1}^{K} \sum_{t=1}^{t_f} t^2 (1 - O_c(D_{kt})) \\ r_c = f_{outputs}(c) &= \frac{1}{2}(\text{genuine}_c + \text{counterfeit}_c) \end{aligned} \tag{4.13}$$



(A) Position (B) Metrics (C) Neighbor Awareness

(D) Metrics (E) Metrics (F) Neighbor Awareness

FIGURE 4.9: We find that there is no noticeable difference between fitness measures in achieving high correctness in categorization. The neighbor awareness metric was most successful, across all fitness functions, in creating a classifier able to distinguish between dispersion and aggregation. (A), (B), and (C) plot the mean fitness score of the population of classifiers over 200 generations. We also plot in (D), (E), and (F) the corresponding mean over all classifiers of the proportion of data samples a classifier correctly categorizes.

4.3.5 Experimental Setup: Comparing Classifier Fitness Functions

We evolved classifiers in all trials for 200 generations, with data samples from 100 swarms executing aggregation and 100 swarms executing dispersion. The population size of classifier models was dependent on the number of weights being optimized. The number of optimized weights was dependent on the data sample chosen. For position, we had a population size of 5000. For metrics, we had a population size of 2500. Finally, for neighbor awareness, we had a population size of 500. Each population was chosen to be approximately $10n$, with n being the number of optimized values. Given our results in section 4.3.3, we set $\vec{x}_0 = \vec{0}$ and $\sigma_0 = 2$ for all trials. We choose not to bound the parameters learned by evolution as the numerical overflow we encountered in early trials resulted from replica controllers, not the classifiers. We ran one evolutionary run for each of our three data sampling techniques using each fitness metric, for a total of 12 experiments.

4.3.6 Experimental Results

The results of all twelve experiments can be found in Figure 4.9. We found that no combination of fitness metric and data sampling method led to the evolution of a population of classifiers that, on average, categorized more than 80% of data samples correctly by the 200th generation. There was a notable difference between the performance of classifiers across data sampling methods. The position data sampling method demonstrated no performance improvement in terms of proportion of samples correctly categorized. We did note slight increases in mean classifier fitness in trials using $f_{diversity}$ and $f_{minimum}$, suggesting some, though minimal, learning occurred. The metrics data sampling method did better: Depending on the fitness function, we found that by generation 200 the proportion of correct categorization ranged from 0.5 to 0.7. In addition, all fitness scores demonstrate evidence of learning, as the mean score decreased over the generations. Of the three data sampling methods, the neighbor awareness method performed best. It did equally well for all fitness functions. By generation 200, the proportion of correct categorization across all classifiers had reached nearly 80%.

The range in performance across data sampling methods mirrors the size of the classifier. The position data sampling method had 621 weights to learn, whereas the neighbor awareness method had only 48 values to learn. This variability suggests that the number of learned weights is an important factor in the effectiveness of evolving classifiers via evolution, an unsurprising result. In the context of Turing Learning, this suggests that minimizing the size of the classifier is an important factor in developing a successful system. Here, the lack of learning using the position method suggests that this method is unsuited to Turing Learning. All the metric and neighbor awareness trials demonstrate learning, and may be suitable for Turing Learning.

