Journal Title
XX(X):1–42

DOI: 10.1177/ToBeAssigned
www.sagepub.com/
SAGE

An Emergent Group Mind Across A Swarm of Robots: Collective Cognition and Distributed Sensing via a Shared Wireless Neural Network

Michael Otte[1,2]

Abstract

We pose the "trained at runtime heterogeneous swarm response problem" in which a swarm of robots must do the following three subtasks: **(1)** Learn to differentiate multiple environmental feature pattern classes, where feature patterns are distributively sensed using the sensors of all robots in the swarm. **(2)** Perform a specific swarm behavior in response to the feature pattern that is recognized in the environment at runtime, where a swarm behavior is defined by a mapping of robot actions to robots. **(3)** Both the specific environmental state pattern classes the swarm learns to differentiate (in subtask 1) and the mapping from feature classes to swarm behaviors used (in subtask 2) are uploaded to the swarm *after* it has been deployed. To solve this problem we propose a new form of emergent distributed neural network that we call the "*artificial group mind*". The group mind transforms a robotic swarm into a single meta-computer that can be programmed at runtime. In particular, the swarm-spanning artificial neural network emerges as each robot maintains a slice of neurons and forms wireless neural connections between its neurons and those on nearby robots. The nearby robots are discovered at runtime. Experiments on real swarms containing up to 316 robots demonstrate the group mind enables collective decision making based on distributed sensor data, and solves the trained at runtime heterogeneous swarm response problem. The group mind is a new tool that can be used to create more complex emergent swarm behaviors. The group mind also enables swarm behaviors to be a function of global patterns observed across the environment—where the patterns are orders of magnitude larger than the robots themselves.

Keywords

Robotic Swarm, group mind, Neural Network, Emergent Behavior, Coordination, Distributed Sensing, Multi-Agent System, Machine Learning, Hive Mind, Artificial group mind.

Introduction

Robot swarms *may* contain *many* robots. In other words, the definition of "swarm" is rooted more in the *scalability* of the algorithms used by a group of robots than in the *size* of the group. A group of robots is a swarm only if it employs methods for sensing, planning, and control that will still work if the number of robots is increased by a few orders of magnitude.

As long as the assumption of scalability is respected, then a particular swarm may contain any number of robots $N > 1$. However, because *by definition* swarm methods are designed to scale to arbitrarily large numbers of robots, they are best suited to problems that have sufficient numbers of robots to break other techniques. Respecting the assumption of scalability means that we cannot provide every robot in a swarm with a unique piece of software. Indeed, to achieve tractability only a small number c of different groups can receive unique programming, where $c \ll N$.

Robotic swarms are useful in situations where having numerous interchangeable robots provides an advantage over using a single robot or a team of uniquely specialized robots. The actuation parallelism provided by swarms enables missions with a high degree of task independence to be completed more quickly. The redundancy provided by large numbers of robots means that a swarm may lose a large percentage of its robots and still accomplish its mission (where the percentage of tolerable loss is problem dependent). Redundancy is particularly useful for dangerous missions and long-term deployments in which we expect some robots to be damaged or destroyed, but we have no way of predicting which robots will be damaged or destroyed.

The traditional approach to producing a particular swarm behavior can be summarized as a two step process: (1) pre-program the same *action* (i.e., a single robot mission program) on each of $c \ll N$ subsets of the swam, and then (2) rely on robot-robot and robot-environment interactions to cause a desired complex swarm behavior to emerge at runtime. This form of complex distributed behavior, resulting from the interaction of multiple agents each performing one of a small set of actions, is called an *emergent behavior*.

The usefulness of emergent behavior is demonstrated in the natural world by the success of biological swarms. For example: bees build hives, ants forage for food, birds flock, fish school; all of these species (and many more) rely on some form of emergent behavior for their survival. The existence of emergent swarm behaviors in the biological

[1]University of Maryland, College Park, MD 20742, USA.
[2]Work performed at: University of Colorado, Boulder, CO 80309, USA
Email: otte@umd.edu

world is a major source of inspiration for robotic swarms. However, many of the methods used to generate emergent behavior in artificial swarms adhere to a design philosophy that limits their diversity of behavior by assuming all robots receive identical programming ($c = 1$). This is much more limiting than many strategies found in the natural world. For example ant colonies have workers, soldiers, males, and queens.

Our work is motivated by the belief that allowing different subsets of robots to be programmed with different behaviors ($1 < c \ll N$) will enable a larger, more diverse, and useful set of swarm behaviors to emerge. Indeed, there likely exist many problems that can only be solved if different robots in the swarm run different programs at the same time, and other problems that can be solved much more efficiently by a swarm that uses a behavior defined by heterogeneous robot actions.

Programming *every single* robot with its own behavior becomes prohibitively expensive as N increases; this is why $c \ll N$. Indeed, if we let $N \to \infty$ then, from a theoretical point of view, we require $c = \mathcal{O}(1)$ to retain tractability over any scale of swarm.

Indeed, even dividing the swarm into c subsets to receive different programming *a priori* can be impractical for a number of reasons. For example, we may not know where a robot will be deployed during a mission, but we may want certain robots to run different action programs depending on where they are deployed. We may also want different robots to run different action programs depending on what else is happening in the environment, or across the environment, or as a combination of deployment location and environmental state.

At a larger scale of organization, we may want the entire swarm to have different swarm-scale emergent behaviors depending on what the swarm senses in (or across) the environment at runtime. Each emergent behavior may require different subgroups of robots to run different action programs, and the swarm may need to be broken into different subgroups depending on which emergent behavior is desired. Assuming the swarm has access to distributed sensor information, then the swarm's behavior may even be selected based on patterns in the environment that are too large for a single robot to discern—but that the swarm as a whole is capable of detecting using its distributed sensors.

Consider the example of a swarm designed to respond to multiple disaster scenarios such as a fires, floods, tornadoes, and earthquakes. The swarm would first need a way to detect which disaster has happened—if any—and then react appropriately. An emergent behavior that is designed to put out a fire will not be helpful if there is actually a flood (and may make things worse). An environment in which "the eastern furnace is burning and the western lake is flooded" will require a different emergent swarm response behavior than "there is a tornado moving south" or even "the eastern furnace is flooded and the western lake is burning".

In practice we may need to deploy a swarm of robots before we know which scenarios the swarm is likely to encounter, e.g., launching a swarm of robots on a mission to another planet. In such a case the swarm can only be trained to differentiate between relevant scenarios after it has been deployed. Similarly, the particular emergent swarm behavior that the swarm should perform in response to any scenario may not be known until robots are deployed; and may depend on how many robots are functional, the locations that the robots end up, etc.

If we are allowed to assume each robot is pre-loaded with a number of single-robot actions (or, alternatively, that such actions can be uploaded to the robots en mass once they are in the environment), then we may wait until the swarm is deployed before telling it (1) which scenarios it needs to differentiate between, and (2) which swarm behaviors (i.e., which subsets of robots perform which actions) are the preferred response to each scenario. In the Preliminaries Section of this paper we formalize the mission described above, and call it the "trained at runtime heterogeneous swarm response problem".

We solve the "trained at runtime heterogeneous swarm response problem" using an emergent swarm-spanning neural network that we call an "artificial group mind". In the artificial group mind each robot maintains a slice of neurons, and the neurons on each robot are linked with the neurons on neighboring robots using ad hoc wireless communication.

The artificial group mind emerges at runtime as neighboring robots discover and then link with each other using wireless communication (See Figure 1). The group mind is trained at runtime to distinguish between different patterns of feature data it detects across the environment using the swarm's distributed sensors. Once trained, the group mind outputs (on every robot) the class label of the current environmental state that is detected. Using this class label, each robot then locally calculates the action that it should perform as part of the desired swarm response behavior. The same class label may map to different actions depending on which subgroup a particular robot belongs. Feature training data, subgroup membership, and class-label mappings are uploaded to the swarm at runtime *after* the swarm is deployed.

The artificial group mind uses wireless signals to train, and wireless signals are unreliable. Therefore, we design a special version of the backpropagation training algorithm that has convergence guarantees despite using unreliable communication between neurons. Part of this paper is devoted to describing this special backpropagation algorithm, and proving its convergence guarantees. The basic idea is to temporarily pause training on a robot whenever information from one of its neighbors gets too far out-of-date; once the neighbor has caught up then the paused robot is unpaused (in theory, any finite number of training iterations may be chosen as the threshold for "too far out-of-date"). Our proofs of convergence are applicable beyond neural network training—they show, given a few nonrestrictive assumptions, this form of pausing will enable any form of distributed stochastic gradient descent to converge in the presence of unreliable communication, almost surely in the limit as the number of iterations increases without bound.

To demonstrate the utility of the group mind we perform experiments with the swarms of Kilobot robots. The Kilobot robotic platform has visual light sensors, communicates using infrared signals, locomotes with vibrating legs, and displays multi-colored LED lights. Once trained, and after a particular projected image is recognized in the environment,

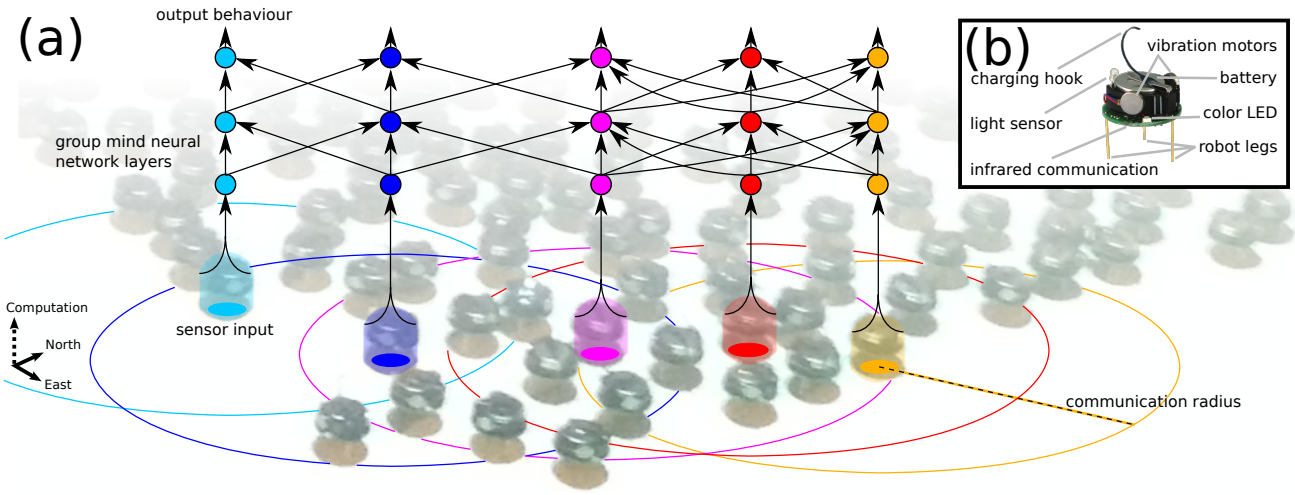


Figure 1. Emergent artificial group mind neural network. (**A**) Each robot maintains a slice of neurons (depicted along the vertical axis) and forms wireless neural connections with its neighbors. (**B**) The Kilobot robot that we use. This Figure is reprinted/adapted by permission from: Springer Proceedings in Advanced Robotics, Vol 1, "2016 International Symposium on Experimental Robotics", COPYRIGHT 2017.

the swarm performs whatever heterogeneous behavior has been prescribed for that particular projected image. We experiment with two different types of heterogeneous swarm behavior output. In the first, the swarm displays special LED images using a single multi-colored LED on each robot such that the entire swarm becomes an emergent LED screen*. In the second, the swarm constructs different physical shapes by having robots move away from certain areas and into others.

The main contributions of our paper are as follows:

1. We both formalize the "trained at runtime heterogeneous swarm response problem" and experimentally demonstrate a solution to it. No previous solution exists that enables a swarm to: (A) be programmed at runtime with different heterogeneous behaviors, such that (B) different behaviors will be performed in response to specific environmental patterns that are (C) detected across the environment using its distributed sensors.
2. The "artificial group mind" technique we present is both new and very general; we believe it can likely solve a number of other problems in which distributed agents need to be trained to make collective decisions.
3. A simple method to achieve distributed stochastic gradient descent in the presence of unreliable communication, along with its proof of almost sure convergence in the limit, is a very general result that can potentially benefit a variety of fields.

The rest of this paper is organized as follows. We begin with a survey of related work, before moving on to a Preliminaries Section in which notation is defined and formal problem statements appear. The High-Level Algorithms Section describes the artificial group mind algorithm and its major subroutines. This is followed by a section in which the medium-level details of the modified backpropagation algorithm are described. Formal analysis of the modified backpropagation algorithm's convergence appears in the Analysis Section. Swarm Behaviors and Actions Used in Our Experiments, the Experiments themselves, Discussion, and Conclusions all appear in their own sections, respectively. Finally, an appendix contains low level implementation details that may be necessary to duplicate our work and additional related work (a survey of related concepts from other fields and a comparison of the group mind to other forms of distributed neural networks).

Related Work from Robotics

In this section we review closely related work in robotics. A high level summary of how previous work relates to our work appears in Table 1. Related ideas from other fields are surveyed in Appendix C, while related work involving neural networks is covered in-depth in Appendix D.

Holland et al. (2005) propose that wireless communication among a swarm of robots can be used to create a distributed computer they call an "UltraSwarm". However, their work is purely speculative with respect to distributed computation, and multi-robot aspects of the problem are not discussed (technical details focus on the design of a robotic platform, and experiments are performed with a single robot). De Nardi et al. (2006) is an extension that details the single-robot motion controller used in the experiments of Holland et al. (2005). More recent work on "UltraSwarm" (De Nardi and Holland 2007) focuses on flocking, and does not mention distributed computation within the swarm.

Distributed computation across a team of six robots is used to find a centralized solution to the multi-robot motion planning problem by Otte and Correll (2010b,a). Robots form teams over ad hoc wireless Ethernet as a response to robot-robot conflicts detected at runtime. The resulting multi robot teams use a distributed algorithm for centralized motion planning. Differences vs. the current paper include the problem that is solved, the size and composition of the team/swarm, the architecture of the resulting distributed computer.

*Note that the light images the swarm detects across the environment are completely different from the LED images that the swarm displays in response.

Giusti et al. (2012) incrementally train a 13 robot swarm to recognize hand gestures that can be used to command the swarm. In extensions to the work a variety of learning methods are applied to the task. Giusti et al. (2012) and Di Caro et al. (2013) use a modified support vector machine (SVM), while Nagi et al. (2012b) use online ridge regression, and Nagi et al. (2012a) combine a SVM with a Neural Network. Each robot has its own camera and all robots observe the same hand. Once trained, each robot locally predicts the hand gesture that it sees and communicates the prediction to the swarm (or a subset of it). Each agent independently runs a consensus algorithm over all predictions to determine the gesture most likely observed. In contrast, the group mind distributes computation and learning across the swarm such that no computation is duplicated on any participant. Another difference is that the swarm response behaviors in our work are allowed to contain heterogeneous robot actions.

Hosseinmardi et al. (2015) embed computational nodes in a smart-wall that sense and share local gesture data to classify the gestures. Our work differs in three ways. First, Hosseinmardi et al. (2015) solves a gesture recognition problem, where gestures take the form of 1-dimensional paths embedded in D-dimensional space. In contrast, we classify D-dimensional feature patterns that have a 2-dimensional geometric component (but no temporal component). Second, Hosseinmardi et al. (2015) require that each computational node (or 7-neighborhood of nodes) maintains a copy of all global data and training examples, and each node (or 7-neighborhood of nodes) computes the classification output in parallel using the k-nearest neighbors algorithm. In contrast, the group mind uses a distributed neural network that does *not* duplicate data used within any strict subsets of swarm. Third, Hosseinmardi et al. (2015) use reliable multi-hop wire communication, while we use unreliable local ad hoc wireless communication.

Distributed mobile sensing is combined with machine learning and data fusion in Chen et al. (2012a, 2013, 2015) to predict both traffic congestion and taxi demand using a form of distributed multidimensional regression. Taxis are equipped with sensors, communication (both centralized and ad hoc are investigated), and *a priori* knowledge of the road network graph structure. Each vehicle maintains its own model of traffic congestion and taxi demand, and summary data from informative parts of the road network are shared between agents in a distributed Gaussian process. The idea of refining predictions by performing coordinated and informed walks is also investigated. In contrast, we solve a variant of multidimensional classification, train the group mind at runtime, and assume robots remains stationary until after a classification has been performed.

Low et al. (2006) implement a neural network on each robot on a swarm of robots. The neural network is used to determine movement based on the locations of targets, obstacles, and other robots. In contrast, the group mind is a single neural network that is distributed across the swarm such that each robot maintains a small set of neurons.

See Table 1 for a summary comparison of the group mind and the closely related works described above.

Previous work related to the parallelization of neural networks is surveyed in-depth in the appendix. In summary, the group mind differs from previous neural networks in at least one of four of ways (depending on the previous work that is being considered): (1) Neurons are spatially distributed in the real world and subject to the network topology that emerges from the placement of participants. (2) Input data is distributed and collected across the swarm by sensors on each robot. (3) The computational nodes communicate using an unreliable (an infrared) ad hoc wireless communication protocol. (4) Each neural strip is embedded on a fully functional robot that can also interact with the world independently of the swarm.

We believe that Noel and Osindero (2014) is the most closely related work from the neural network literature. Noel and Osindero (2014) use unreliable multicast communication to spread messages between the nodes of a distributed neural network, and weight updates are unlocked and asynchronous. Noel and Osindero (2014) differ from our work in that they perform data-parallelism over a wired master-slave system in which all nodes sync to the master, while we use model-parallelism over an ad hoc wireless (infrared light) network that forms between a loose confederation of robots (i.e., computational nodes) of equal priority.

The group mind has 'Any-Com' properties in that performance gracefully declines as the quality of communication between the robots decreases. The definition of Any-Com comes from Otte and Correll (2010b). Previous work on Any-Com algorithms has focused on problems that require search through an initially unknown space to find a set of mutually collision free paths (Otte and Correll 2010b,a) or a space's point-wise collision properties (Otte et al. 2014). In contrast, as we prove in the Analysis Section, the group mind's Any-Com properties are rooted in the fact that stochastic gradient descent (in general) and the training of a neural network (in particular) work even when some messages are dropped.