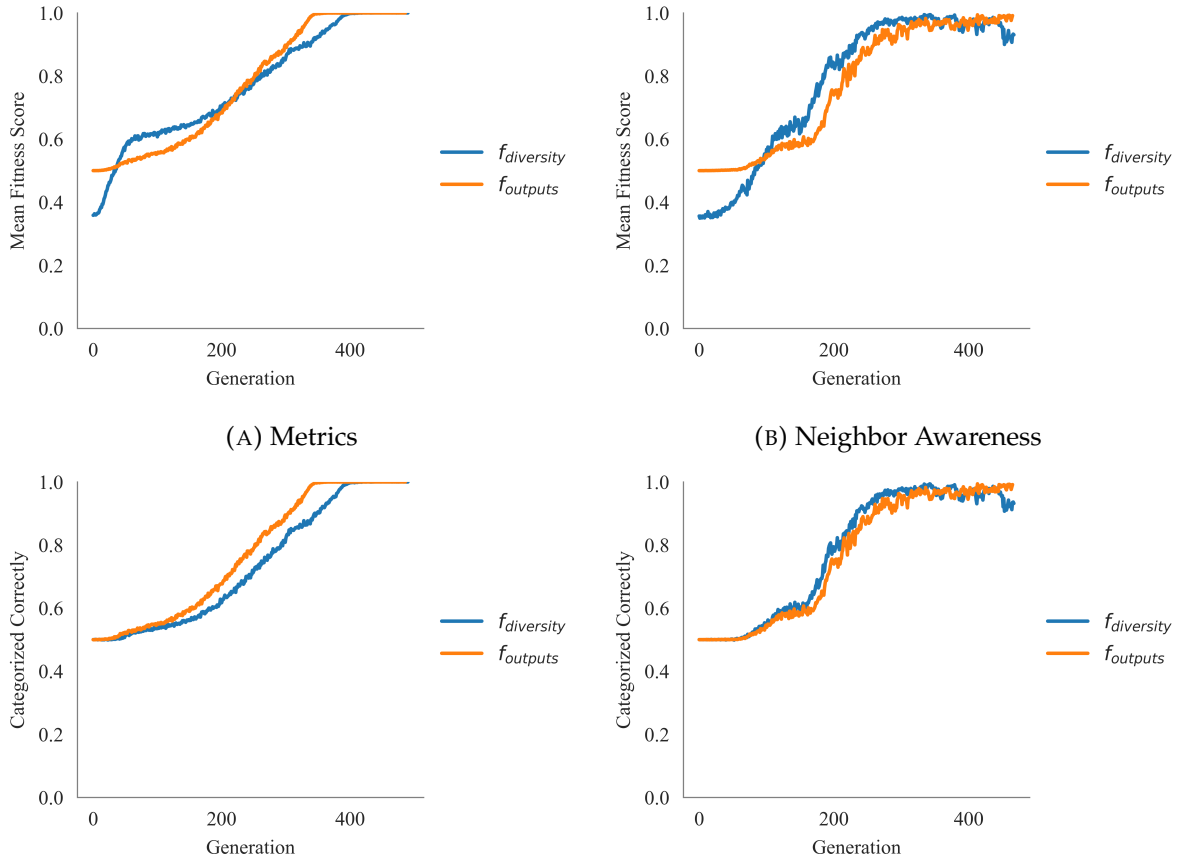
We also compare the learning rates across fitness functions. We first note that the metrics and neighbor awareness trials using f_{naive} demonstrated more improvement over 200 generations than we had seen in preliminary trials. Modifying evolution parameters using results from 4.3.3 had a clear and unexpectedly notable impact on the evolutionary process. We examine Figures 4.9b and 4.9c to compare the impact of alternative fitness scoring methods on the dynamics of evolution. A common problem in evolutionary robotics is termed the *bootstrap* problem. This problem occurs when all initial solutions in an evolutionary process are deemed equally poor, leading to a lack of gradient along which evolution can occur. This is a problem we identified in 4.3.2 and attempted to fix via modifying the starting conditions of evolution in 4.3.3. An alternative solution to the bootstrap problem is a fitness function that is more capable of distinguishing between solutions. We found that in the metrics and neighbor awareness trials, using $f_{minimum}$ or $f_{diversity}$ to calculate fitness decreased apparent bootstrap problems. Both functions led to incremental improvement in the first 100 generations of learning, as evidenced by upward sloping fitness scores

in Figures 4.9b and 4.9c.

The start of improvement in mean fitness score occurs at a later generation in trials using the neighbor awareness data sampling than in trials using metrics data sampling for nearly all fitness functions. This likely results from the need for evolution of both classifier parameters and radii values in the neighbor awareness trials. The evolved radii need to create *distinguishable* data samples before a classifier will be judged high quality, even if the classifier could discriminate between aggregation and dispersion data samples generated using alternative radii. The later steepness of learning seems to indicate that such radii have been found and classifiers are now able to improve based on those data samples.

The relatively poor performance of $f_{outputs}$ in overcoming the bootstrap problem in the first 50 generations was a surprising result. As this fitness function considers all outputs of classifiers over a simulation, rather than only the final judgement, we had hypothesized it would recognize and reward incremental improvement and have the steepest initial learning curve. In addition, data samples from simulations of ideal fish dispersing and aggregating are most distinct in early time steps. The ideal fish behavioral model leads the fish to attract and/or repulse until reaching equilibrium. Fish distance at equilibrium is constant across aggregation and dispersion simulations. Thus, examining only the final data regarding a swarm may lead one to conclude that the two swarms executed the same behavior, even though they arrived in similar formations from different initial positions. Unlike other fitness functions, f_{output} directly considers classifier outputs for each time step of a data sample. We predicted this would lead trials using f_{output} to perform better than alternatives. However, in both the metrics and neighbor awareness trials, notable improvement in quality first appears at a later generations and is slower than in trials using other fitness functions. We suggest the following explanation for this result. Consider the following hypothetical scenario: One classifier examines an aggregation data sample and a dispersion data sample, and outputs values from 0.8 to 0.9 for the aggregation data sample, and values from 0.6 to 0.7 for the dispersion data sample. A second classifier outputs values ranging from 0.5 to 0.6 for aggregation, and 0.4 - 0.5 for dispersion. The first classifier is likely (depending on exact scores) to be considered of higher quality than the second. However, the first classifier had guesses closer to uniformly 1.0 than the second, which actually generates correct dispersion and aggregation *judgements*. This functionality appears to cause initial bootstrapping problems.