A preliminary version of the current paper first appeared in Otte (2016a). The current journal version contains significant new material including a theoretical analysis of algorithmic convergence properties, additional experiments, a more refined and detailed presentation of algorithms, and additional discussion of related work. The Kilobot robot system that we use in our experiments has previously appeared in a number of papers, including Rubenstein et al. (2012, 2014); Becker et al. (2013).

Preliminaries

Robots and Set Ordering Assumption

Let n be the index of a robot. The n-th Robot's location is denoted x_n. Let N be the number of robots in the swarm. The set of locations of all robots in the swarm is denoted $\mathbf{X}$,

$$\mathbf{X} = \{x_1, \ldots, x_N\} = \bigcup_{n \in [1,N]} \{x_n\}.$$

For the convenience of using $\mathbf{X}$ as an argument and output of set functions that will be used in our presentation, we assume that $\mathbf{X}$ is ordered by robot ID number when an ordering is necessary or implied. In our presentation we use boldface to indicate sets such than an ordering assumption is made.

	scope of computation	communication	sensing	data storage	deployment	number of robots	how programmed	when programmed	moving senors	prediction
Group mind (this paper)	distributed	wireless	distributed	distributed	yes	4-316	individual + G.M	a priori and runtime	no	state class
Giusti et al. (2012) Nagi et al. (2012b,a) Di Caro et al. (2013)	local/parallel+consensus	wireless	distributed	distributed	yes	13	individual	a priori	no	gesture
Hosseinmardi et al. (2015)	local/parallel+consensus	wired-lossy	distributed	duplicate	simulation	4X4-12X12	individual	a priori	no	gesture
Chen et al. (2015)	distributed and local/parallel+consensus	wireless	distributed	partial-duplicate	real/sim	8/30	individual	a priori	yes	env state
Low et al. (2006)	local	none(visual)	distributed	independent	simulation	60	individual	a priori	yes	output is control
Levine et al. (2016)	centralized (disSampColction)	wired	dis/cent	cist/cent	yes	6-14	individual	a priori	no	grasp control
Hamann and Wörn (2007)	distributed	unspecified	distributed	distributed	simulation	4-1400	emergent	a priori	yes	
Holland et al. (2005) De Nardi et al. (2006)	distributed	wireless	distributed	distributed	real/speculative	1	individual	a priori	yes	
Otte and Correll (2010b)	distributed	wireless	distributed	partial-duplicate	yes	6	individual	a priori	yes	

Table 1. Summary of relationships between our work and closely related work. Top: Group mind. Middle: Works that involve learning in swarms. Bottom: works that involve distributed computation in swarms of robots.

Workspace

We assume the workspace $\mathcal{X}$ of the robot swarm is a subset of euclidean space $\mathcal{X} \subset \mathbb{R}^2$; however, our results generalize to higher dimensional spaces and more complex topologies. Robots exist in the workspace, $x_n \in \mathcal{X}$ for all $x_n \in \mathbf{X}$. Similarly, the set of robot locations exists in a product space $\mathcal{X}^N$ containing N copies of the workspace,

$$\mathbf{X} \in \mathcal{X}^N = \mathcal{X}_1 \times \ldots \times \mathcal{X}_N.$$

Robot Actions

Each robot is assumed to have one or more *actions* that it is capable of performing. Actions can be simple or complex, such as "display LED" or "forage for building materials". A single action may be equivalent to performing multiple simpler actions.

We assume robots have a finite library of actions, and that each action is represented by a unique integer $z \in \mathbb{Z}$. Let α_n denote the action performed by the n-th robot (See Figure 2-top). If the n-th robot performs action z then $\alpha_n = z$.

Swarm Behaviors, Mission Objectives, and Behavior Sets

A swarm *behavior* is defined by each robot performing its own particular action. Let $\mathbf{B}$ denote a swarm behavior. The behavior of a swarm with N robots is described by the set of actions performed by its robots, ordered by robot ID (See Figure 2-bottom).

$$\mathbf{B} = \{\alpha_1, \ldots, \alpha_N\} = \bigcup_{n \in [1,N]} \{\alpha_n\}.$$

A *heterogeneous swarm behavior* is a swarm behavior in which at least two robots perform different actions. Formally, in a heterogeneous swarm behavior there exists at least two robot actions $\alpha_n, \alpha_m \in \mathbf{B}$ such that $\alpha_n \neq \alpha_m$.

Defining a swarm behavior $\mathbf{B}$ as a set of robot actions is convenient for expressing a particular mapping of tasks to robots; however, it is insufficient to express the fact that completely different swarm behaviors may be capable of achieving the same high-level mission objective such as "build space station".

The relationship between a swarm's behavior and its ability to achieve a mission objective is complicated by the facts that: (1) certain swarm behaviors may achieve high-level mission objectives through the use of emergence. In other words, the mission objectives are achieved through the culmination of many robot-robot and robot-environment interactions, and/or on a time-scale that is much longer than any single robot action's control loop. (2) The ability of a behavior to achieve a mission objective may also depend on where the swarm is located.

We assume that different high-level mission objectives can be represented by unique integers $k \in \mathbb{Z}$. Reasoning about a swarm's ability to achieve a high-level mission objective is now formalized. First, we define a swarm behavior tupel $(\mathbf{X}, \mathbf{B})$ that represents a swarm behavior $\mathbf{B}$ performed at a swarm location $\mathbf{X}$. Next, we define a *swarm behavior tupel set* $\mathcal{B}_k$ to be the set of all swarm behavior tupels that achieve the k-th mission objective (See Figure 3).

Let $\mathcal{B}$ be the set containing *all* possible swarm behavior tupels[†],

$$\mathcal{B} = \bigcup \{(\mathbf{X}, \mathbf{B})\}.$$

Any swarm behavior set $\mathcal{B}_k$ is a subset of $\mathcal{B}$.

$$\mathcal{B}_k \subseteq \mathcal{B} \text{ for all } k.$$

Let function $g_k^{mission}$ return true or false if a swarm performing a particular behavior at a particular location will eventually fulfill the k-th mission objective or not, respectively.

$$g_k^{mission} : \mathcal{B} \rightarrow \{true, false\}.$$

$g_k^{mission}$ is a tool that we introduce into our formulation that both: (1) simplifies our presentation, and (2) enables our formalization to generalize to more complex missions. We note that the swarm's location need not remain fixed for the duration of the mission (or even be known to the swarm itself). In the case that $\mathbf{X}$ changes over time as the result of $\mathbf{B}$, the function $g_k^{mission}((\mathbf{X}, \mathbf{B}))$ is assumed, *for the purposes of our formalization*, to account for the future movement and interactions of the swarm, and return true or false based on the current $(\mathbf{X}, \mathbf{B})$. In practice, it may often be impossible to determine $g_k^{mission}$ without actually deploying a swarm to location $\mathbf{X}$, telling it to run behavior $\mathbf{B}$, and then observing if the mission objective is met or

[†]More formally, $\mathcal{B}$ is the smallest possibly infinite set containing *all* possible swarm behavior tupels over all possible swarm sizes. Here "smallest" is used to prevent unnecessary duplication (e.g., in a similar way to how a sigma algebra is defined as the "smallest" collection of all subsets of a set).

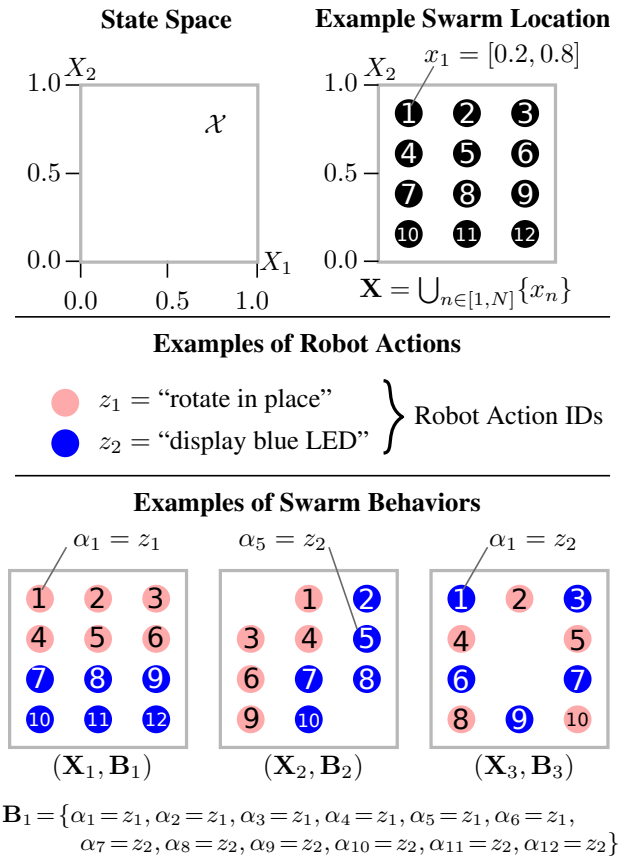


Figure 2. Top: Examples of State Space $\mathcal{X}$, Robot Locations $x_1, \ldots, x_{12}$, The swarm locations $\mathbf{X}_1, \ldots, \mathbf{X}_6$ are sets of robot locations. Middle: Different robot actions z_1 and z_2 are depicted in different colors. Bottom: swarm behaviors $\mathbf{B}_1, \ldots, \mathbf{B}_6$ are each defined by sets of robot actions that are ordered by robot ID. Swarm behavior tupels $(\mathbf{X}_j, \mathbf{B}_j)$ are defined by a set of behaviors performed at a corresponding set of locations.

not. In the special cases considered in this paper, e.g., in Figure 3 and in our experiments, temporal effects are absent from mission requirements such that $g_k^{mission}$ is grounded in a time-independent mapping that is easy to visualize.

Swarm behavior set $\mathcal{B}_k \subset \mathcal{B}$ is formally defined as the (possibly infinite) set of all swarm behavior tupels that fulfill the k-th mission objective:

$$\mathcal{B}_k = \bigcup\{(\mathbf{X}, \mathbf{B}) \,|\, g_k^{mission}(\mathbf{X}, \mathbf{B})\}.$$

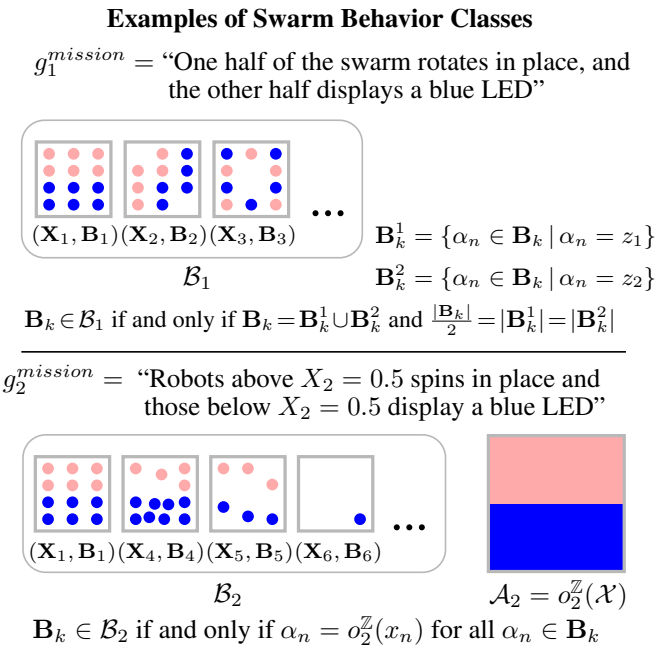


Figure 3. Examples of swarm behavior classes. Classes $\mathcal{B}_1$ and $\mathcal{B}_2$ contain sets of swarm behaviors that fulfill a particular mission objective. In $\mathcal{B}_1$ half of the robots display blue and the other half display red. In $\mathcal{B}_2$ robots display red or blue if they are in the top or bottom halves of the environment, respectively. Mission objective functions $g_1^{mission}$ and $g_2^{mission}$ return true if a particular mission objective is met by a swarm behavior tupel, and so $\mathcal{B}_k = \bigcup\{(\mathbf{X}, \mathbf{B}) \,|\, g_k^{mission}((\mathbf{X}, \mathbf{B}))\}$. A single swarm behavior $\mathbf{B}_k$ is an ordered set of robot actions (ordered by robot ID). In this case, both $\mathcal{B}_1$ and $\mathcal{B}_2$ are infinite sets, and so we show only a few member of each as examples.

Because the k-th mission objective may be met by swarms of various sizes, different swarm behaviors tupels in $\mathcal{B}_k$ will contain $\mathbf{X}$ (and $\mathbf{B}$) with varying cardinality, in general. $\mathbf{X}$ and $\mathbf{B}$ that form a particular tupel have the same cardinality.

The set of all swarm behavior tupel sets is denoted $\mathscr{B}$, where

$$\mathscr{B} = \bigcup_{k\in\mathbb{Z}}\{\mathcal{B}_k\}.$$

Swarm behaviors can be defined in many different ways, and Figure 3 shows two different examples. In the first example a behavior is defined by half the swarm (by cardinality) doing action α_1, and the other half of the swarm doing action α_2. In the second example the behavior is defined such that, at the instant the behavior starts, robots in the top half of the environment perform α_1 and robots in the bottom half of the environment perform α_2. We find the second method convenient, and use it extensively in this work. We now formalize a few concepts that make this possible. Let $o^{\mathbb{Z}}$ be a function that maps robot actions to spatial locations. A swarm behavior template is defined $\mathcal{A} = o^{\mathbb{Z}}(\mathcal{X})$ and can be displayed as an image in which the colors represent different robot actions (See Figure 3-Bottom-Right). Given a $o_k^{\mathbb{Z}}$ for a particular class $\mathcal{B}_k$, a swarm at location $\mathbf{X}$ can calculate the actions $\{\alpha_1, \ldots, \alpha_N\} = \mathbf{B}$ that make up a valid behavior in the behavior tupel set $\mathcal{B}_k$,

$$o_k^{\mathbb{Z}} \implies \mathbf{B} = o_k^{\mathbb{Z}}(\mathbf{X}) \text{ such that } (\mathbf{X}, \mathbf{B}) \in \mathcal{B}_k.$$

Environmental Features

Each robot in the swarm gathers data about the environment using its sensors, and the swarm's group mind uses data from all robots to infer the global state of the environment. We now define notation for the data collected by the swarm about the environment.

An *environmental feature* Φ is a one-dimensional space containing all possible values of a particular property of the environment; for example, the property "temperature". A *feature value* is a single value of a feature; for example "30 degrees Celsius". We denote a value of the i-th feature as observed by the n-th robot as $\phi_{n,i} \in \Phi_i$. It is important to note that each feature value is associated with the robot that observes it—and not, e.g., a particular location in the environment. While a feature value observation is implicitly linked to a particular location in the environment via the position of the robot that observes it; in general, a robot need not know its own location in order for the feature values that it observes to be useful.

The *environmental feature space* $\mathscr{F}$ is defined over one or more environmental features.

$$\mathscr{F} = \Phi_1 \times \Phi_2 \times \ldots$$

We assume that $\mathscr{F}$ contains a single dimension for each mission-relevant feature that can be sensed by at least one robot in the swarm. A *feature vector* defines a point in the feature space. The feature vector of the n-th robot is denoted $\vec{F}_n \in \mathscr{F}$, where

$$\vec{F}_n = [\phi_{n,1}, \phi_{n,2}, \ldots].$$

The *swarm's feature vector set* $\mathbf{F} \in \mathscr{F}$ is the set containing all robots' feature vectors,

$$\mathbf{F} = \{\vec{F}_n, \ldots, \vec{F}_N\} = \bigcup_{n \in [1,N]} \{\vec{F}_n\}.$$

Collectively, the feature vectors in set $\mathbf{F}$ contain the current distributed feature data collected across the entire swarm.

Environmental Feature Patterns

An *environmental feature pattern* $o^{\mathscr{F}}$ is a piecewise continuous function that maps locations in the workspace $\mathcal{X}$ to feature vectors in the feature space $\mathscr{F}$,

$$o^{\mathscr{F}} : \mathcal{X} \rightarrow \mathscr{F}$$

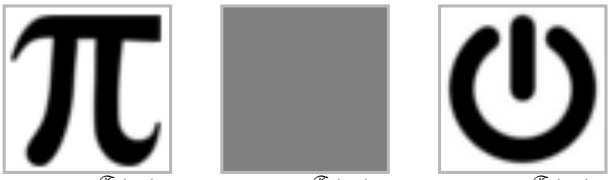
such that

$$o^{\mathscr{F}}(x_n) = \vec{F}_n.$$

We assume that any environmental feature pattern $o^{\mathscr{F}}$ is defined at all points in $\mathcal{X}$ (although it may not be observable by a robot lacking an appropriate sensor). Each feature vector $\vec{F}_n$ is assumed to be a sample of the piecewise continuous environmental feature pattern $o^{\mathscr{F}}$ taken at a particular robot's location. $\vec{F}_n$ contains one feature value per feature[‡].

With a slight abuse of notation we overload function $o^{\mathscr{F}}$ and allow it to operate on both elements of $\mathcal{X}$ and sets of elements of $\mathcal{X}$; thus,

$$\mathbf{F} = o^{\mathscr{F}}(\mathbf{X}).$$



$y_1 = o_1^{\mathscr{F}}(\mathcal{X})$ $y_2 = o_2^{\mathscr{F}}(\mathcal{X})$ $y_3 = o_3^{\mathscr{F}}(\mathcal{X})$

Figure 4. Examples of three environmental states y_1, y_2, and y_3, defined by environmental feature patterns $o_1^{\mathscr{F}}$, $o_2^{\mathscr{F}}$, $o_3^{\mathscr{F}}$, respectively. In this example the feature space contains a single dimension for visual light intensity.

We assume that many different environmental feature patterns $o^{\mathscr{F}}$ may exist, but only one may be active in the environment at a particular time. We use subscripts to differentiate between different environmental feature patterns.

For example, if we define $\mathscr{F} \equiv \Phi_1$, where feature Φ_1 is "visual light intensity", and we also assume that a particular image of a pi symbol is projected onto the workspace $\mathcal{X}$, then environmental feature pattern $o_1^{\mathscr{F}} = o_{\text{pi\_symbol}}^{\mathscr{F}}$ is the function that maps locations in $\mathcal{X}$ to the visual light intensity values that create that particular pi symbol image (See Figure 4).