Finally, although f_{naive} performed better in these trials than preliminary trials, it still caused slower learning beginning at a later generation than all other fitness functions. This can be seen both directly through the mean classifier scores (Figures 4.9b and 4.9c) and through the mean proportion of correctly categorized data samples (Figures 4.9e and 4.9f). Thus, it is a poor choice for determining classifier quality in Turing Learning.



(A) Metrics

(B) Neighbor Awareness

(C) Metrics

(D) Neighbor Awareness

FIGURE 4.10: Classifiers trained with combinations of $f_{diversity}$ and $f_{outputs}$ fitness functions and metrics and neighbor awareness data samples are able to distinguish between aggregation and dispersion data samples with high accuracy by generation 400. (A) and (B) plot mean classifier score against generation, while (C) and (D) plot the mean proportion of data samples correctly categorized.

We note that learning had not reached a point of stagnation by generation 200. We thus ran trials to 500 generations for $f_{diversity}$ and $f_{outputs}$ using the metrics and neighbor awareness data samples. The results can be found in 4.10. We found that by generation 400, all four trials demonstrated extremely high classification accuracy. The neighbor awareness trial reached a point of high accuracy at an earlier generation, reinforcing our belief that it is a simpler classifier to train. There was no long term difference in the performance of $f_{diversity}$ and $f_{outputs}$, suggesting that though the bootstrap problem may be a significant factor initially for $f_{outputs}$, once overcome this function is capable of evolving high-quality classifiers. The high classification accuracy in the final generation demonstrates that the classifiers are sufficiently expressive for our purpose.

4.4 Preliminary Recombination of Component Pieces

We ran two full Turing Learning trials informed by the investigation of component parts. In these trials, we do not modify the replica or ideal components of Turing learning. We modify the data samples generated by these systems, the fitness function used to determine the fitness of classifiers, and the parameters used to initialize the evolutionary optimizer. A lack of time prevented a rigorous analysis of the impact of varying each component piece on a full Turing Learning trial. Nevertheless, we demonstrate the successful inference of dispersion via Turing Learning using our modifications in preliminary trials.

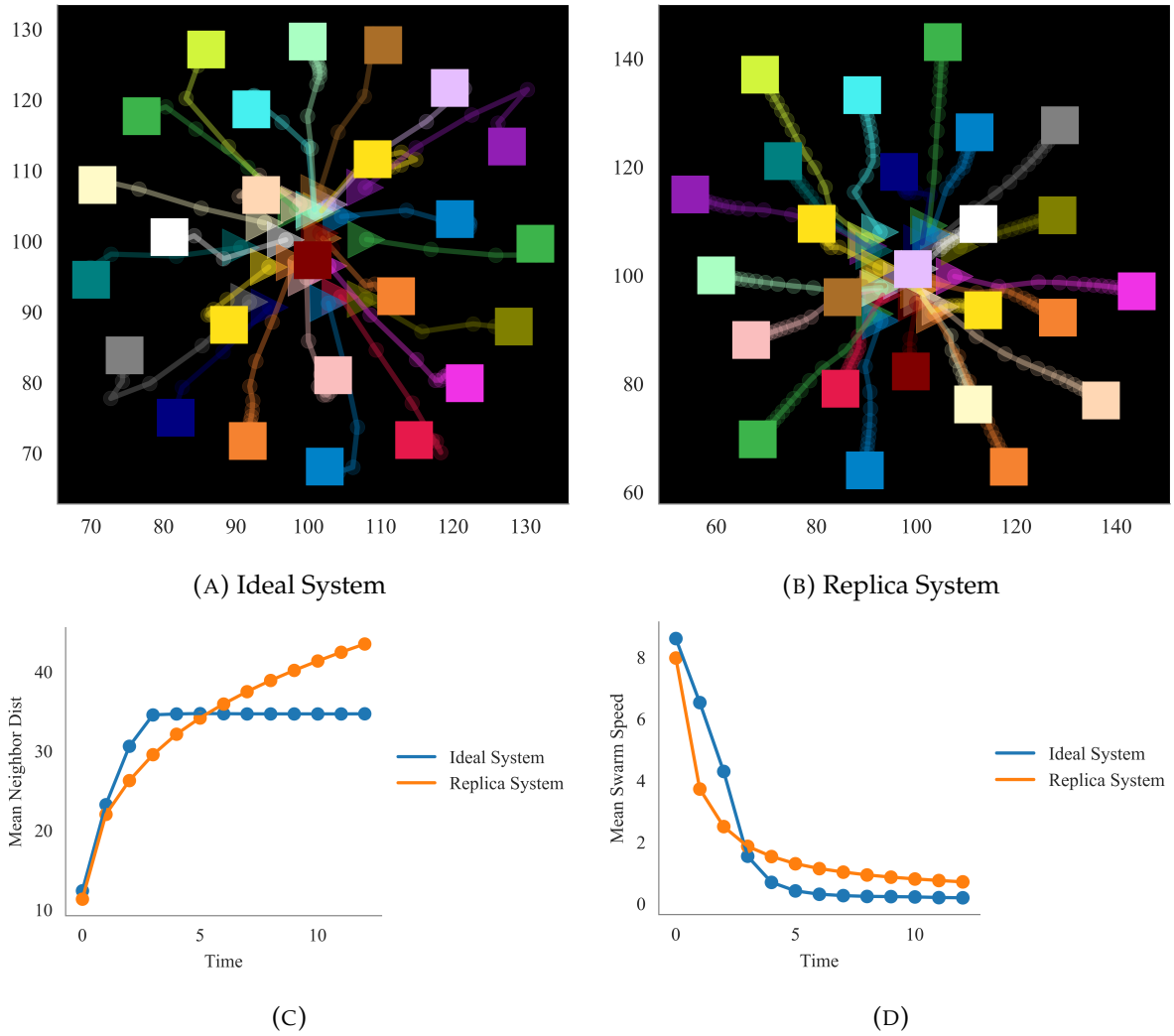
4.4.1 Experiment Set Up: Complete Turing Learning Trial