Environmental State

The feature pattern that is active in the environment determines the current *environmental state*. Let y_i be the i-th *state* of the environment. Formally,

$$y_i = o_i^{\mathscr{F}}(\mathcal{X}).$$

Let $\mathcal{Y}$ be the set of all environmental states,

$$\mathcal{Y} = \bigcup \{y\}.$$

It is convenient to define an *environmental state class* as the set of all environmental states that meet a particular criteria. An environmental class may be simple, "the average temperature is between 20 and 21 degrees Celsius", or complex "visual light creates a 1-by-1 meter pi symbol when viewed from above."

We assume that there are H different mission-relevant environmental state classes; these can be represented by unique integers $h \in [1, H]$ (See Figure 4 for an example). Function g_h^{state} returns true or false if an environmental state is a member of the h-th environmental state class or not, respectively. $g_h^{state} : \mathcal{Y} \rightarrow \{true, false\}$.

Environmental state set $\mathcal{Y}_h \subset \mathcal{Y}$ is defined as the (possibly infinite) set of all environmental states that belong to the h-th environmental state class:

$$\mathcal{Y}_h = \bigcup \{y \mid g_h^{state}(y)\}.$$

We define the set of all mission-relevant environmental state classes as $\mathscr{Y}$, where

$$\mathscr{Y} = \bigcup_{h \in [1,H]} \{\mathcal{Y}_h\}.$$

[‡]In our work we assume that all robots have identical sensors; however, it this were not the case, then feature values from robots that did not have an appropriate sensor should be given a value of "undefined".

This paper focuses on the case in which environmental state categories are defined by environmental feature patterns that may be spatially heterogeneous across the environment (in other words, feature patterns that are not the same everywhere). Spatially homogeneous patterns are also allowed, but are arguably simpler to detect, and so we assume the heterogeneous case in our formalization. For example, we perform experiments in which $\mathcal{Y}_1$ is defined as "a particular pi symbol is projected onto the environment using visual light" and $\mathcal{Y}_2$ is defined as "A particular on-off symbol is projected onto the environment using visual light" (See Figure 5). We assume the features that define a state category can be measured by the robots' onboard sensors. Swarms are particularly well suited to recognize and distinguish between spatially heterogeneous environmental feature patterns because they can contain many robots spread across a relatively large set of space.

The Heterogeneous swarm response problem

In the *heterogeneous swarm response problem* the swarm must perform a behavior from the correct behavior tupel set $\mathcal{B}_k$, where $\mathcal{B}_k$ is determined as a function f of the environmental state class $\mathcal{Y}_h$ that contains the current environmental state y. Let f be the required injective mapping function from environmental state classes to behavior tupel set, $f : \mathscr{Y} \rightarrow \mathscr{B}$. In other words, $f : \mathcal{Y}_h \mapsto \mathcal{B}_k$.

For simplicity we assume that the mission-relevant environmental state classes are mutually exclusive. That is, each environmental state belongs to only a single mission-relevant environmental state class, $\mathcal{Y}_h \cap \mathcal{Y}_{h'} = \emptyset$ for all $h \neq h'$. We now formally define the heterogeneous swarm response problem (a less rigorous high level description can be found in this footnote[§]).

The heterogeneous swarm response problem:
given a swarm of N robots at locations $\mathbf{X}$, a set of mutually exclusive environmental state classes $\mathscr{Y} = \{\mathcal{Y}_1, \ldots, \mathcal{Y}_H\}$, a set of behavior tupel sets $\mathscr{B} = \{\mathcal{B}_1, \ldots, \mathcal{B}_{\max}\}$, and an injective mapping $f : \mathscr{Y} \rightarrow \mathscr{B}$; the swarm must perform a swarm behavior $\mathbf{B}$ such that both of the following requirements are met:

- $\{\alpha_1, \ldots, \alpha_N\} = \mathbf{B}$
- $(\mathbf{X}, \mathbf{B}) \in \mathcal{B}_k = f(\mathcal{Y}_h)$ if and only if the current environment state $o_i^{\mathscr{F}}(\mathcal{X}) = y_i \in \mathcal{Y}_h$.

Estimating environmental state

Solving the heterogeneous swarm response problem requires, among other things, detecting which $o_i^{\mathscr{F}}$ is active in the environment. In principle this can be done by sensing all of $\mathcal{X}$ to get y, and then calculating $o_i^{\mathscr{F}}$ from the relationship $y = o_i^{\mathscr{F}}(\mathcal{X})$. However, although a swarm may be numerous in robots, it may not be able to observe all points in the environment. Therefore, the environmental state y must be inferred given the feature vector set $\mathbf{F}$ collected across all robots' locations $\mathbf{X}$.

Knowledge of $\mathbf{X}$ is implicit in the deployment of the swarm (each robot is at its own location), the swarm can observe $\mathbf{F}$ directly, and knowledge of various mission-relevant $o_y^{\mathscr{F}}$ can—in principle—be provided to the swarm at runtime.

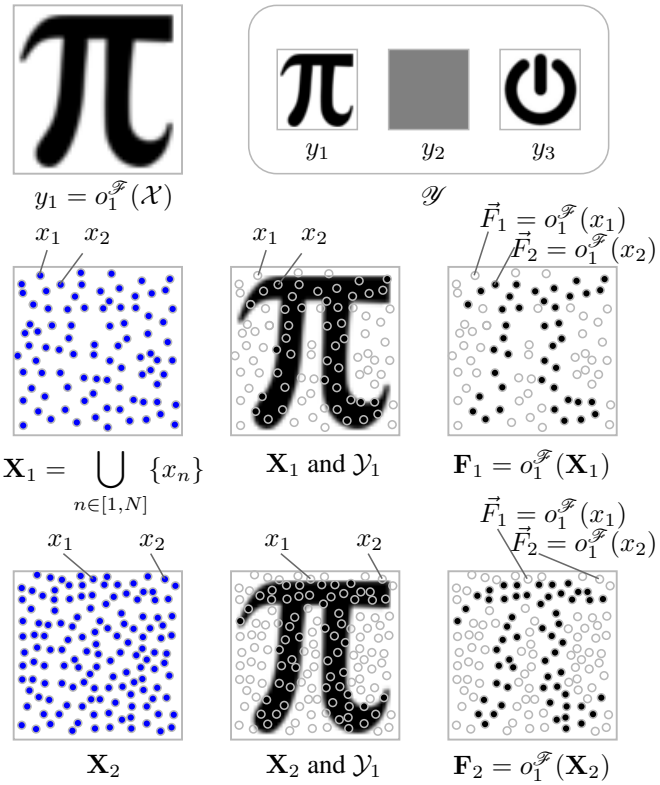


Figure 5. Top: Examples of three environmental states y_1, y_2, and y_3, defined by environmental feature patterns $o_1^{\mathscr{F}}$, $o_2^{\mathscr{F}}$, $o_3^{\mathscr{F}}$, respectively. The set of mission-relevant feature patterns $\mathscr{Y}$. Middle Left: swarm location $\mathbf{X}_1$ with the specific locations x_1 and x_2 of robots 1 and 2 labeled. Middle Center: The location of the swarm, depicted within the environmental state. Middle Right: the set of feature vector samples $\vec{F}_1$ obtained by the swarm at its current location assuming the environmental state is y_1. Selected feature vectors for robots 1 and 2 are also shown. Bottom: the same quantities are depicted as in the middle row, assuming the same environmental state y_1 but a different swarm location $\mathbf{X}_2$ that results in a different feature vectors set $\vec{F}_1$. There are more robots in $\mathbf{X}_2$ and the location of x_1 and x_2 differ between $\mathbf{X}_1$ and $\mathbf{X}_2$. Note that we abuse our notation and overload each function $o_i^{\mathscr{F}}$ such that if it operates on a location then it returns the feature vector at that location (e.g., $\vec{F}_1 = o_i^{\mathscr{F}}(x_1)$); and if it operates on a discrete or continuous set of locations then it returns a discrete or continuous set of feature vectors at those locations, respectively (e.g., $\mathbf{F}_1 = o_i^{\mathscr{F}}(\mathbf{X}_1)$ and $y_i = o_i^{\mathscr{F}}(\mathcal{X})$).

In practice, given $\mathbf{X}$ and $\mathbf{F}$ it is only possible to infer an approximate function $\hat{o}_i^{\mathscr{F}}$ such that $\mathbf{F} = \hat{o}_i^{\mathscr{F}}(\mathbf{X})$ is locally equivalent to $o_i^{\mathscr{F}}$ at $\mathbf{X}$. While $\hat{o}_i^{\mathscr{F}}$ is *only* guaranteed to be valid for $\mathbf{X}$; it provides an approximation $\hat{y}_i = \hat{o}_i^{\mathscr{F}}(\mathcal{X})$ for the overall environmental state, $\hat{y}_i \simeq y_i = o_i^{\mathscr{F}}(\mathcal{X})$.

In summary, $\mathbf{X}$ and $\mathbf{F}$ can be used to obtain the approximation $\hat{o}_i^{\mathscr{F}}$, and $\hat{o}_i^{\mathscr{F}}$ is then used to calculate the most likely $o_i^{\mathscr{F}}$ that is active in the environment. Once we have determined the most likely $o_i^{\mathscr{F}}$, then we can calculate

[§]From a high level point of view, the *heterogeneous swarm response problem* is solved by having a swarm perform a "correct" behavior in response to the current environmental state, where the specification of what behaviors are "correct" for various environmental states is assumed to be provided *a priori*, e.g., by a user. Many different behaviors may be considered "correct," and any "correct" behavior may involve various robots performing different actions (or even the same actions).

$y = o_i^{\mathscr{F}}(\mathcal{X})$ for the current y observed at runtime, and then determine $\mathcal{Y}_h$ such that $y_i \in \mathcal{Y}_h$.

Learning a direct mapping from sensor data to environmental state class index

As discussed in the previous subsection, the goal of having the swarm sense the environment is to calculate $\mathcal{Y}_h$. In practice, we can remove one level of indirection and have the swarm instead learn a direct mapping from various $(\mathbf{X}, \mathbf{F})$ to the relevant environmental state class index h, which uniquely determines $\mathcal{Y}_h$.

This is done by providing the swarm with examples of each environmental state class $\mathcal{Y}_h$, as observed from the swarm's current position, and then having the swarm learn a classification function based on all examples.

Given $\mathbf{X}$, each training example $\mathcal{T}_{h,i} = (h, \mathbf{F}_i)$ is a tupel implicitly defined by $o_i^{\mathscr{F}}$ such that both $o_i^{\mathscr{F}}(\mathbf{X}) = \mathbf{F}_i$ and $o_i^{\mathscr{F}}(\mathcal{X}) = y_i \in \mathcal{Y}_h$.

Let the training example set be denoted $\mathscr{T}$, where $\mathscr{T} = \bigcup_h \bigcup_i \mathcal{T}_{h,i}$.

Given $\mathscr{T}$, the most we can hope is that the swarm learns the approximate classification $\hat{J} : \mathscr{F} \to [1, H]$ such that $\hat{J}(\mathbf{F}_i) = h$ for all $(\mathbf{F}_i, h) = \mathcal{T}_{h,i} \in \mathscr{T}$.

Learning a direct mapping to behavior sets

The final part of solving the heterogeneous swarm response problem requires determining an appropriate swarm behavior $\mathbf{B}$ to perform in response to the particular $\mathcal{Y}_h$ that contains the current environmental feature pattern.

Given $\mathbf{X}$ it is also possible to provide the swarm with (possibly multiple) examples from each behavior class $\{\alpha_{1,k}, \ldots, \alpha_{N,k}\} = \mathbf{B}$ and such that $(\mathbf{X}, \mathbf{B}) \in \mathcal{B}_k$ for all k.

Moreover, it is possible to provide the swarm with an injective mapping $\hat{f} : [1, H] \to \mathscr{B}$ such that $\hat{f}(k) = \mathcal{B}_k$.

Trained at runtime heterogeneous swarm response problem

The variant of the heterogeneous swarm response problem that we address in this paper is called the *trained at runtime heterogeneous swarm response problem*. In this variant both (1) training data regarding the potential environmental feature patterns, as well as (2) the appropriate behaviors that the swarm should perform in response to each feature pattern class are uploaded to the swarm at runtime. The problem variant is now formally defined.

Trained at runtime heterogeneous swarm response problem:
Given a swarm of N robots at locations $\mathbf{X}$, a set of mutually exclusive environmental state classes $\mathscr{Y} = \{\mathcal{Y}_1, \ldots, \mathcal{Y}_H\}$ a set of behavior sets $\mathscr{B} = \{\mathcal{B}_1, \ldots, \mathcal{B}_{\max}\}$ with examples $\mathcal{B}_k = \{(\mathbf{X}, \mathbf{B}_{k,1}), \ldots\}$ for each k, a set of training examples $\mathscr{T}$ and an injective mapping $\hat{f} : [1, H] \to \mathscr{B}$; then the swarm must first learn (during a training period) the classification function $\hat{J} : \mathscr{F} \to [1, H]$ such that $\hat{J}(\mathbf{F}_i) = h$ for all $(\mathbf{F}_i, h) = \mathcal{T}_{h,i} \in \mathscr{T}$, and then (after the training phase) perform a swarm behavior $\mathbf{B}$ such that $\{\alpha_1, \ldots, \alpha_N\} = \mathbf{B}$ and $(\mathbf{X}, \mathbf{B}) \in \mathcal{B}_k = \hat{f}(\hat{J}(\mathbf{F}))$.

Emergent Artificial Group Mind Neural Network

The function $\hat{J}$ is learned by a swarm-spanning artificial neural network that emerges at runtime. We call the resulting computational architecture an "artificial group mind neural network", or *group mind* for short.

Each robot in the swarm is responsible for maintaining a slice of L neurons within the group mind (Figure 1), where L is defined *a priori*. Wireless neural connections are established from neurons at layer ℓ on a robot n to those at layer $\ell + 1$ on each of its neighbors $m \in \mathcal{N}_n$ for $0 \leq \ell < L$, where $\mathcal{N}_n$ is the neighbor set of n, and n is considered a neighbor of itself ($n \in \mathcal{N}_n$), see Figure 6.

A neural signal originating from the neuron at layer ℓ on robot n is denoted s_ℓ^n. Each neural connection has a weight associated with it that is maintained by the receiving robot. Let w_ℓ^{mn} be the weighting term applied by robot n to signal s_ℓ^m.

As with standard artificial neural networks (ANNs), each neuron's output is calculated by performing a weighted sum over incoming signals and then passing the result through a step-like activation function $step$. The neural signal originating from neuron at layer ℓ on robot n is calculated:

$$s_{\ell+1}^n = step\left(\sum_{m \in \mathcal{N}_n} s_\ell^n w_\ell^{mn}\right) \text{ for } \ell > 0$$

and the signal values s_0^n at layer 0 are set by the real-time environmental sensor (light sensor) readings (of the robot responsible for that part of the neural network).

$$s_0^n = \vec{F}_n.$$

In our experiments $step$ is the hyperbolic tangent suggested by LeCun et al. (2012).

When properly trained, output signal from layer L on each robot should be the correct environmental class ID $h = \hat{J}(\mathbf{F}_i)$ for all $(\mathbf{F}_i, h) = \mathcal{T}_{h,i} \in \mathscr{T}$.

Training the group mind is accomplished with a version of the backpropagation training algorithm we have modified to work with unreliable communication. In general, backpropagation involves adjusting all link weights w to improve performance vs. the training example set. More specifically, backpropagation involves iteratively sending update signals in the backward direction along neural connections. Let u_ℓ^n be the update signal sent from the neuron at layer $\ell + 1$ on robot n. Update signal u_ℓ^n is sent to all neurons at layer ℓ that provide input to the neuron at layer $\ell + 1$ on robot n.

Updates from the final layer L contain the signal error for each training example $u_L^n = h - \hat{h}_i$ and those from internal layers $\ell < L$ encode the cumulative error at L ascribed to local error at ℓ, which is calculated differently depending on the particular backpropagation algorithm being used (details are provided in Appendix A).

Given updates from $\mathcal{N}_n$, a node n at layer ℓ can adjust the weights assigns incoming neural signals such that overall ANN performance improves.

For the analysis of convergence, it is convenient to concisely represent the group mind's model weights. Let $\mathbf{W}$ be a vector with one element for each link weight in the group

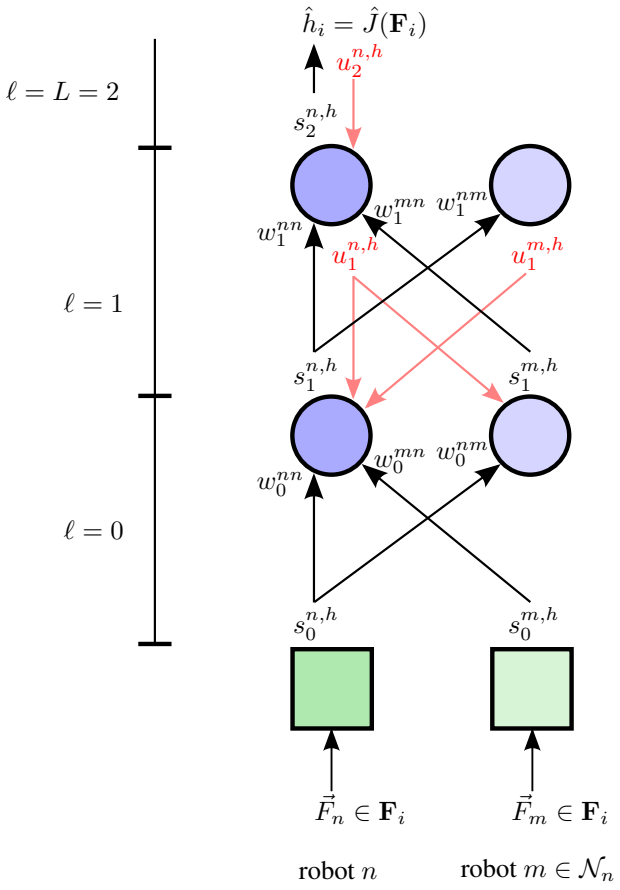


Figure 6. Diagram of variables used in the distributed backpropagation algorithm. Assume code is running on robot n. Robot m is a neighbor of n. Layers $\ell = 0, \ldots, L$ refer to the neural links (black arrows). Layers $\ell = 0, \ldots, L-1$ refer to neurons (purple discs). Each neuron broadcasts a signal s to downstream neighbors, where signals depend on training example/real-time data h. Each node adjusts the weight factor w that it assigns to each incoming link. Update messages u travel in the reverse direction (red arrows), and are used to perform gradient descent (training) in the backpropagation algorithm. Input s_0^n from training examples or real-time sensor readings is assumed to come from a calibrated sensor (green boxes). The group mind neural network outputs the appropriate environmental class index h on each robot.

mind.

$$\mathbf{W} = [w_0^{11}, \ldots, w_\ell^{nm}, \ldots, w_{L-1}^{NN}]$$

The model space of the group mind includes all possible link-weight vectors,

$$\mathcal{W} = \bigcup \{\mathbf{W}\}$$

where $\mathcal{W}$ has one dimension per link weight,

$$\mathcal{W} \subset \mathbb{R}_1 \times \ldots \times \mathbb{R}_{|\mathbf{W}|}$$

and $|\mathbf{W}|$ is the total number of link weights in the neural network. Each vector $w \in \mathcal{W}$ defines a particular decision function $\hat{J}$ that can possibly be encoded by the group mind.

The training error of any neural network can be defined in terms of a cost function $C_{\hat{J}} : \mathcal{W} \to \mathbb{R}$, such that $C_{\hat{J}}(w)$ defines a cost gradient over $\mathcal{W}$.

Training any neural network with a backpropagation algorithm is equivalent to performing gradient descent with respect to $C_{\hat{J}}(w)$.

High Level Algorithms

A diagram of the high-level state machine that runs on each robot appears in Figure 7. The collective distributed operation of the swarm emerges as each robot runs this state machine in parallel and robots communicate.

The overall system contains five states named: Link, Upload, Train, Observe, and Act. Each of these is described shortly in its own subsection. The states Train and Act are significantly more complex then the other states; Train is responsible for training the swarm-spanning neural network based on training data provided by the user, while Act is responsible for running whatever local robot action α_n is part of the greater swarm behavior $\mathbf{B}$ that the group mind decides the swarm should perform. Each of the five phases is now described at a high level. More specific details of states Train and Act are presented later in dedicated (full) sections, while specific low-level implementation details necessary to duplicate our work are available in the appendix.

State Link

Each robot is assumed to start in state Link. In state Link a robot continually broadcasts wireless messages advertising its presence to nearby robots. This allows each robot n to discover the members of its neighbor set $\mathcal{N}_n$.

Each robot is assumed to have an identification number (ID) that is unique within its neighborhood (in other words, no robot in the swarm should have two neighbors with the same ID). This can be achieved by assigning each robot a unique ID *a priori*. However, it can also be ensured with high probability (and more conveniently) by having the swarm to run a distributed unique ID generation algorithm as a pre-processing subroutine during state Link (this is what we do in our experiments).

The structure of the swarm-spanning artificial group mind neural network emerges during state Link. Each robot is pre-programmed to know the depth L of the neural network that will be created over the swarm. For each neighbor $m \in \mathcal{N}_n$, robot n allocates memory to hold incoming and outgoing neural signals as well as training update signals. Incoming data includes neural signals for each neuron at layer $\ell - 1$ on each neighbor m to the neuron at layer ℓ on robot n. Outgoing data includes the output of neurons at layer ℓ on robot n (the same outgoing signal is sent to the neuron at layer $\ell + 1$ that resides on each $m \in \mathcal{N}_n$). Training signals travel in the opposite direction as neural signals and contain slightly different types of data depending on the training algorithm being used. Training signals are described in greater detail in the next section, which is titled "Slice-wise Parallel Backpropagation Neural Network Training Algorithm". In general, a separate set of training signals is required for each of the environmental state classes $\mathcal{Y}_h$ that we train the group mind to recognize.

State Link ends when training data from the user is detected, although many other suitable stopping criteria can also be used (timeouts, special messages, etc.).

State Upload

In state Upload each robot in the swarm uploads data from the user. Uploading this data is what enables the group mind to be programmed at runtime, and it includes:

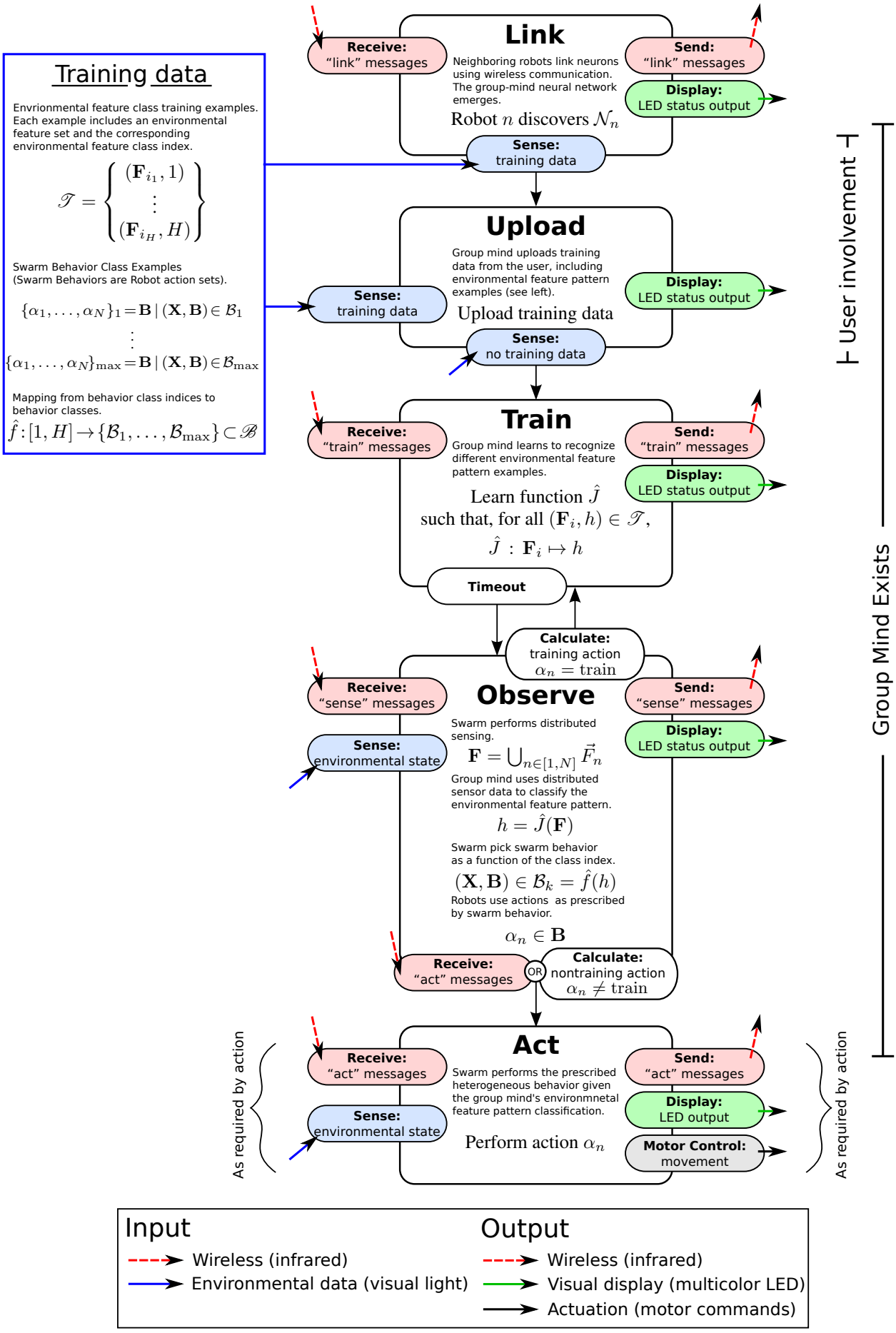


Figure 7. High-level diagram of a swarm hosted group mind solving the "trained at runtime heterogeneous swarm response problem".

- $\mathscr{T} = \{(\mathbf{F}_{i_1}, 1), \ldots, (\mathbf{F}_{i_H}, H)\}$ The training example set that includes at least one example pair $(\mathbf{F}_{i_h}, h)$ for each environmental state class we wish the swarm to recognize. This is the data used to train the neural network to learn function $\hat{J}$.
- $\{\alpha_1, \ldots, \alpha_N\}_j = \mathbf{B}_j \,|\, (\mathbf{X}, \mathbf{B}_j) \in \mathcal{B}_k$ Examples of robot actions that belong to behaviors in all relevant behavior classes. In particular robot n is told $\alpha_n \in \mathbf{B} \,|\, (\mathbf{X}, \mathbf{B}) \in \mathcal{B}_k$ the action that it should perform as part of the swarm behavior $\mathbf{B}$. We assume each robot (swarm) is provided with at least one action (swarm behavior) for each swarm behavior class that the swarm may be expected to perform during its deployment.
- $\hat{f} : [1, H] \rightarrow \{\mathcal{B}_1, \ldots, \mathcal{B}_{\max}\}$ The mapping from class indices to behavior classes. The mapping $\hat{f}$ may be provided *a priori* without sacrificing the ability to program the group mind at runtime.

We assume that this data is uploaded via the robots' onboard sensors. In our experiments, the user is able to change the state of the environment directly so that they can provide the swarm direct examples of environmental states. (An alternative method, which we do not use, is to send such data over the swarm's ad hoc wireless network—this would require the user to address messages to different robots based on their locations in the workspace.)

State Train

State Train, when run collectively across all robots in the swarm, is responsible for training the artificial group mind neural network.

Because wireless neural signals are used, it is probable that messages between neurons will be dropped. As we prove in the Analysis Section, convergence is still guaranteed as long as no robot gets too many training iterations ahead of its neighbors. This is achieved by having robots pause their own training whenever they are more than a user-chosen number of training iterations ahead of their neighbors (this threshold is set to $c_{wait} = 100$ in our experiments). To detect when training needs to be paused, all signals sent along neural connections are tagged with the number of training iterations a neuron's host robot has completed. The function out_of_sync() returns **true** whenever this robot has gotten so far ahead of a neighbor that it must pause its own training.

Algorithm 1: Train

```
1 if not already_initialized then
2     init_neural_network()
3 while time_left() do
4     if not out_of_sync() then
5         backpropagation_training_iteration()
6     send_messages()          // on a separate thread
7     receive_messages()       // via a callback function
```

High-level pseudocode for state Train appears in Algorithm 1. The subroutine init_neural_network() is responsible for initializing the neural network. backpropagation_training_iteration() is responsible for running a single training iteration on the local robot. Messages are continually sent using and received using the callback receive_messages(). We recommend running send_messages() on a separate thread to minimize latency. Medium and lower-level details related to the neural network itself, including pseudocode for the aforementioned subroutines, are described in the "Slice-wise Parallel Backpropagation Neural Network Training Algorithm" Section and in the appendix, respectively.

We now describe the pseudocode for state Train, which appears in Algorithm 1. If the slice of the group mind neural network that resides in this robot has not yet been initialized, then it is initialized (Lines 1-2 in Algorithm 1). While training time remains, the backpropagation training algorithm is run one iteration at a time (line 3-5) — but only if this robot is not out of sync with its neighbors (line 4-5). Message-passing is never paused because neighboring robots need data from this robot to continue their training (lines 6-7).

Although state Train will transition to state Observe upon timeout, it will likely re-enter state Train numerous times as the result of a prescribed action (e.g., if a "keep training" action is set as the default action in state Act, which is described shortly). Alternative exit criteria could also be used; for example, the swarm could detect that it has converged sufficiently vs. the training set, a global signal could be used, etc.

State Observe

State Observe, when run in parallel by all robots in the swarm, passes the current environmental state $\mathbf{F}$ into the group mind neural network to obtain the environmental state classification h. Locally on robot n, this amounts to the robot feeding $\vec{F}_n$ into the neural network input located at level 0 on robot n, then performing forward passes on this data (communicating over an ad hoc wireless network with neurons on neighboring robots) until the group mind's collective computation of $\hat{J}(\mathbf{F})$ works its way through the group mind and $h = \hat{J}(\mathbf{F})$ pops out of the neurons at layer L.

There are a few details of this collective computation that are particularly important. First, it takes multiple rounds of message-passing for any pass through the neural network to complete (and message drops will slow the progress of this calculation). The send and receive functions are identical between state Learn and state Observe, and they cycle through all training messages as well as the signals of the forward pass calculation given the current environmental data. Thus, assuming the robot has spent adequate time in some combination of states Learn and Observe, the output from $h = \hat{J}(\mathbf{F})$ will reflect a slightly delayed environmental state. To be useful, this implementation implicitly assumes that the state of the environment changes at a much slower rate than swarm message-passing and computation.

Another important detail is that, at least in the particular high-level architecture presented in Figure 7, the active state continually switches between Learn and Observe. Thus, $\hat{J}$ is being trained at the same time it is being used. In general this may cause incorrect actions to be output early in the training phase unless appropriate precautions are taken. For example a default "keep training" action should both

(1) be the appropriate action to perform in response to an environmental state that the user is reasonably sure will exist for the duration of the training phase, and (2) be associated with the easiest behavior class index to learn (this depends on the particular training algorithm being used. The latter is possible, for example, by priming the neural network with random numbers such that all environmental states initially output the training state until training enables pattern differentiation. Alternatively, other mechanisms can be used to ensure that a robot stays in state Learn until training is complete.

All robots operate asynchronously and in parallel. Some robots may be observing while others are learning. Messages sent between robots are handled in a separate thread from the main control loop. Thus, data in a message only affects a receiving robot once that robot re-enters the state for which the message was relevant.

Algorithm 2: Observe

```
1  h ← forward_pass(F⃗, 0)        // h ← Ĵ(F⃗)
2  B_k ← f̂(h)
3  (X, B) ∈ B_k                  // pick swarm behavior tupel in B
4  α_n ∈ B                       // α_n is this robot's action in B
5  if α_n = train then
6      run state Train
7  else
8      run state Act with α_n
9  send_messages()               // on a separate thread
10 receive_messages()            // via a callback function
```

Pseudocode for state Observe appears in Algorithm 2. Subroutine $\text{forward\_pass}(\vec{F}, 0)$ returns the result $h = \hat{J}(\mathbf{F})$ as calculated by the neural network. In this pseudocode, we assume the convention that setting the second argument to 0 indicates that the current environmental data is being fed into the neural network and not training example data (which must also be passed around in order to train the group mind).

On Line 1 of Algorithm 2 the group mind neural network is used to calculate h from the $\vec{F}_n$ on robot n as well as all other feature vectors (on other robots) that collectively form $\mathbf{F}$. Given h the appropriate behavior class is calculated on Line 2, using the user-provided mapping $\hat{f}$. Next, a swarm behavior tupel is chosen such that $(\mathbf{X}, \mathbf{B}) \in \mathcal{B}_k = \hat{f}(h)$, line 3; and this robot's action α_n within behavior $\mathbf{B}$ is found (e.g., in a look-up table), line 4. If α_n is "keep training" then the robot goes back to state Train, otherwise it transition to state Act to perform α_n, lines 5-8. As previously noted, messaging (lines 9-10) is identical between states Observe and Train to facilitate training in parallel to observation.

If there exist multiple $(\mathbf{X}, \mathbf{B}) \in \mathcal{B}_k$ then the swarm must decide which one to choose using a distributed consensus algorithm. If each $\mathcal{B}_k$ contains a single tupel for $\mathbf{X}$, then running a consensus algorithm is unnecessary. We take the single tupel per $\mathcal{B}_k$ approach in our experiments. The more general case, in which a variety of different behaviors are allowed in response to a particular environmental state, is presented because we believe that others may find it useful.

State Act

State Act is the final state of the high-level state machine. Once a robot n enters state Act it begins to perform its (non-training) action $\alpha_n = z$ that is part of the swarm's behavioral response to the current environmental state data.

Actions themselves may be arbitrarily complex. For example an action may be encoded as a separate state machine (and that state matching may even allow transitions to the previously described states such as Train, Consider, etc.). In the section titled "Swarm Behaviors and Actions Used in Our Experiments" we discuss the particular actions used in our experiments.

If some robot actions involve movement then the group mind must orchestrate a graceful dissolution before the movement starts. This is due to the fact that movement will break neural connections as robots move away from from their neighbors. Even after the group mind dissolves, the swarm continues to perform the heterogeneous behavior that the group mind calculated in response to the environmental state. This is illustrated in some of the actions used in our experiments, and which are described later.

On the other hand, if no actions contain movement, then actions may continue to make use of the trained neural network. For example, if the environmental state class is dynamic such that $\mathbf{F}$ is expected to change while the swarm is deployed, then the swarm can continually recalculate $h = \hat{J}(\mathbf{F})$ and update its heterogeneous response behavior as necessary. This case is also illustrated in our experiments, and is described in more detail later.

Slice-wise Parallel Backpropagation Neural Network Training Algorithm

The calculation of $h = \hat{J}(\vec{F})$ is what enables the group mind to recognize which environmental feature pattern is active in the environment. Function $\hat{J}$ is encoded in a swarm-spanning neural network that is trained at runtime using examples provided by the user. The training of $\hat{J}$, combined with the user-provided mapping $\hat{f}$ (from feature class indices to behaviors) is how a user programs the group mind to have heterogeneous behaviors in response to different environmental states.

In this section we discuss the middle-level details of two neural-network backpropagation training algorithms the group mind can use to learn $\hat{J}$. In general, all backpropagation algorithms incrementally update internal signal wights so that the neural network's performance improves vs. a training data set. Such training operations are technically a form of gradient descent over the space of network weights, the convergence of which is discussed in detail in the Analysis Section. **Readers that are more interested in the high-level swarm aspects of our work than the distributed training of $\hat{J}$ and its convergence guarantees are encouraged to skip both this section and the Analysis Section on an initial read**.

We experiment with both "batch" and "stochastic" variants of the backpropagation training algorithm. Each is now outlined at a high level, and then described in more detail in its own subsection.

In *batch backpropagation*, each training iteration improves performance of the neural network with respect

to all examples in the training set simultaneously. In other words, the direction of gradient descent is an average of the optimal directions for each training example. In *stochastic backpropagation* training each weight update is tuned especially to improve the performance of $\hat{J}$ with respect to a single training example. Stochastic improvement vs. all examples happens over separate training iterations; gradient descent in beneficial directions is reinforced and unhelpful changes (in directions that increase performance vs. one example at the expense of another) tend to cancel out. The batch method is arguably more stable, but it must wait until new examples exist for all training examples before each training iteration can occur. In contrast, the stochastic method is less stable, but more convenient in that training is allowed to progress as soon as new neural signal data exists for any training example.