We ran one trial using the metrics data sample and one using the neighbor awareness data sample. We bounded the weights learned for the replica controller to the range (-10, 10), as we found that even during direct evolution failure to bound these weights led to overflow in the network. When independently trained, classifiers did not generally overflow with unbounded weights, so the weights learned for classifiers were left unbounded. We set $\vec{x}_0 = 0$ and $\sigma_0 = 2$ to initialize the optimizers for both the replica models and the classifiers, based on the results from section 4.3.3. We used $f_{outputs}$ to determine the quality of classifiers. We used the same function to judge the quality of replicas as originally proposed. This fitness function calculates the proportion of a replica's data samples it tricks classifiers into categorizing as genuine. We ran each trial for 800 generations. The metrics trial took 3 days to run.

We found that the neighbor awareness trial crashed when parallelized to the same extent as the metrics trial for reasons we have not yet determined. It repeatedly crashed a few hundred generations into evolution even when fewer processes were run in parallel. We were not able to determine the cause in the time available to us and thus were not able to complete the full trial. Thus, though we attempted to run this trial, we do not have results to present. This error appears to result from the learning infrastructure, rather than from the learning processes itself. We are thus unable to draw conclusions regarding the use of the neighbor awareness data sample in a full Turing Learning trial.

4.4.2 Results

We found that our trial using the metrics data sample successfully inferred dispersion behavior, as can be seen in Figure 4.11. This behavior both qualitatively appears similar to the ideal system, and has similar graphs of mean neighbor distance and mean swarm speed.



(A) Ideal System

(B) Replica System

(C)

(D)

FIGURE 4.11: We show a simulation of the ideal system (A) and a simulation of the highest quality learned replica (B), running dispersion for 15s with a school of 25 fish. We also compare the mean neighbor distance (C) and mean swarm speed (d) of the system. The replica visually appears to mimic the system quite well. Agents in the replica system disperse slightly farther than than in the real system, but show a similar propensity to reach equilibrium.

In addition to comparing the final learned behavior with the ideal behavior, we examine the learning dynamics of the full trial. We examine the mean fitness of classifiers and replicas over generations. As these fitnesses are subjective, reaching a particular fitness value may not be important. However, the change in fitness over generations can demonstrate when changes are occurring. We find that by generation 500, the most significant changes in the classifier fitness cease,