Backpropagation requires both a gradient descent update direction as well as an update step size. The step size is called the *learning rate* in neural network literature.

The overall learning rate is denoted γ, and the learning rate at layer ℓ is denoted γ_ℓ. We experiment with a number of different learning rate tuning methods that are used to calculate the size of each gradient descent step. For example, decreasing the learning rate such that it is always proportional to the inverse of iteration count is proven to converge, in the limit, as the number of iterations increases toward infinity; $\gamma_\ell = \gamma = \frac{c}{\tau}$ for all ℓ and some user-defined constant $c > 0$. Other methods that use heuristics to estimate the Hessian of the weight space have been shown to often perform better in practice, but have no formal convergence guarantees (and in which case γ_ℓ is often allowed to differ for each ℓ). All tuning rate methods we experiment with are discussed in detail in the appendix. In this section the subroutine $\gamma_\ell \leftarrow \text{tune\_learning\_rate}(\cdot)$ is used to represent whatever tuning rate method is being used.

The batch and stochastic variants of our distributed group mind neural network training algorithm appear in Algorithms 3 and 4, respectively.

Both versions are specially modified to run on a robotic swarm in which the neural signals are sent over unreliable wireless communication that may drop messages. The key modification is to have each robot n pause its own training weight updates if it gets more than a predefined number c_{wait} of training iterations ahead of a neighbor $m \in \mathcal{N}_n$. Note, however, that this pausing is accomplished in Algorithm 1 (described in the previous section), the output of which determines if a training iteration is allowed to happen. Iff (if and only if) a training iteration is allowed to happen, then Algorithm 3 or 4 is called, depending on which training method is being used.

To ease our presentation, both the batch and stochastic versions make use of the forward and backward pass subroutines presented in Algorithms 5 and 6, respectively. $\text{forward\_pass}(\mathbf{F}_i, h)$ is responsible for propagating neural data forward through the network (for both training examples, as well as the current environmental data). $\text{backward\_pass}(h)$ propagates weight update gradient information backward through the network so that nodes at earlier layers can tune their weights to improve performance.

Batch Backpropagation

Algorithm 3: backpropagation_training_iteration() (batch variant)

```
1  if all_new_data_exists() then
2      for (F_i, h) ∈ 𝒯 do
3          forward_pass(F_i, h)
4      for ℓ ← L, ..., 0 do
5          for (F_i, h) ∈ 𝒯 do
6              if ℓ = L then
7                  ε_ℓ^h ← ĥ_i − s_ℓ^{n,h}
8              else
9                  ε_ℓ^h ← Σ_{m∈𝒩} u_{ℓ+1}^{m,h} w_{ℓ+1}^{nm}
10             u_ℓ^{n,h} ← calc_update_parameter(s_ℓ^{n,h}, ε_ℓ^h)
11         if ℓ < L then
12             γ_ℓ ← tune_learning_rate(γ, ε, ℓ, τ)
13     for (F_i, h) ∈ 𝒯 do
14         backward_pass(h)
15     τ ← τ + 1
```

The learning rates at all levels are updated during the same training iteration (lines 11-12). If a cost gradient Hessian method is used then the learning rate update will utilize the change in error to heuristically estimate the Hessian (see Algorithm 13). After all update parameters have been calculated and learning rates adjusted, a backward pass is performed for each training example (lines 1-14). We note that $\hat{h}_i = \hat{J}(\mathbf{F}_i)$ is the current output of the group mind given input $\mathbf{F}_i$. Finally, we increase the iteration counter for the current robot (line 15).

Stochastic Backpropagation

The stochastic variant of a backpropagation training iteration appears in Algorithm 4. The basic idea is very similar to batch version (Algorithm 3); the main distinction is that we perform an update whenever we receive new information from any h, n, and ℓ — instead of waiting until a message is received for every h, n, and ℓ. The rationale behind this idea is that, although a single iteration may not take us down the cost gradient function, we still expect the cost to decrease when averaged over many iterations. The benefit of the stochastic method in the context of our application is that dropped messages do not prevent us from training with respect to whatever messages are successfully transmitted.

Forward Pass

A forward pass involves calculating the outputs for each neuron and sending the results forward through the network from neurons at layer ℓ to their neighbors at layer $\ell + 1$. Neural output (at each layer) is calculated by summing weighted input values and passing the result through an activation function[¶]

[¶]We use the hyperbolic activation function recommended by LeCun et al. (2012) in our experiments; however, many other activation functions exist and could also be used. In practice, input signals may need to be scaled

Algorithm 4: backpropagation_training_iteration() (stochastic variant)

```
1  for (F_i, h) ∈ 𝒯 do
2      if any_new_data_exists(h) then
3          forward_pass(F_i, h)
4          for ℓ ← L, ..., 0 do
5              if ℓ = L then
6                  ε_ℓ^h ← ĥ_i − s_ℓ^{n,h}
7              else
8                  ε_ℓ^h ← Σ_{m∈𝒩} u_{ℓ+1}^{m,h} w_{ℓ+1}^{nm}
9                  γ_ℓ ← tune_learning_rate(γ, ε, ℓ, h, τ)
10             u_ℓ^{n,h} ← calc_update_parameter(s_ℓ^{n,h}, ε_ℓ^h)
11         backward_pass(h)
12 τ ← τ + 1
```

Neural signals in the group mind are passed using wireless communication that runs in a separate thread. Therefore, the pseudocode in Algorithm 5 assumes that $s_{\ell-1}^{m,h}$ contains the most recent incoming signals and outgoing messages will be sent from $s_{\ell}^{n,h}$.

Algorithm 5: forward_pass($\vec{F}_n \in \mathbf{F}_i, h$)

```
1  s_0^{n,h} ← F⃗_n ∈ F_i
2  for ℓ ← 1, ..., L do
3      s_ℓ^{n,h} ← activation(Σ_{m∈𝒩} w_{ℓ−1}^{mn} s_{ℓ−1}^{m,h})
4  ĥ_i ← s_L^{n,h}
5  return ĥ_i
```

We maintain separate network forward pass computations for the real-time environmental data observed by the swarm as well as each training example. The value h specifies the source of the input data (e.g., corresponding to real-time sensor data or a particular training example). At the swarm level, running the forward pass subroutine (Algorithm 5) with a particular h—in parallel and asynchronously across all robots—generates the corresponding output signal from each neuron on each robot for that h. Due to the distributed nature of the algorithm, L message-passing rounds are required for the calculation associated with a particular h to propagate through the network. The computations for different h happen independently as Algorithm 5 is repeatedly called with different h from within Algorithm 3 or Algorithm 4.

The signals from the 0-th layer are the feature data $\vec{F}_n \in \mathbf{F}$ from a robot's sensor(s) (or the training data that was provided during the upload phase), line 1; in our experiments this is provided from a calibrated light sensor that returns values on the range $[-1, 1]$. The values at layers $\ell > 0$ are calculated by summing over $w_{\ell-1}^{mn} s_{\ell-1}^{m,h}$, the incoming link weights multiplied by the corresponding signals experienced for h, and then passing the result through an activation function (lines 2-3). The final output signal from layer L is returned as the raw $h_i = \hat{J}(\mathbf{F}_i)$ output at this robot given the group mind's current distributed calculation of $\hat{J}$ (lines 4-5). In general, it is possible to use a variety of activation functions; most take a form similar to a sigmoid in order to approximate a step function while maintaining continuity. The particular activation function used in our experiments is a hyperbolic tangent described in Algorithm 10.

Backward Pass

A backward pass is responsible for adjusting the link weights on robot n such that the error experienced by each neuron is reduced. A separate backward pass calculation is required for training example h.

Algorithm 6: backward_pass(h)

```
1  for ℓ ← L − 1, ..., 0 do
2      for m ∈ 𝒩 do
3          δ ← u_{ℓ+1}^{n,h} s_ℓ^{m,h}
4          w_ℓ^{mn} ← w_ℓ^{mn} + γ_ℓ δ
```

The backward pass subroutine is presented in Algorithm 6. The update signals γ_ℓ are calculated in Algorithm 3 or 4 (depending on the training method being used) before backward_pass(h) is called. γ_ℓ is multiplied by the signal weight to provide δ (line 3), where δ is the raw gradient descent update given h, n, ℓ. As discussed later in section 4, δ can be interpreted as the projection of the gradient descent update directional vector onto the weight-space dimension associated with w_ℓ^{mn}. The update to w_ℓ^{mn} involves multiplication by a learning rate γ_ℓ (line 4), where γ_ℓ is the magnitude of the gradient descent update step and is calculated in a variety of ways depending on the particular backpropagation variant used, e.g., one of Algorithms 12-14 presented in the appendix.

Analysis (Convergence of parallel stochastic gradient descent with dropped messages)

Let $\mathcal{W}$ denote the model space of a particular neural network with fixed topology and activation functions. Each decision function that can possibly be encoded by the neural network is represented by a point $\mathbf{W} \in \mathcal{W}$. The k-th component of $\mathbf{W}$ corresponds to the weighting factor along the k-th link in the neural network. The training error of a model can be defined in terms of a cost function $C_{\hat{J}} : \mathcal{W} \to \mathbb{R}$, such that $C_{\hat{J}}(\mathbf{W})$ defines a cost gradient over $\mathcal{W}$. Thus, training the neural network amounts to finding $\mathbf{W}$ such that $C_{\hat{J}}(\mathbf{W})$ is minimized. All known training algorithms work by performing gradient descent on $C_{\hat{J}}(\mathbf{W})$ and therefore achieve a *local* minima. The fact that a *global* minima is not guaranteed is a shortcoming shared by all neural networks, including our group mind neural network. That said, convergence to a local minima is important because it means that performance will stabilize.

We now prove that the distributed group mind neural network training algorithm will converge to a local minimum

appropriately for the activation function that is used. For example, at layer 0, the raw light sensor input data is rescaled from the output range of the sensor to the range $[-1, 1]$, as required by the activation function we use. Similarly, signal output values at internal layers exist on the range $[-1, 1]$, and output signals at the final layer are rescaled such that rounding them to the nearest integer yields a valid class index $\hat{h}_i$.

almost surely, in the limit, as the number of training iterations approaches infinity. The main contribution of this analysis is to show that lossy communication between neurons will not prevent local convergence (almost surely), provided the probability of message transmission is bounded away from zero.

Tsitsiklis et al. (1986) prove conditions of convergence for distributed asynchronous gradient descent algorithms; we use their analytical machinery as a starting point. In our analysis we use similar notation as in the algorithms presented in the previous section except that n and m now technically refer to processors instead of robots, and processors may compute on arbitrary subsets of nodes (indexed by l) instead of being restricted to a finite number of layers (i.e., which we indexed using ℓ in other sections of this paper). These slight notational distinctions help to generalize our results, but can likely be ignored by most readers.

Time is considered discrete and represented with the index τ. There are N processors that independently update (possibly overlapping) subsets of $\mathbf{W}$ and communicate using message-passing. $w_l^m(\tau)$ is the value, at time τ, of the l-th subset of $\mathbf{W}$ maintained at processor m. Note that the l-th subset of $\mathbf{W}$ may contain multiple elements. Also note that $w_l^n(\tau) \neq w_l^m(\tau)$ is allowed for $\tau < \infty$; however, message-passing facilitates mutual convergence, in the limit as $\tau \to \infty$. If a message containing w_l is sent from m and received by n at time τ, then the time (in the past) that the message was sent from m is denoted $t_l^{nm}(\tau)$ and the particular value w_l contained in the message is denoted $w_l^m(t_l^{nm}(\tau))$. The relative weight that processor n gives to w_l values sent from m and received at time τ is denoted $a_l^{nm}(\tau)$. The gradient descent update "step" $\delta_l^n(\tau)$ is a vector that points from $w_l^n(\tau+1)$ in a direction of local gradient descent. The particular calculation of $\delta_l^n(\tau)$ varies depending on the neural network training algorithm that is used. A step weighting factor is denoted $\gamma^n(\tau)$, and may either be constant vs. time or decreasing according to $1/\tau$.

Tsitsiklis et al. (1986) prove the convergence of asynchronous gradient descent algorithms that use the following update rule (the necessary assumptions are discussed later and in depth):

$$w_l^n(\tau+1) = \sum_{m=1}^{N} a_l^{nm}(\tau) w_l^m(t_l^{nm}(\tau)) + \gamma^n(\tau)\delta_l^n(\tau). \tag{1}$$

Equation 1 performs an update that uses a convex combination of the w_l values stored at all communication processors plus a small step in the direction of local gradient descent. By definition $a_l^{nm}(\tau) = 0$ for elements for which messages have not been received, and $\sum_{m=1}^{N} a_l^{nm}(\tau) = 1$. Also, the a_l^{nm} values do not need to remain constant vs. different τ, and so both $a_l^{nm}(\tau) = a_l^{nm}(\tau')$ and $a_l^{nm}(\tau) \neq a_l^{nm}(\tau')$ are allowed for $\tau' \neq \tau$.

At each time step a processor may transmit between 0 and all components of $\mathbf{W}$ to other processors, with messages that may arrive immediately or some time later. Each processor may also choose to perform some relevant computation (via a, δ and γ values) at each time step altering the state. In our distributed neural network each processor, i.e., robot, is responsible for maintaining the unique slice w_l containing the weights of its own neural links. However, in the general case of distributed gradient descent, the state updates can be distributed across processors such that each w_l is updated by between 1 and all processors. The weighting used to combine different updates to the state (i.e., that is performed on different processors) allows different and even contradictory updates to the same elements of the state on different processors. **Clearly Equation 1 represents a very general model—relevant to much more than our particular group mind neural network. The following proofs inherit this generality, and are relevant to any form of distributed gradient descent with lossy communication between processors.**

Tsitsiklis et al. (1986) use a notion of time in which at most 1 message is sent per time step. They note that this convention does not affect generality greatly because messages between different processors can be assumed to happen at slightly different times or some other method of arbitration used.

We now list all relevant assumptions made by Tsitsiklis et al. (1986) in order to guarantee gradient descent convergence to a local minima (the validity of each assumption will later be discussed in-depth). It is important to note that the following numbering schema is unique to our presentation; indeed, most of these assumptions are so common and/or straightforward that they are neither numbered nor discussed at length by Tsitsiklis et al. (1986):

Assumption 1: The initialization of the algorithm is random, and the distribution from which the random weights are drawn has finite mean and variance.
Assumption 2: The objective of the algorithm is to minimize a nonnegative cost function $C_{\hat{J}}$.
Assumption 3: $C_{\hat{J}}$ is defined on 0 to infinity. $C_{\hat{J}} : \mathcal{W} \to [0, \infty)$.
Assumption 4: $C_{\hat{J}}$ has a continuously differentiable nonnegative cost function with a Lipschitz continuous derivative. $\|\nabla C_{\hat{J}}(\mathbf{W}) - \nabla C_{\hat{J}}(\mathbf{W}')\| \leq K\|\mathbf{W} - \mathbf{W}'\|$ for some K such that $0 \leq K < \infty$.
Assumption 5: "In route" communication delays are bounded. $0 \leq t_l^{nm}(\tau) - \tau < c_1$, where $0 \leq c_1 < \infty$.
Assumption 6: The time duration between consecutive communications from one processor to another is bounded. $t_l^{nm}(\tau) - t_l^{nm}(\tau') < c_2$, where $0 \leq c_2 < \infty$, and for all τ' and τ such that a message is sent from n to m at τ' and the next message from n to m is sent at τ.
Assumption 7: Gradient descent updates satisfy the property that each component is, in expectation, going down the cost slope when conditioned on the past history of the algorithm. That is, $\mathbb{E}\left(\frac{\partial C_{\hat{J}}}{\partial w_l}(w^n(\tau))\delta_l^n(\tau)|\mathcal{F}_\tau^\Omega\right)$, with respect to the probability space $(\Omega, \mathcal{F}^\Omega, \mathbb{P}^\Omega)$, where $\mathcal{F}_\tau^\Omega$ is an increasing sequence of the smallest σ-algebra s.t. $\delta^n(k)$, where $k < \tau - 1$, and $w^n(1)$ for all n from 1 to N are $\mathcal{F}_\tau^\Omega$-measurable.
Assumption 8: Processors make progress toward convergence when they are not idle (assuming they have not already reached a minimum). $C_{\hat{J}}(w^n(\tau')) < C_{\hat{J}}(w^n(\tau))$, for all τ and τ' such that τ is an iteration in which n was not idle and τ' is the next iteration after τ that n is not idle.
Assumption 9: The variance of any stochasticity in the updates (i.e., stochasticity of $\delta_l^n(\tau)$ given the current history of the algorithm) goes to 0 in the limit for all n, l, and

τ. Formally, $\mathbb{E}\left(\|\delta_l^n(\tau)\|^2\right) \leq -c_3\mathbb{E}\left(\frac{\partial C_{\hat{J}}}{\partial w_l}(w^n(\tau))\delta_l^n(\tau)\right)$, where $0 \leq c_3 < \infty$.

Assumption 10: In the limit as $\tau \to \infty$, processor n updates component w_l either an infinite number of times or 0 times.

Assumption 11: Each component w_l has at least one processor that computes on it.

Assumption 12: There is a directed path of information flow from every computing processor n computing a particular component w_l to every other processor m computing the same component w_l.

Assumption 13: All update weights γ_l^m are positive, finite, nonzero, and deterministic (but not necessarily known beforehand).

Assumption 14: All combining coefficients $a_l^{nm}(\tau)$ are deterministic (but not necessarily known beforehand), and have a hard lower bound $a_l^{nm}(\tau) > c_4$, where $0 < c_4 < \infty$.

Assumption 15: Non-computing processors (i.e., those for which $\gamma^n(\tau)\delta_l^n(\tau) = 0$ for all n, l) with in-degree greater than 2 still share the hard lower-bound $a_l^{nm}(\tau) > c_4$.

Assumption 16: Each processor n has a buffer in its memory in which it keeps the element w_l of the state vector that it updates.

By inspection of Assumptions 1-16 it is clear that the model used by Tsitsiklis et al. (1986) already describes our scenario in the special case that no messages are dropped. However, it does not explicitly consider the effects of dropped messages. We show how it can be extended to handle dropped messages in the next section.

Dropped messages, general case

Dropped messages affect the recursive relation defined in Equation 1 by causing the term $a_l^{nm}(\tau)w_l^m(t_l^{nm}(\tau))$ to be replaced with 0 for each message that was sent from m to n and would have arrived at τ (if it had not been dropped). Furthermore, if updates $\delta_l^n(\tau)$ would have been triggered by the arrival of a message at n then $\gamma^n(\tau)\delta_l^n(\tau)$ is also replaced by 0. In either case, both $w_l^n(\tau')$ and $\gamma^n(\tau')\delta_l^n(\tau')$ for all $\tau' > \tau$ may be altered as a result of not performing the update $w_l^n(\tau+1) = \sum_{m=1}^N a_l^{nm}(\tau)w_l^m(t_l^{nm}(\tau)) + \gamma^n(\tau)\delta_l^n(\tau)$. Fortunately, this is not a problem. By construction, any infinite sequence $(\mathbf{W}(1), \mathbf{W}(2), \ldots)$ that results from performing distributed gradient descent with dropped messages due to lossy communication is identical to some other infinite sequence $(\mathbf{W}(1), \mathbf{W}(2), \ldots)$ that would have resulted from performing distributed gradient descent with lossless communication and simply never sending the dropped messages in the first place. The convergence of the latter sequence is guaranteed as long as it meets Assumptions 1-16. The only assumption at risk of being violated is Assumption 6, since dropping messages reduces the effective communication from m to n we can no longer guarantee an upper bound $c_2 < \infty$ on communication time between all n and m.

To compensate for dropped messages, we pause training on robot (processor) n whenever it is more than c_{wait} training iterations out of sync with n; in particular, whenever $\mathrm{out\_of\_sync}()$ evaluates to **true** in Algorithm 1. As we will prove shortly, this throttles the incrimination of τ in such a way that $c_2 < \infty$ is guaranteed[‖] ensuring the validity Assumption 6 with probability 1. Thus we are to achieve almost sure convergence in the case of randomly dropped messages. The aforementioned algorithmic modification is formalized in Assumption 17:

Assumption 17: Training is paused at n whenever it is more than c_{wait} training iterations out of sync with m (where m has previously communicated with n).

Almost sure convergence requires that, with probability 1, communication between n and m does not fail indefinitely. We formalize this requirement in Assumption 18, after the introduction of additional notation. The probability space of the k-th message transmission is defined $(S, \mathcal{F}^S, \mathbb{P}_k^S)$ with sample space $S_k = \{0, 1\}$, where 0 denotes the message is dropped and 1 that it is successful. $\mathcal{F}^S$ is the (smallest) collection of all possible outcomes, and $\mathbb{P}_k^S$ is the probability measure on $\mathcal{F}^S$ for the k-th transmission, where $\mathbb{P}_k^S : \mathcal{F}^S \to [0, 1]$. Let $failure = \{0\}$ and $success = \{1\}$.

Assumption 18: $\mathbb{P}_k^S$ are pairwise independent for all $k \in \mathbb{N}$ and $\sum_{k=1}^{\infty} \mathbb{P}_k^S(success) = \infty$.

In other words, the sum of probability of success, taken over an infinite number of trials, diverges without bound. The Second Borel-Cantelli lemma (Émile Borel 1909) tells us that Assumption 18 is sufficient to guarantee that, with probability 1, the infinite message transmission sequence $(S \times S \times \ldots)$ results in an infinite number of successful transmissions. We note that Assumption 18 is satisfied by many common models, including:

- A Bernoulli sequence defined by $\mathbb{P}_k^S(success) = c$ where $0 < c \leq 1$ for all $k \in \mathbb{N}$.
- The Gilbert-Elliott Markov communication model, assuming the nontrivial case where all state transition probabilities are greater than 0, and at least one state has nonzero probability of message send.
- A sequence such that $\mathbb{P}_k^S(success) = \frac{c}{\text{message number}}$ for any constant $c > 0$, and where "message number" is the number of messages that a processor has previously sent (successfully plus unsuccessfully).
- Any sequence such that the number of consecutive message drops is upper-bounded by a constant.
- Many other models.

This leads to the following Theorem.

Theorem 1. *Given Assumptions 1-5 and 7-18, the iteration in Equation 1 will converge to a local minimum with probability 1.*

Proof. Convergence is guaranteed by Tsitsiklis et al. (1986) whenever assumptions 1-16 hold. Of these assumptions, lossy communication between processors affects only Assumption 6. By Assumption 17 the algorithm pauses whenever Assumption 6 is in danger of being violated. The probability of indefinite pause is 0 by Assumption 18 and the Second Borel-Cantelli lemma, and so with probability 1 Assumption 6 holds. Thus, convergence to a local minimum is guaranteed with probability 1. □

[‖] The amount of throttling is a function of both c_{wait} and the network topology, given that neighbors of a paused node will only pause themselves once they become more than c_{wait} out of sync with the latter.

Verification that Assumptions Hold

Assumptions 1 and 2 hold by construction. We define $C_{\hat{\jmath}}$ to be squared error over output neurons over training examples which meets Assumption 3. However, in order to meet Assumption 4 we require that the activation function and network topology are "well behaved" as formalized in the following two assumptions:

Assumption 19: The derivative of the activation function of the neuron located at robot n and layer ℓ is Lipschitz continuous with Lipschitz parameter $K_{n,\ell}$ s.t. $0 \leq K_{n,\ell} < \infty$ for all n and ℓ.

Assumption 20: Each node communicates with a finite number of neighbors.

Together, 19 - 20 are sufficient to ensure that Assumption 4 holds; in particular, they are necessary to ensure the Lipschitz constant of $\nabla C_{\hat{\jmath}}$ is finite given that we are using squared error. Assumption 19 prohibits the use of any activation function that has a discontinuity—including a pure step function. However, it allows the use of other common activation functions such as sigmoids and hyperbolic tangents.

Assumption 5 is met by the physics of any wireless communication system. Assumption 6 is handled due to our introduction of Assumptions 17-18 in the previous section.

Assumption 7 can be met by choosing an appropriate update function. One possibility is to adjust the learning rates γ as a function of both the current cost values, the messaging parameter c_{wait}, and the Lipschitz constant K of the cost gradient $\nabla C_{\hat{\jmath}}$ (the latter can be bounded by the maximum neighborhood size and Lipschitz constant of the derivative of the signal function). However, this requires knowing the Lipschitz constant of $\nabla C_{\hat{\jmath}}$, which may be difficult or impossible to obtain, and may lead to a slower-than-necessary learning rate. Indeed, much neural network literature is dedicated to different methods of picking the update function such that this assumption is met**.

Assumptions 8 and 9 are true by construction as long as Assumption 7 holds. We enforce Assumption 10 by keeping robots stationary during training and observing that communication radii of each robot are non-shrinking—note that a less restrictive way to guarantee Assumption 10 is to use multi-hop communication between nodes and to enforce communication graph connectivity. Assumption 11 is true by construction. Assumption 12 is met as long as the communication graph is connected, which we can also ensure in practice. Assumptions 13, 14, and 15 are true by construction. Assumption 16 is true given our hardware. Assumption 17 is true by construction of the algorithm. Assumption 18 is true by the way our robots communicate — their communication range and rates are limited such that it is impossible to place enough robots in a 2-D area such that the communication channel is saturated. Assumption 19 is true for the activation functions we use. Assumption 20 is true given the physical constraints imposed by communication radii and robot radii.

Theorem 2. *Assuming that the update function meets Assumption 7, then the algorithms presented in Section 1 will almost surely converge to a local minimum when implemented on our hardware, in the limit, as the number of training iterations approaches infinity.*

Proof. Assumptions 1-5 and 8-20 are met for our hardware and implementation (as demonstrated above). Given Assumption 7 is met, then Theorem 1 guarantees that the gradient descent training of our group mind neural network will almost surely converge to a local minimum, in the limit, as the number of training iterations approaches infinity. □

We note that, in general, Assumption 7 is met by batch and stochastic variants of the backpropagation algorithm that use a decreasing learning rate (Algorithm 12) but *not* by the heuristic Hessian versions (Algorithms 13 and 14).

Swarm Behaviors and Actions Used in Our Experiments

A particular swarm behavior is defined by the collective actions of all robots $\{\alpha_1, \ldots, \alpha_N\} = \mathbf{B}$. In general, we assume a broad definition of "action"; a particular action can be defined by any piece of software code or state machine, and may include transitions to other actions (for example, based on sensor readings, messages, etc.) and/or other states in the group mind state machine depicted in Figure 7.

In this section we discuss the swarm behaviors classes, swarm behaviors, and robot actions used in our experiments; the experiments themselves appear later in the Experiments Section.

Swarm Behaviors and Actions used in experiments with stationary robots

One of our experiments involves a large stationary swarm, and robots display different color LED lights depending on which action they perform (as a consequence of the environmental feature data that is detected in the environment). Swarm behaviors tupels $(\mathbf{X}, \mathbf{B})$ cause different LED images to be collectively displayed across the surface of the swarm (i.e., so that an LED picture image is created when the swarm is viewed from above). Each robot displays a single color such that the swarm as a whole produces the desired LED image.

Different behavior sets are defined by different LED light patterns, as follows:

- $\mathcal{B}_0$ = collectively display nothing.
- $\mathcal{B}_1$ = collectively display the firewire symbol.

**That said, practical implementations of standard algorithms often forfeit this assumption (and thus convergence guarantees) in favor of more practical alternatives. Arguably, the easiest such alternative is to define γ as a small constant, in which case the algorithm is guaranteed to converge such that $\lim_{\tau \to \infty} \|C_{\hat{\jmath}}(w_l(\tau)) - C^*_{\hat{\jmath}}\| \leq \gamma K c_{wait}$, where $C^*_{\hat{\jmath}}$ is the cost at some minimum cost point. Another alternative is to assume a continuously differentiable cost gradient such that the Hessian has Lipschitz continuity, and then choose updates as a function of the Hessian directly (this is similar to Newton's method), or as a function of the Hessian Lipschitz constant. There is much literature regarding the use of such methods in batch backpropagation (which require additional information be passed in messages the enable the estimation of the Hessian) and also experimental results showing that a number of successful heuristics loosely based on Hessian information work well in the stochastic backpropagation setting.

- $\mathcal{B}_2$ = collectively display the wifi symbol.

Each action used in this experiment requires the executing robot to display a particular LED color (or to turn off the LED so that no color is displayed):

- z_0 = Off_LED.
- z_1 = Red_LED.
- z_2 = Blue_LED.

We restrict the size of each behavior set to 1, such that $\mathcal{B}_1$ involves displaying a particular wifi symbol and $\mathcal{B}_2$ involves displaying a particular firewire symbol (these are shown in Figure 13). However, we note that it would be possible for one behavior class to include two or more different images—if this were done, then a valid swarm behavior would require the swarm to display one of the images in the set, and a distributed consensus algorithm would need to be used to pick which image in the set was displayed.

Each behavior tupel $(\mathbf{X}, \mathbf{B}) \in \mathcal{B}_1$ that is used in this scenario contains a swarm behavior with actions $\{\alpha_1, \ldots, \alpha_N\} = \mathbf{B}$ such that $\alpha_n = z_1$ iff α_n exists at a location in the state space that maps to a red part of the firewire image, $\alpha_n = z_2$ iff α_n exists at a location in the state space that maps to a blue part of the firewire image. A complimentary definition exists for swarm behaviors in $\mathcal{B}_2$, except that the wifi symbol is used instead of the firewire symbol. See Figure 8 for an example.

The state machine used in state Act in this (large stationary swarm) experiment is displayed in Figure 9. Because the robots are guaranteed to remain stationary, it is possible to have the swarm change the image it displays in response to changing environmental feature data; this is accomplished by having each action state transitions back to the state Observe of the high-level algorithm (that is, in Figure 7). While training, the state machine continually transitions back to the Learn state until the swarm is trained, enabling us to watch how the LED images displayed by the swarm improve as the group mind neural network learns to recognize different environmental feature states.

We also run a similar but much smaller experiment with four robots. The high-level swarm behaviors and low level actions used in the four robot experiment are conceptually similar to those described above, and are described in detail in Appendix B. Differences include: (1) Only four robots are used in the small swarm experiments. (2) The neural network is taught to differentiate between four different environmental feature patterns instead of three. (3) Swarm behaviors are homogeneous in that each behavior requires all robots to display the same color; though a different color is displayed in response to each environmental feature pattern that the swarm recognizes.

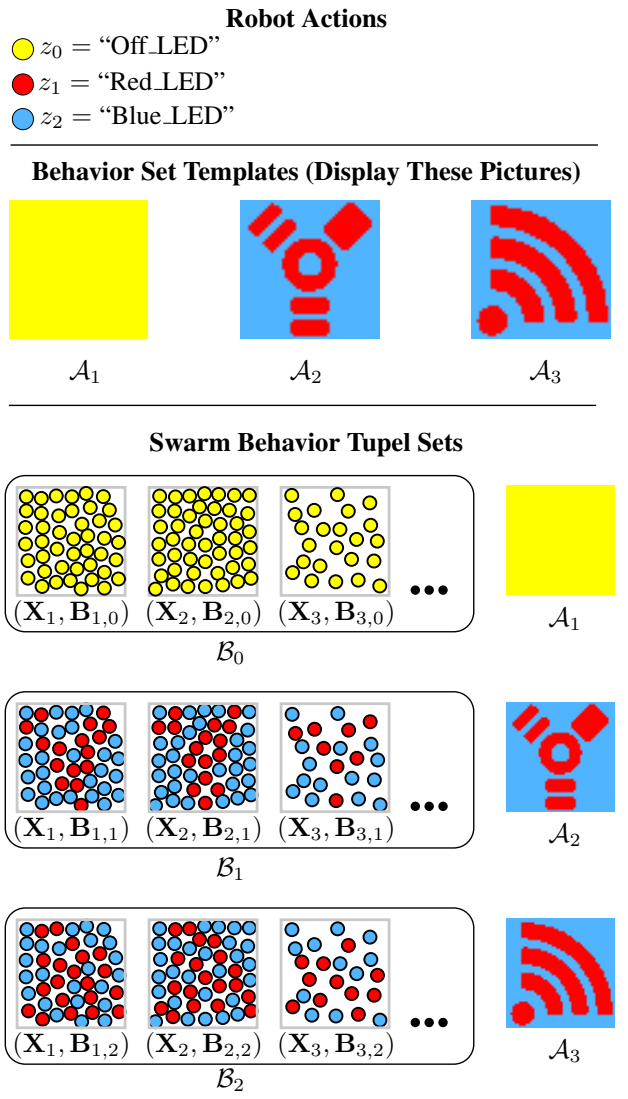


Figure 8. Robot actions and swarm behaviors used in the large swam experiment without movement. Each robot displays a single LED color such that an LED image is collectively displayed by the swarm. Different colors represent different robot actions (Top). Different swarm behavior set tupels $\mathcal{B}_0$, $\mathcal{B}_1$, and $\mathcal{B}_2$ (Middle) are defined by robots doing the action from the templates $\mathcal{A}_1 = o_1^{\mathbb{Z}}(\mathcal{X})$, $\mathcal{A}_2 = o_2^{\mathbb{Z}}(\mathcal{X})$, and $\mathcal{A}_3 = o_3^{\mathbb{Z}}(\mathcal{X})$ that maps to the position of their light sensor location, respectively; allowing the swarm to calculate $\mathbf{B} = o^{\mathbb{Z}}(\mathbf{X})$ for $o_1^{\mathbb{Z}}$, $o_2^{\mathbb{Z}}$, and $o_3^{\mathbb{Z}}$ given the swarm's current location (Bottom). The behavior tupels for three example swarm locations are shown within each behavior tupel set. In this example light sensors are assumed to be at the centers of robots.

Swarm Behaviors and Actions used in experiments with moving robots

In a separate set of experiments we define swarm behavior sets such that robots physically move to form shapes. Different behavior sets are defined as follows:

- $\mathcal{B}_0$ = collectively display nothing.
- $\mathcal{B}_1$ = collectively form a blue smiley face.
- $\mathcal{B}_2$ = collectively form a red frowny face.

Such shapes can be formed by having some robots move while others remain stationary. Thus, robot actions include both actions with movement and actions without movement. Moreover, stationary actions fall into two separate categories depending on the LED color that is displayed by a robot as it remains stationary. Robot actions include:

- z_0 = Train: transition to state Train.
- z_1 = Red_Attract: stop moving, display red LED, broadcast "attraction" messages.
- z_2 = Blue_Attract: stop moving, display blue LED, broadcast "attraction" messages.