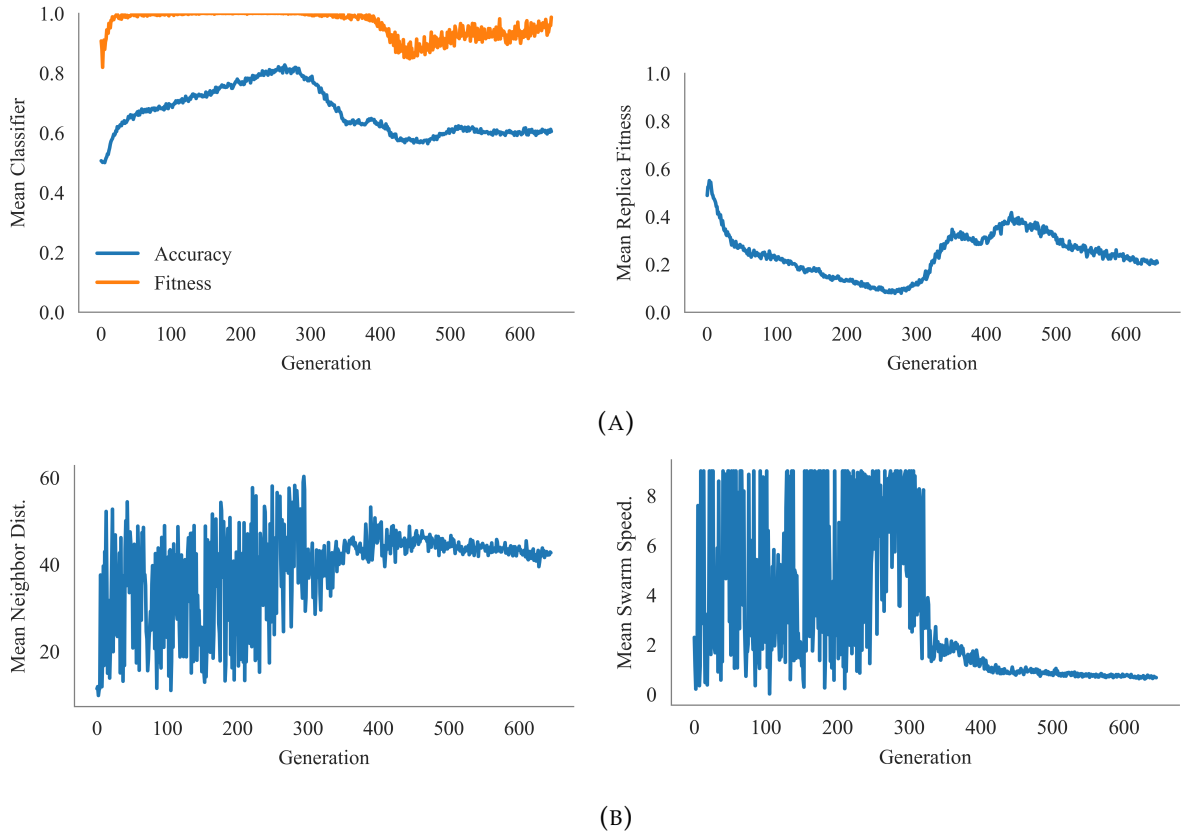


FIGURE 4.12: (A) We graph the learning dynamics of the metrics full trial. We see that by approximately generation 500, learning has stabilized. Though models may be somewhat decreasing in quality, the change is slight. (B) We graph the mean neighbor distance and mean swarm speed at the final time step of a simulation of the highest fitness replica swarm in each generation. Between 400 and 500 generations, all best replicas have similar final mean swarm speeds and mean neighbor distances. Further, a mean neighbor distance of ≈ 40 and a mean swarm speed of ≈ 1 is similar to the final mean neighbor distance of ≈ 35 and swarm speed of ≈ 0 characteristic of the ideal model.

though the model fitness is slightly decreasing. To measure the fitness of replicas via an objective measure, we also plot the mean neighbor distance and mean swarm speed at the final time step of a simulation of a swarm of the generation's highest fitness replica. We find that these measures stabilize by approximately generation 400. In addition, at this point, the values are similar to the corresponding values produced by the ideal system. These results can be seen in Figure 4.12. These results demonstrate that our proposed modifications, namely the use of the metrics data sample and using $f_{outputs}$ to score classifiers, lead to the successful inference of dispersion. Without these modifications, such inference did not occur.

5 Conclusion

5.1 Discussion

We have presented modifications to Turing Learning such that it may infer a replica that mimics dispersion of artificial schools of fish. In doing so, we analyze the strengths and limitations of this system. We found that simply transferring the exact implementation of Turing Learning from (Li, Gauci, and Groß, 2016) was unsuccessful. Turing Learning depends on the co-evolution of replica models able to mimic ideal behavior and classifiers able to distinguish between data samples from replica and ideal models. A crucial piece of this system is the formulation of the data samples generated by the replica and ideal systems. Our initial data sample consisted of the trajectory of a single agent in a swarm. Such a data sample led to the inference of a behavior that locally appeared to mimic ideal behavior well, yet poorly mimicked global behavior. We found that an alternative data sample, metrics, and classifier fitness function, $f_{outputs}$, was required for the successful inference of dispersion.

In analyzing our replica architecture, we found that it was sufficiently expressive to learn behavior similar to our ideal dispersion behavior, but not sufficiently expressive to exactly mimic the ideal behavior. Indeed, when we directly evolved replicas independently of the full Turing Learning system, we found that the ideal model performed better than the best learned replica after 200 generations, at which point the evolutionary process had demonstrated a sustained period of stagnation (lack of further improvement). Further, traditional back-propagation that fit the controller of the replica fish to data generated by the ideal model demonstrated a poor fit to the data. However, Turing Learning can be used to generate replicas that appear to mimic artificial systems in which agents have substantially different capabilities. We thus chose not to modify this architecture, and instead worked to realize this promise of Turing Learning.