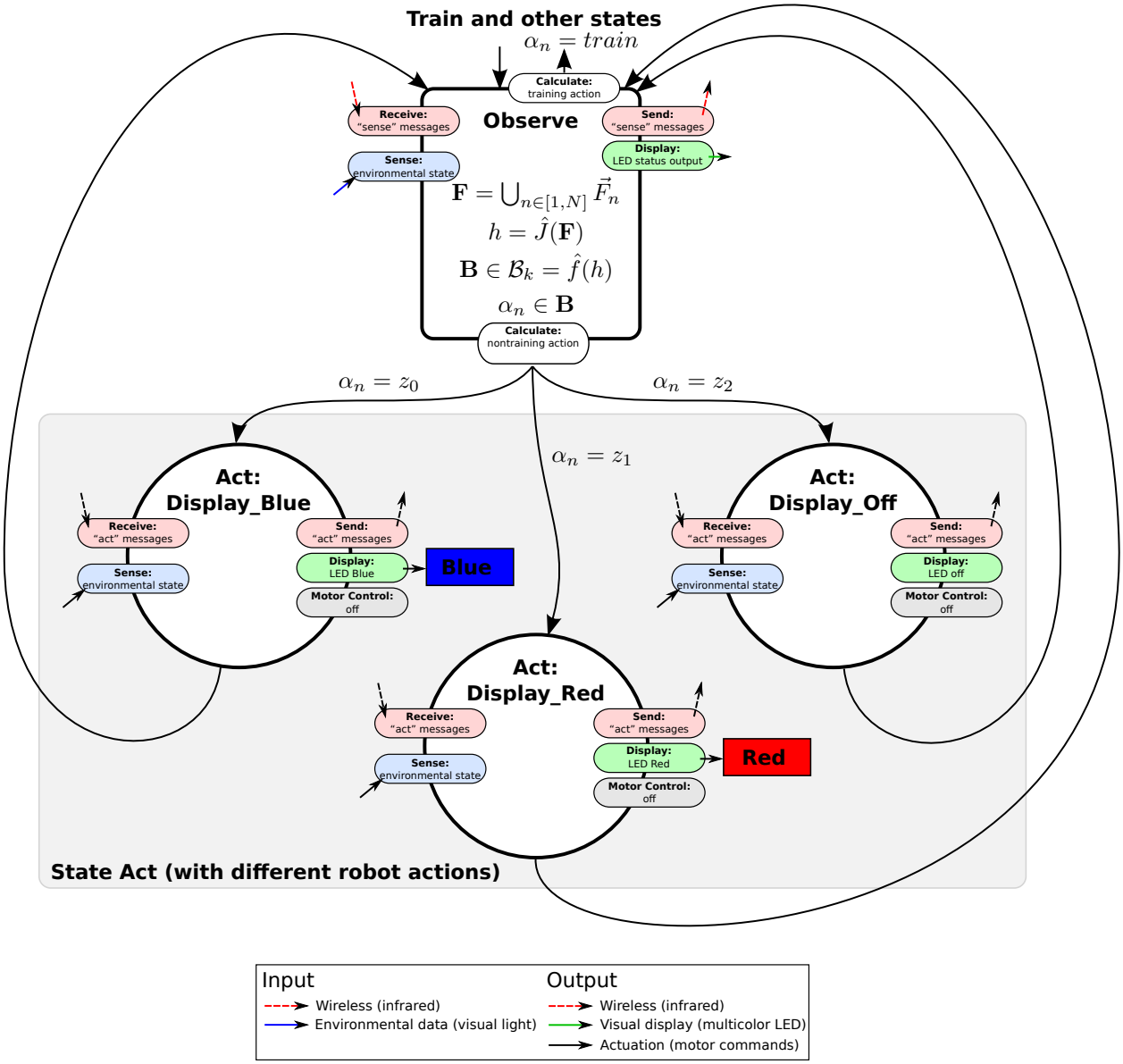


Figure 9. The various actions used in the experiment with a large swarm (316 robots) that does not involve movement. Because there is no movement, the group mind remains intact as long as the swarm is deployed. In this case, all robot actions transition back to state Observe, which is part of the overall state-machine described in Figure 7. Each action requires the robot to display a particular color (red or blue LED) or no color at all (by turning the LED off). This allows the swarm to collectively display an image across the environment using all robots' LEDs. Each state automatically transitions back to state Observe so that the swarm can update the image that it collectively displays as the group mind detects different environmental feature patterns (e.g., as the feature patterns change in the environment as the swarm is deployed). The environmental feature patterns used in the experiment are shown in Figure 13-B.

- z_3 = Rand_Search: move randomly while displaying a rainbow LED sequence, until receiving an "attraction" message.
- z_4 = Display_White: stop movement and display white LED, unless an "attraction" message is not received within a small amount of time.

The actions used in the experiments with moving robots are displayed in Figure 10. Movement breaks neural connections, which makes recomputation of $h_i = \hat{J}(\vec{F}_i)$ impossible once movement has started. Thus, robots do not transition back to state Train or Observe after beginning a non-training action.

The physical shapes formed by the swarm are an emergent property resulting from having robots within the desired shape remain stationary, and having robots outside of the desired shape randomly move until they either become part of the desired shape or leave the environment.

Experiments

We experimentally evaluate the group mind in a variety of scenarios. Experiments are performed both with a physical robot swarm and in simulation. Physical experiments are necessary to test how well the group mind works in practice, while simulation facilitates performing large numbers of trials to evaluate the convergence properties of the four different variations of the distributed backpropagation algorithm. We use both small (4 robot) swarms and large (up to 316 robot) swarms. The small swarms facilitate

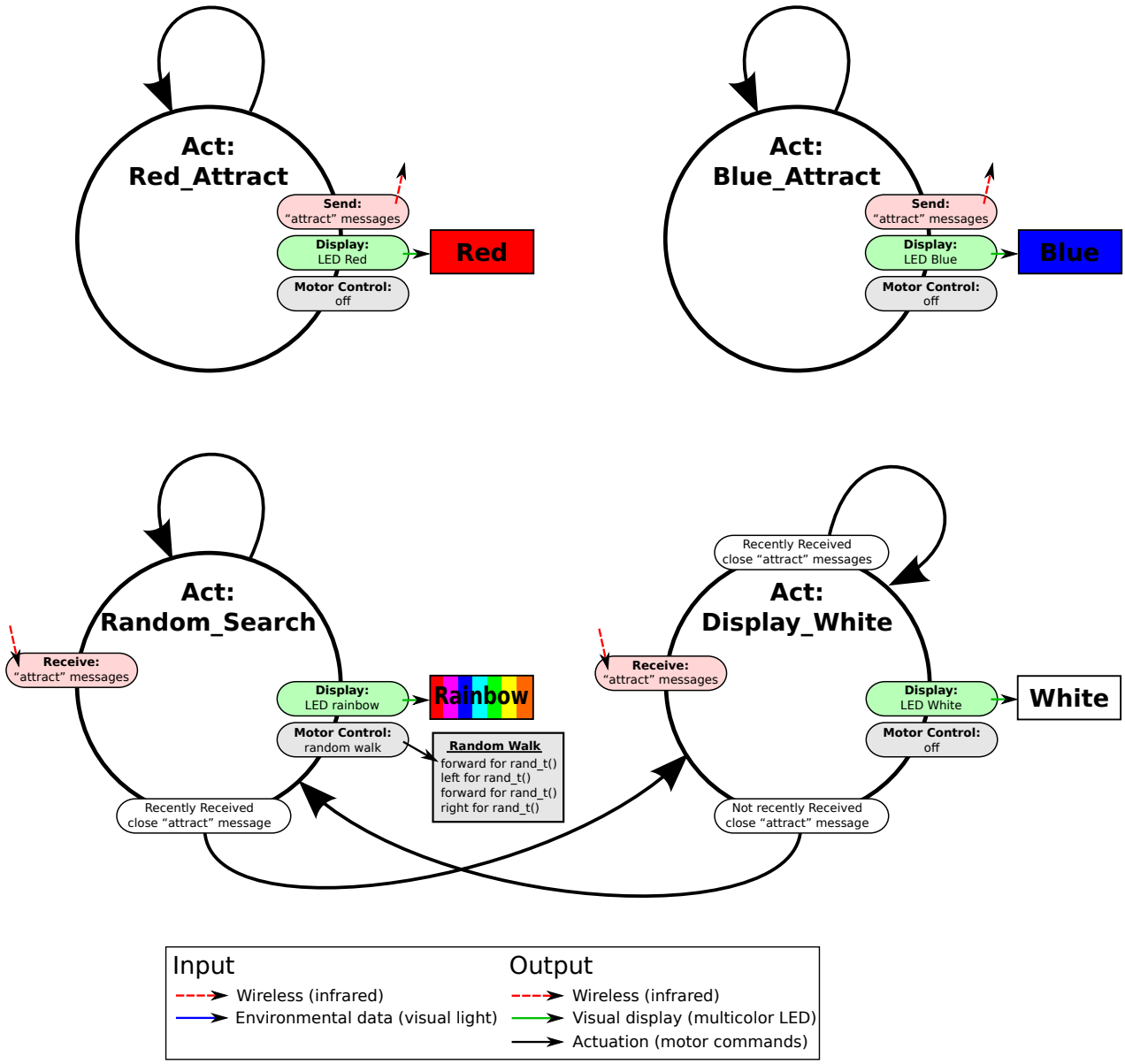


Figure 10. The various actions used in the experiment with a large swarm (167-262 robots) involving movement (and hence dissolution of the group mind), and the portion of the state machine connecting them with the overall behavior described in Figure 7. Red_Attract and Blue_Attract involve the swarm displaying red or blue LEDS, respectively, and broadcasting "attract' messages. Both of these continually self-transition, so a robot that starts in either of them will remain in it. Rand_Search and Display_White form a self-contained two-state state machine. Robots that receive a close "attract" messages (i.e., from a robot in Red_Attract or Blue_Attract that is less than 5 cm away) transition to (or remain in) Display_White. Robots that do not receive close "attract" messages within a small timeout period transition to (or remain in) Rand_Search, in which case they perform a random walk around the environment. Overall this causes the swarm to exhibit emergent behavior such that robots performing Rand_Search move toward robots performing Red_Attract or Blue_Attract which creates physical shapes in the environment. The environmental feature patterns used in the experiment are shown in Figure 14.

running repeated trials, while the large swarms demonstrate that the algorithm scales to hundreds of robots. We also use two different classes of behavioral responses: (1) the swarm displays different coordinated light patterns depending on global environmental input, and (2) the swarm physically constructs different shapes in response to the global environmental input. The details of both (1) and (2) are described in the previous section. Note that (1) is performed both with real robot swarms and in simulation, while (2) is only performed with real robot swarms.

Experimental Setup

Each experiment involves a swarm of physical robots or a swarm of simulated robots (we do not mix real and simulated robots). The swarm consists of 3.3 cm Kilobots by Rubenstein et al. (2014) (see Figure 1-B). Kilobots locomote via vibration, communicate wirelessly using infrared light (range 10 cm), have AtMega328 microprocessors (8 MHz 32K memory), visual light-intensity sensors, and a multi-color light emitting diode (LED).

Experiments with large swarms are performed on a surface created by placing a whiteboard on the ground. A digital light projector mounted above the environment controls

environmental light intensity patterns by projecting 50 by 50 pixel grayscale images onto the swarm. Light projections are used both for human-to-swarm communication and also to create various global light patterns which the swarm is trained to differentiate.

Grayscale images are projected onto the swarm (on the whiteboard) to provide training examples $\mathcal{T}_{h,i} = (h, \mathbf{F}_i)$ for each $\mathcal{Y}_h$ (during the "upload" phase) and to provide real-time global environmental data to the swarm (during the state Observe) which the swarm samples to get the current $\mathbf{F}$. Grayscale images are also used (during the state Upload) to project template images $\mathcal{A}_j = o_j^{\mathbb{Z}}(\mathcal{X})$ of swarm behaviors (where each robot action is associated with a particular light value); combined with the swarm's current location $\mathbf{X}$, this enables the swarm to calculate $\mathbf{B}_j = o_j^{\mathbb{Z}}(\mathbf{X})$ based on light data readings, and thus the tupel $(\mathbf{X}, \mathbf{B}_j)$ for each j.

In experiments with movement, robots are manually removed from the environment once they travel outside of the illuminated area. A digital video camera mounted on a tripod to the side of the whiteboard is used to capture videos (Videos have been included as supplementary material, and they are also available at `https://tinyurl.com/yasy3fl9` as a YouTube play-list; the full url also appears in our references (Otte 2016b)). Code is written in C and runs on Atmega328 microprocessors onboard the Kilobots.

In experiments with (only) small swarms of physical robots, training examples are hard-coded a priori. Robots are placed on white printer paper in non-direct natural light. Real-time small swarm training error (Figure 11-C-Right) is produced by modifying the algorithms, such that all robots output their internal training error in the form of a color LED code. These are recorded via a digital video camera and then averaged at 1-minute time increments to determine swarm training error.

Simulations are run in C on one core of a Dell Optiplex 3020 with 4 gigabytes of RAM. Simulated robots use nearly all of the same C code as the physical hardware experiments; except that code libraries related to hardware are replaced and simulate this functionality in software; in particular, message-passing, visual sensor input, and LED color output. Movement is not considered in simulation. Simulated robots are randomly distributed in a 200 cm area, have radius 5cm, and communication radius 25cm. Overlapping simulated robots are randomly re-positioned, if any simulated robots remain overlapping after 1000 re-positions then they are allowed to overlap. Simulated robots are programmed to have random and slightly different message loop cycles (distributed uniformly at random on the interval 0.5 seconds +/- 0.1 seconds). Control loops are run only if a message has been sent or received by a simulated robot. Messages are assumed to take 0.01 seconds to send. If any robot receives two messages at the same time than that robot drops both messages.

We evaluate four variants of the distributed backpropagation algorithm, including:

- **i** *Batch Decrease*, the batch weight update is used (Algorithms 3) and learning rates decrease as a function of iteration (Algorithm 12).
- **ii** *Stochastic Decrease*, the stochastic weight update is used (Algorithms 4) and learning rates decrease as a function of iteration (Algorithm 12).
- **iii** *Batch Tune*, the batch weight update is used (Algorithms 3) and a batch heuristic Hessian algorithm is used for on-line learning rate tuning (Algorithm 13).
- **iv** *Stochastic Tune*, the stochastic weight update is used (Algorithm 4) and a stochastic heuristic Hessian algorithm is used for on-line tuning of the learning rate (Algorithm 14).

Each method (**i**-**iv**) requires between one and three tuning parameters to be chosen a priori. These affect training speed and reliability. Because we do not know the optimal value of these parameters *a priori*, we follow the standard machine learning practice of selecting parameters based on performance vs. a *tuning data set* that is different from the *test data set*. We use the following procedure to select tuning parameters:

1. A parameter sweep in simulation vs. a tuning set yields simulation parameters.
2. The simulation parameters (from step 1) are used vs. the test set in simulation.
3. The simulation parameters (from step 1) are the *starting point* for a manual greedy search with the physical swarm vs. the tuning set. This search yields physical parameters.
4. The physical parameters (from step 3) are used vs. the test set on the physical robot swarm.

This process is repeated for each algorithm variant that is tested.

Experiments with a small robot swarm

In this set of experiments the group mind is created over a 4-robot swarm, where the robots are organized in a square. The group mind is taught to differentiate between 4 different global environmental input signals (Figure 11-A,B). We use the average performance over 10 repeated experiments for each measurement. Mean group mind classification error vs. training time is presented in Figure 11-C. To reduce figure clutter we only show results for the tuning data that use the particular tuning parameters selected for use vs. the test data.

Experiments with a large stationary robot swarm

In this experiment we randomly distribute hundreds of robots within a square environment. The group mind is trained to distinguish between three global light intensity patterns: (1) a pi symbol, (2) an on-off symbol, and (3) a "blank" pattern (Figure 12-B,C). When the group mind detects a non-blank pattern, then the swarm collectively displays a prescribed response image by having robots adjusts their color LED lights according to the appropriate response behaviors (firewire symbol or wifi symbol, respectively).

Simulated experiments involve 250 robots, are performed across all four algorithm variants. Datapoints reported for the simulated experiments represents the average performance over 10 random trials. Physical experiments involve 303-316 Kilobot robots; however, only one algorithm variant (**i** batch decrease) is evaluated and only one trial is performed per measurement used in the large physical robot swarm. The reason for the limited number of physical experiments (e.g.,

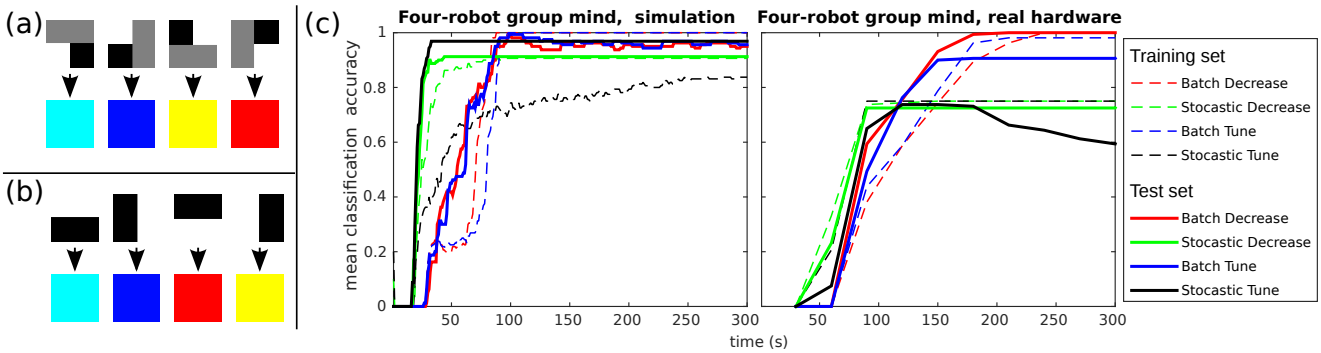


Figure 11. Small robot swarm experiment. Tuning (**A**) and test (**B**) sets, where the top row (grayscale) is the light intensity input patterns the swarm is taught to differentiate between, and the bottom row (color) is the corresponding class the swarm must learn to recognize. (**C**) Results depicting classification error vs. time for the swarm in simulation and physical hardware (left and right).

vs. the 4-robot experiments in the previous section) is due to the fact that running an experiment with hundreds of robots is quite laborious, and we prefer to focus our efforts on the case involving movement presented in the next section. We chose the method **i** batch decrease because it proved to be the most stable variant vs. tuning parameter selection in the small swarm experiments and in large swarm simulation.

Results depicting group mind classification error vs. time in simulation appear in Figure 12-A, while results of experiments with real hardware appear in Figure 13-E.

Figure 13 shows experiments in which the collective responses involve displaying 2-D color images across the swarm's distributed LED array. Robot behaviors include display "Red_LED," "Blue_LED," and "Off_LED." Thus, the swarm is able to display a yin-yang, wifi symbol, etc., by having different robots perform different behaviors (Figures 13 A-D). In these experiments, the swarm may safely use its group mind as it is being trained, allowing direct assessment of the group mind's training status via its improving collective behavior.

Experiments with a large robot swarm and actions involving movement

In this experiment the swarm learns to differentiate between a set of heterogeneous environmental light feature patterns (peace symbol, biohazard symbol, and blank pattern), and then perform a physical heterogeneous swarm response (make a smiley face, make a frowny face, and keep training) depending on the particular feature pattern that is detected across the environment at runtime.

Figure 14 depicts a set of experiments in which the response behaviors require physical movement to create one of two different shapes (blue smiley face or red frowny face) depending on which environmental feature pattern is observed at runtime (peace and biohazard symbols, respectively). Robot behaviors include: "Random_Search," "Red-Attract," "Blue_Attract," and "Continue_Training." Red_Attract and Blue_Attract cause a robot to broadcast "Attract" messages while remaining stationary and displaying red or blue LEDs, respectively. A robot performing Random_Search will move around the environment at random until receiving an "Attract" message sent from closer than 5cm, in which case it halts and displays a white LED. Physical shapes emerges as Random_Search robots move from their original positions to fill the space around attracting robots (or leave the environment). Videos of this experiment have been included as supplementary material, and they are also available at `https://tinyurl.com/yasy3fl9` as a YouTube play-list; the full url also appears in our references (Otte 2016b).