Analysis of cost functions to directly evolve replicas out of the context of Turing Learning informed our new formulations for data samples to be generated by ideal and replica swarms. We defined three data samples: **position**, **metrics**, and **neighbor awareness**. All data sample formulations were designed to provide to the classifier information that contextualized a single agent's

behavior, either by simply providing only swarm level behavior (as with metrics) or by providing information of agents in context with other agents (as with position and neighbor awareness). We found that the classifiers proposed are sufficiently expressive to distinguish between schools aggregating and schools dispersing when given the metrics or neighbor awareness data sample. The position data sample was too large for the classifier to effectively make sense of it and utilize it in decision making.

Further, we determined that evolution parameters should be considered carefully to ensure a sufficiently diverse initial population of classifiers and replicas. We also found that the original formulation of determining classifier quality led to a bootstrap problem. Initial candidate classifiers were deemed equally poor and thus the optimization algorithm could not determine which candidate solutions were closer to an ideal solution and optimize accordingly. We defined three classifier fitness functions, $f_{diversity}$, $f_{minimum}$, and $f_{outputs}$, that overcame this problem.

5.1.1 Limitations of Turing Learning applied to Fish Schooling

In applying Turing Learning to fish schooling, we discovered and began to address important limitations in the original implementation of Turing Learning. We first discovered that the choice of data sample generated by the artificial and replica systems is enormously consequential in the quality and nature of inferred behavior. To infer behavior such that a swarm of replicas mimics a swarm of ideal agents, it is likely that the data sample will need to place a single agent's behavior in the context of more global information. The data sample in previous work did not do this, and instead contained only agent level movement information. However, in (Li, Gauci, and Groß, 2016), the swarm behavior inferred was executed by computation-free agents. These agents had limited sensing capabilities and no memory. Their behavior was entirely reactionary, with their velocity entirely determined by one of up to three discrete inputs. Thus, a classifier could observe a trajectory and detect a replica if the replica displayed a velocity different from one of the three displayed by ideal agents. The replica quickly learned to map the inputs to these three possible velocities. That the inferred behavior matched the correct input to the correct velocity can be explained by the fact that the ideal agents observed each possible input, and thus were at each possible velocity, for a different proportion of the trial, a fact noted in (Li, Gauci, and Groß, 2016) to explain why some input-velocity pairings in replicas took more generations until they matched those of the ideal agents. The computation-free nature of the behaviors studied in (Li, Gauci, and Groß, 2016) thus made the desired behaviors significantly easier to infer via Turing Learning than other swarm behaviors. That aggregation and object clustering, the behaviors inferred in (Li, Gauci, and Groß, 2016), could be executed by computation-free agents were notable contributions to swarm robotics literature (Gauci et al., 2014b; Gauci et al., 2014a). There is no reason to believe

that many swarm behaviors will have this quality, nor that this is characteristic in natural multi-agent systems such as schools of fish. Thus, it is important to develop a data sample that includes more than independent agent information.

We found that our metrics and neighbor awareness data samples provided sufficient swarm-level information for classifiers to distinguish between aggregation and dispersion. Fish aggregating and fish dispersing seem to have the same trajectory if observed out of context. The distinction between behaviors lies in the relations between individual agents. Dispersing fish move away from each other, and aggregating fish towards each other, even if they both move in approximately straight lines. We verified that these data samples reveal differences in swarm behavior even when the difference in behavior is characterized by interactions between agents, as in aggregation and dispersion. Although the neighbor awareness data sample was specifically formulated to provide information relevant to dispersion and aggregation, the metrics data sample was created by compiling metrics used to evaluate a variety of quintessential swarm robotics tasks. We thus believe this data sample could be used with Turing Learning to infer a wide range of behaviors.

We further found that determining the quality of classifiers based only on their proportion of correct guesses tends to lead to uniform judgements by classifiers. They categorized all data samples as 1 or all as 0. This causes a bootstrapping problem. All classifiers are initially given the same score and in future generations classifiers that do not guess uniformly often have fewer total correct guesses, leading many generations to pass before any classifiers emerge that do better than a uniform guess. Turing Learning is actually partially designed to overcome bootstrapping problems. In initial generations replicas are extremely poor mimics of ideal agents. Because of this, replicas initially create counterfeit data samples significantly distinct from genuine data samples, making it easier for classifiers to distinguish between samples until replicas improve. That being said, we found alternative classifier fitness functions that led to faster and more rapid improvement of classifiers, and thus advise using one of these alternatives. In a preliminary full test of Turing Learning, we demonstrated that $f_{outputs}$ could be used successfully to evolve dispersion behavior.

Finally, we found that the choice of model for replica agents is more significant than previous work may suggest. Our architecture for the replica was a grey box model informed by the architecture of the ideal model. Each ideal fish i determined its velocity at time step t by summing the independent influence of each neighbor j's position at time t on i's velocity. The replica fish followed this same model. However, the influence of j's position at time t on i was modeled via a neural network instead of an explicit function. The neural network architecture was chosen after the successful inference of behavior with a similar replica controller architecture in (Li, Gauci, and Groß, 2016). However, we found that while this architecture was sufficiently expressive to learn

dispersion-like behavior, directly attempting to train or evolve the ideal behavior failed. Further, simply increasing the number of hidden nodes in the neural network of replicas did not improve their ability to match the ideal controller. Turing Learning claims to be able to train behavior more effectively than direct evolution, and so we chose not to modify the ideal architecture. In doing so, we verified this claim, as we *were* able to evolve successful dispersion behavior where direct evolution approaches failed. Nonetheless, this difference in apparent expressivity of the replica may make it difficult to determine if future failures by Turing Learning result from a failure in the learning process or from a failure to use a sufficiently expressive replica.

5.2 Future Work

This work identified a series of limitations in applying Turing Learning to inferring schooling behavior. We proposed and validated methods that appeared to overcome these limitations in preliminary trials. Future work includes further analysis of the modifications in full Turing Learning trials and validating the solutions on novel swarm behaviors.

Our research initially led us to believe that the proposed replica architecture was insufficiently complex to execute perfect dispersion behavior. While this ultimately proved false, as we did evolve successful dispersion replicas, future work should investigate the use of optimization algorithms that can evolve more complex architectures in addition to optimizing network weights. One such evolutionary algorithm is NEAT: NeuroEvolution of Augmenting Topologies. Embedding NEAT into Turing Learning could be a general solution to the problem of insufficiently expressive architectures. Such a problem will be particularly important when applying Turing Learning to the inference of natural systems.

Although we found that the classifier architecture was expressive enough to distinguish behaviors, evolution of such successful architecture required many generations. To evaluate a candidate solution required simulating a swarm, a non-trivial computation that we anticipate will be common in applications of Turing Learning to swarm robotics. These simulations need to occur each generation, and are extremely time-consuming, making each generation a lengthy and computationally intensive process. An important practical consideration in future work will be to find ways to minimize the number of generations required for evolution of good solutions. The recent work by (Zonta et al., 2018) applying Turing Learning to the generation of human trajectories in crowds found that alternative architectures for classifiers, such as LTSMs, a common deep-learning architecture, performed better in terms of both learning speed and classification quality. Evaluating these alternative architectures in Turing Learning for swarm robotics thus may aid in decreasing the computation time required for a successful evolutionary run. Our successful

evolutionary runs required 3+ days of computation even on servers with 16 CPUs. This significantly limits the practicality of using Turing Learning in novel situations. Discovering failures and proposing modifications is computationally expensive. Decreasing the computation time required is critical to widespread adoption of Turing Learning.

We proposed a series of data samples and classifier fitness functions that led to successful inference of dispersion. The generalizability of our modifications should be tested by applying Turing Learning with these modifications on other swarm robotics tasks such as foraging. Thus far, Turing Learning implementations have needed to be modified for the particular application, and these modifications have been greatly informed by an understanding of the ideal system. An implementation that effectively infers behavior across a variety of tasks is needed to reasonably attempt to apply Turing Learning to the inference of behavior in a natural system.

Bibliography

Bayındır, Levent (2016). "A review of swarm robotics tasks". In: *Neurocomputing* 172, pp. 292–321.

Berlinger, Florian and Fritz Lekschas (2018). "A Distributed and Self-organized Network of Mobile Underwater Robots for Collective Search and Sampling in Dynamic Environments". In: *Harvard School of Engineering and Applied Science: Computer Science 262 - Distributed Systems*.

Berlinger, Florian, Jeff Dusek, Melvin Gauci, and Radhika Nagpal (2018). "Robust maneuverability of a miniature, low-cost underwater robot using multiple fin actuation". In: *IEEE Robotics and Automation Letters* 3.1, pp. 140–147.

Brambilla, Manuele, Eliseo Ferrante, Mauro Birattari, and Marco Dorigo (2013). "Swarm robotics: a review from the swarm engineering perspective". In: *Swarm Intelligence* 7.1, pp. 1–41.

Cazenille, Leo, Yohann Chemtob, Frank Bonnet, Alexey Gribovskiy, Francesco Mondada, Nicolas Bredeche, and José Halloy (2018). "How to blend a robot within a group of zebrafish: achieving social acceptance through real-time calibration of a multi-level behavioural model". In: *Conference on Biomimetic and Biohybrid Systems*. Springer, pp. 73–84.

Couzin, Iain D, Jens Krause, Nigel R Franks, and Simon A Levin (2005). "Effective leadership and decision-making in animal groups on the move". In: *Nature* 433.7025, p. 513.

De Castro, Leandro Nunes (2006). "Evolutionary Computing". In: *Fundamentals of natural computing: basic concepts, algorithms, and applications*. Chapman and Hall/CRC. Chap. 3, pp. 61–122.

Delight, Magnus, Sankaran Ramakrishnan, Thomas Zambrano, and Tyler MacCready (2016). "Developing robotic swarms for ocean surface mapping". In: *2016 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, pp. 5309–5315.

Dorigo, Marco, Vito Trianni, Erol Şahin, Roderich Groß, Thomas H Labella, Gianluca Baldassarre, Stefano Nolfi, Jean-Louis Deneubourg, Francesco Mondada, Dario Floreano, et al. (2004). "Evolving self-organizing behaviors for a swarm-bot". In: *Autonomous Robots* 17.2-3, pp. 223–245.

Duarte, Miguel, Vasco Costa, Jorge Gomes, Tiago Rodrigues, Fernando Silva, Sancho Moura Oliveira, and Anders Lyhne Christensen (2016). "Evolution of collective behaviors for a real swarm of aquatic surface robots". In: *PloS one* 11.3, e0151834.

Elman, Jeffrey L (1990). "Finding structure in time". In: *Cognitive science* 14.2, pp. 179–211.

Gauci, Melvin, Jianing Chen, Wei Li, Tony J Dodd, and Roderich Gross (2014a). "Clustering objects with robots that do not compute". In: *Proceedings of the 2014 International Conference on Autonomous agents and multi-agent systems*. International Foundation for Autonomous Agents and Multiagent Systems, pp. 421–428.

Gauci, Melvin, Jianing Chen, Wei Li, Tony J Dodd, and Roderich Groß (2014b). "Self-organized aggregation without computation". In: *The International Journal of Robotics Research* 33.8, pp. 1145–1161.

Hansen, Nikolaus (2011). *CMA-ES Source Code: Practical Hints*. http://cma.gforge.inria.fr/cmaes_sourcecode_page.html#practical.

Hansen, Nikolaus, Youhei Akimoto, and Petr Baudis (2019). *CMA-ES/pycma on Github*. Zenodo, DOI:10.5281/zenodo.2559634. DOI: 10.5281/zenodo.2559634. URL: https://doi.org/10.5281/zenodo.2559634.

Hansen, Nikolaus, Sibylle D Müller, and Petros Koumoutsakos (2003). "Reducing the time complexity of the derandomized evolution strategy with covariance matrix adaptation (CMA-ES)". In: *Evolutionary computation* 11.1, pp. 1–18.

Katz, Yael, Kolbjørn Tunstrøm, Christos C Ioannou, Cristián Huepe, and Iain D Couzin (2011). "Inferring the structure and dynamics of interactions in schooling fish". In: *Proceedings of the National Academy of Sciences* 108.46, pp. 18720–18725.

Kim, Changsu, Tommaso Ruberto, Paul Phamduy, and Maurizio Porfiri (2018). "Closed-loop control of zebrafish behaviour in three dimensions using a robotic stimulus". In: *Scientific reports* 8.1, p. 657.

Li, Wei, Melvin Gauci, and Roderich Groß (2013). "A coevolutionary approach to learn animal behavior through controlled interaction". In: *Proceedings of the 15th annual conference on Genetic and evolutionary computation*. ACM, pp. 223–230.

— (2016). "Turing learning: a metric-free approach to inferring behavior and its application to swarms". In: *Swarm Intelligence* 10.3, pp. 211–243.

Reynolds, Craig W (1987). "Flocks, herds and schools: A distributed behavioral model". In: *ACM SIGGRAPH computer graphics*. Vol. 21. 4. ACM, pp. 25–34. URL: http://www.cs.toronto.edu/~dt/siggraph97-course/cwr87/.

Turing, A. M. (1950). "Computing Machinery and Intelligence". In: *Mind* LIX.236, pp. 433–460. ISSN: 0026-4423. DOI: 10.1093/mind/LIX.236.433.

Vicsek, Tamás, András Czirók, Eshel Ben-Jacob, Inon Cohen, and Ofer Shochet (1995). "Novel type of phase transition in a system of self-driven particles". In: *Physical review letters* 75.6, p. 1226.

Zonta, Alessandro, SK Smit, Evert Haasdijk, and AE Eiben (2018). "Modelling Human Movements With Turing Learning". In: *2018 IEEE Symposium Series on Computational Intelligence (SSCI)*. IEEE, pp. 2254–2261.