The "Continue_Training" behavior causes a robot to continue training until its training error has fallen below 5%, and then to display a yellow LED. By training the group mind to "Continue_Training" in response to a (uniform medium-gray) pattern displayed during training, the overall group mind training status can be evaluated by observing the proportion of the swarm displaying yellow LEDs.

Physical movement breaks neighborhood connectivity which causes the group mind to dissolve. Thus, the group mind must coordinate an organized deliquesce prior to the start of movement. Each robot n continually evaluates the group mind's calculation of n's output behavior based on the real-time distributed sensor data. If this behavior is not "Continue_Training" for more than a predefined length of time (30 seconds), then robot n begins performing the prescribed behavior while broadcasting messages indicating the pattern detected. Any robot $m \neq n$ in a poorly trained subset of the group mind can calculate its own behavior by combining the data from n's message with its own behavior map. m then performs the appropriate behavior and re-broadcasts the message from n.

This experiment is designed to demonstrate that a group mind can be used to coordinate swarm movement in response to a global environmental input pattern and is only performed with the physical swarm. Robots are trained to differentiate between three different input patterns, one of which is a "blank" pattern (as in the previous set of experiments). As long as the blank pattern is observed, robots output their training status by lighting a yellow LED once they have finished training. However, if/when the group mind detects one of the non-blank patterns, then it dissolves back into a decentralized swarm, and the swarm collectively moves to physically form one of two different shapes depending on which patterns is observed — a blue smiley face or a red frowny face depending on if a peace sign or biohazard symbol is detected, respectively (Figure 14). Figure 15 shows the swarm state from data upload to swarm action for a particular experiment trial.

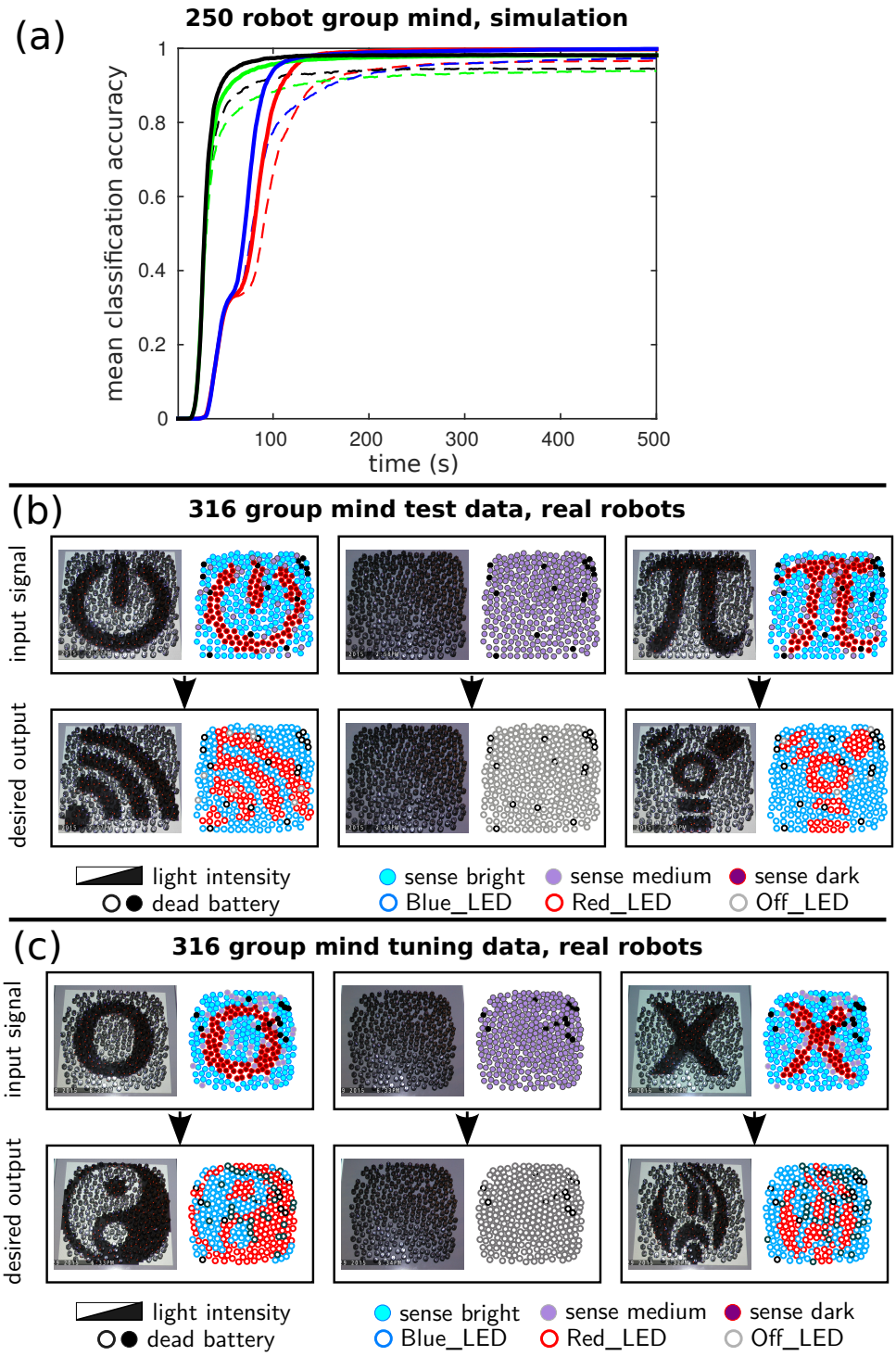


Figure 12. Large Swarm Experiments Without Movement. (**A**) group mind classification accuracy vs. time. The swarm's interpretation of the test data set (**B**) and tuning data set (**C**). In (**B**, **C**) The top row contains input signals and the bottom contains the light pattern used to communicate desired output behaviors to the group mind (different light intensity correspond to different behaviors). Each input/output is shown with the raw light pattern projected onto the swarm (left image in each pair) as well as the resulting intensity/class that is detected by each robot and reported via LED light color (right image in each pair).

Although the output behavior of the swarm (physical movement) is different from the previous experiment (display LED picture), the group mind training algorithm remains the same. Thus we use the same tuning parameters as in the large swarm experiment without movement. We perform five repeated trials of this experiment using between 167 and 262 robots in the swarm and evaluating both possible outputs. Batch decrease (variant **i**) is the only training variant tested in this experiment. Results appear in Figure 14-B.

Experiments with robot failures

In order to achieve the theoretical convergence guarantees that we proved in the Analysis Section, we require that robots pause their own training whenever a neighbor falls too far behind schedule. This works well as long as the robots themselves never malfunction. However, if a robot malfunctions such that it is unable to send messages, then its neighbors will cease training, and then their neighbors

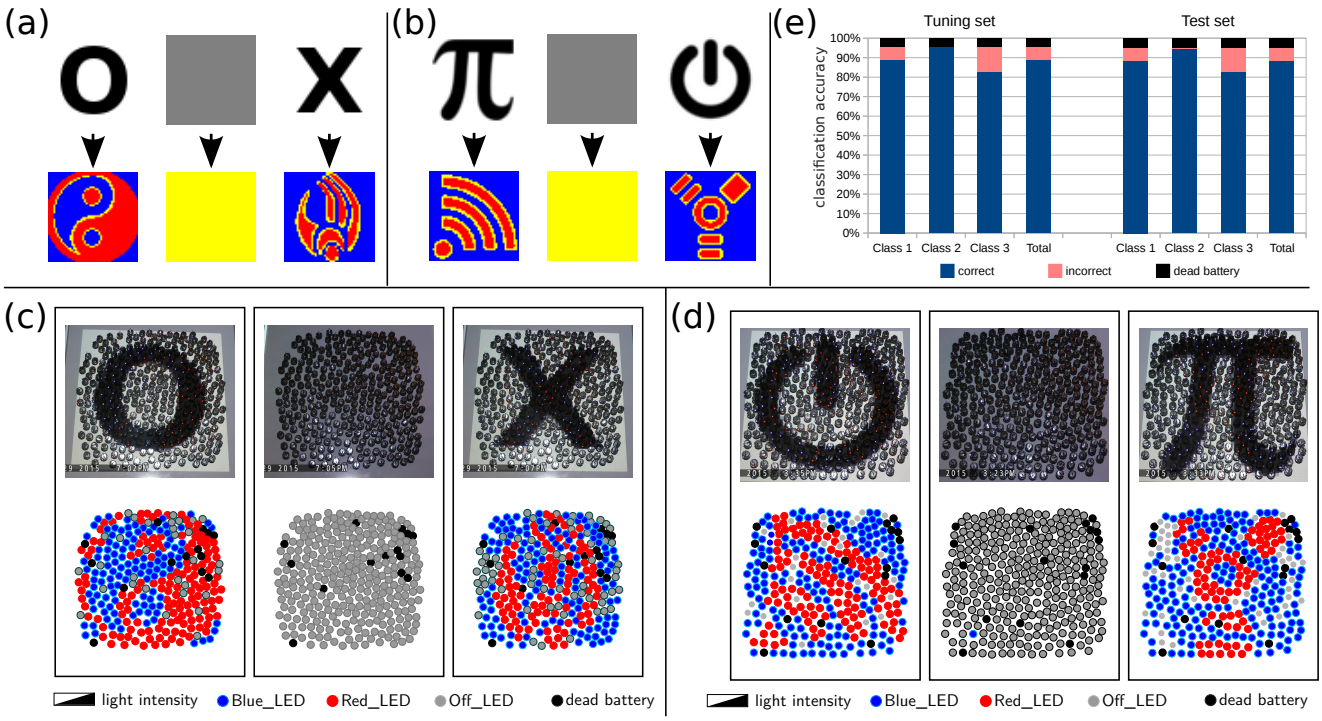


Figure 13. Large Swarm Experiment Without Movement on Real Hardware. The light pattern tuning set (**A**) and test set (**B**) used for experiments with 303 and 316 robot swarms, respectively. Top rows are the raw light intensity feature patterns and bottom rows are the corresponding output behaviors (represented by color) the group mind must learn to perform in response. Feature patterns projected onto the swarm (**C**,**D**) for tuning and test cases, and the behavior that was actually performed as a result. Classification accuracy vs. different input patterns (**E**). This Figure is reprinted/adapted by permission from: Springer Proceedings in Advanced Robotics, Vol 1, "2016 International Symposium on Experimental Robotics", COPYRIGHT 2017.

will cease training, and eventually the entire swarm will cease training. In this section we experimentally evaluate the effects of robot malfunctions on the group mind's ability to train.

We perform two sets of experiments, the results of which are depicted in Figures 16 and 17. In the first set we have robots pause training according to our algorithm, in the second we do not pause robots (and thus forfeit the theoretical convergence guarantees, but also eliminate the possibility that a malfunctioning robot pauses training on the entire swarm). In both sets of experiments we run repeated trials for the four different versions of the training algorithm that were previously discussed. Experiments are run in simulation to facilitate a large number of repeated trials.

Robots are programmed to malfunction according to an exponential Poisson decay process that has a known expected failure rate. For example, a failure rate of 10^{-3} means that the n-th robot is expected to malfunction once every 10^3 seconds, on average. All robots have the same failure rate in a particular experimental trial; thus, if the swarm contains N_{work} functional robots and the failure rate is 10^{-3} then we expect half of the robots, e.g., $N_{\text{work}}/2$, to fail after 10^3 seconds have elapsed.

Failure rates are depicted in Figures 16 and 17 as different line styles, and each datapoint represents the mean result from 50 trials. Once the n-th robot malfunctions, then the n-th robot will not send or receive messages for the rest of that experimental trial. We note that the class error rates that are shown in Figures 16 and 17 are with respect to the set of robots that has not yet malfunctioned by a particular time.

Discussion

Discussion regarding communication

Communication and layer depth: We use a single hidden layer in our experiments due to the slow communication rate (2 Hz) and small packet size (10 bytes) of the Kilobot robot platform. However, it is theoretically possible to use a group mind neural network of any depth. Since deeper networks are arguably capable of learning a larger set of functions, it is natural to ask: what negative ramifications would using a deeper group mind have in practice?

Assuming that both hardware and communication bandwidth are held constant, then using a deeper network is likely to increase training time (as with any type of neural network). Increasing layer depth will also require more data to be passed between robots as part of the backpropagation algorithm, since each robot will be responsible for managing more neurons. As more and more communication becomes necessary, the communication channel will eventually become saturated. While total communication failure can be avoided by throttling the rate at which data is sent, this would tend to increase training time even more. Data storage capacity may also be a concern for lightweight platforms such as the Kilobot, but would not be an issue for larger platforms.

That said, one of the take-home messages of our work is that even shallow neural networks can be useful—and these

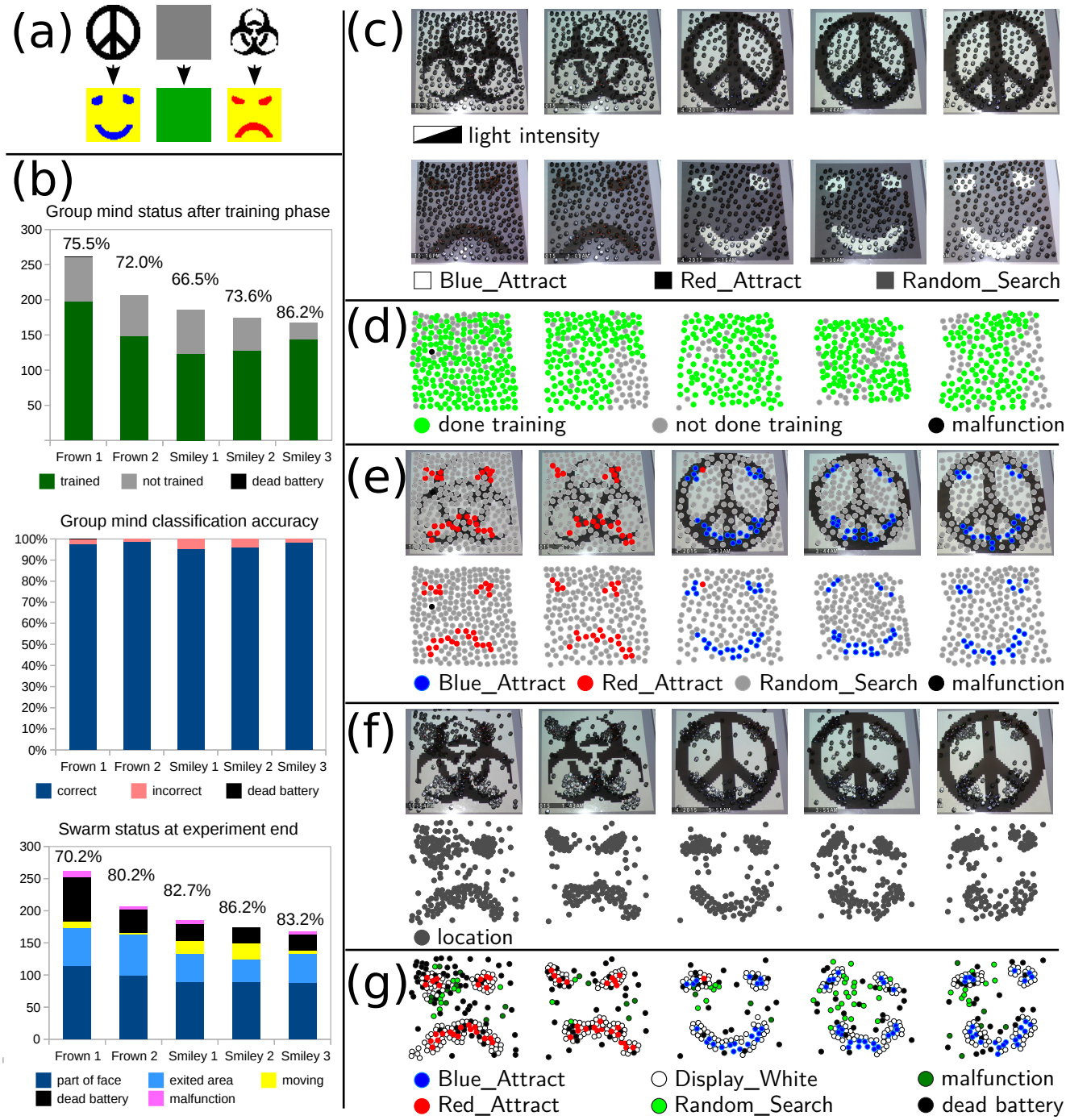


Figure 14. Large Swarm Experiments With Movement on Real Hardware. Light intensity pattern test set (**A**) for experiments with movement. (**B**) Results (top to bottom): the swarm's training status when movement started, classification accuracy vs. the raw light intensity pattern that initiated movement, and the breakdown of the swarm's behavior at experiment end. (**C**-**G**) Columns correspond to experiments. (**C**) Training data (light intensity and output behavior pattern, top and bottom) for the behavior that was eventually chosen. (**D**) Training status when movement started. (**E**) Real-time light intensity feature data and resulting output behavior (top, top and bottom). (**F**) Swarm position at experiment end. (**G**) Swarm behavior at experiment end. Videos of this experiment have been included as supplementary material, and they are also available at `https://tinyurl.com/yasy3fl9` as a YouTube play-list; the full url also appears in our references (Otte 2016b). This Figure is reprinted/adapted by permission from: Springer Proceedings in Advanced Robotics, Vol 1, "2016 International Symposium on Experimental Robotics", COPYRIGHT 2017.

can be trained within a reasonable amount of time, even on lightweight platforms such as the Kilobot.

Communication range effects: In general, communication range (combined with layer depth) is directly tied to the number of robots that are involved in a decision in any particular part of the network. If an area of the network that is larger than $radius * depth$ has training data such that all inputs are the same in this area for all training examples, then it is possible that robots at the center of the area will not be able to train a discriminative classifier. Such robots will need to rely on the classification performed in other parts of the network to inform them of what feature pattern has been detected.

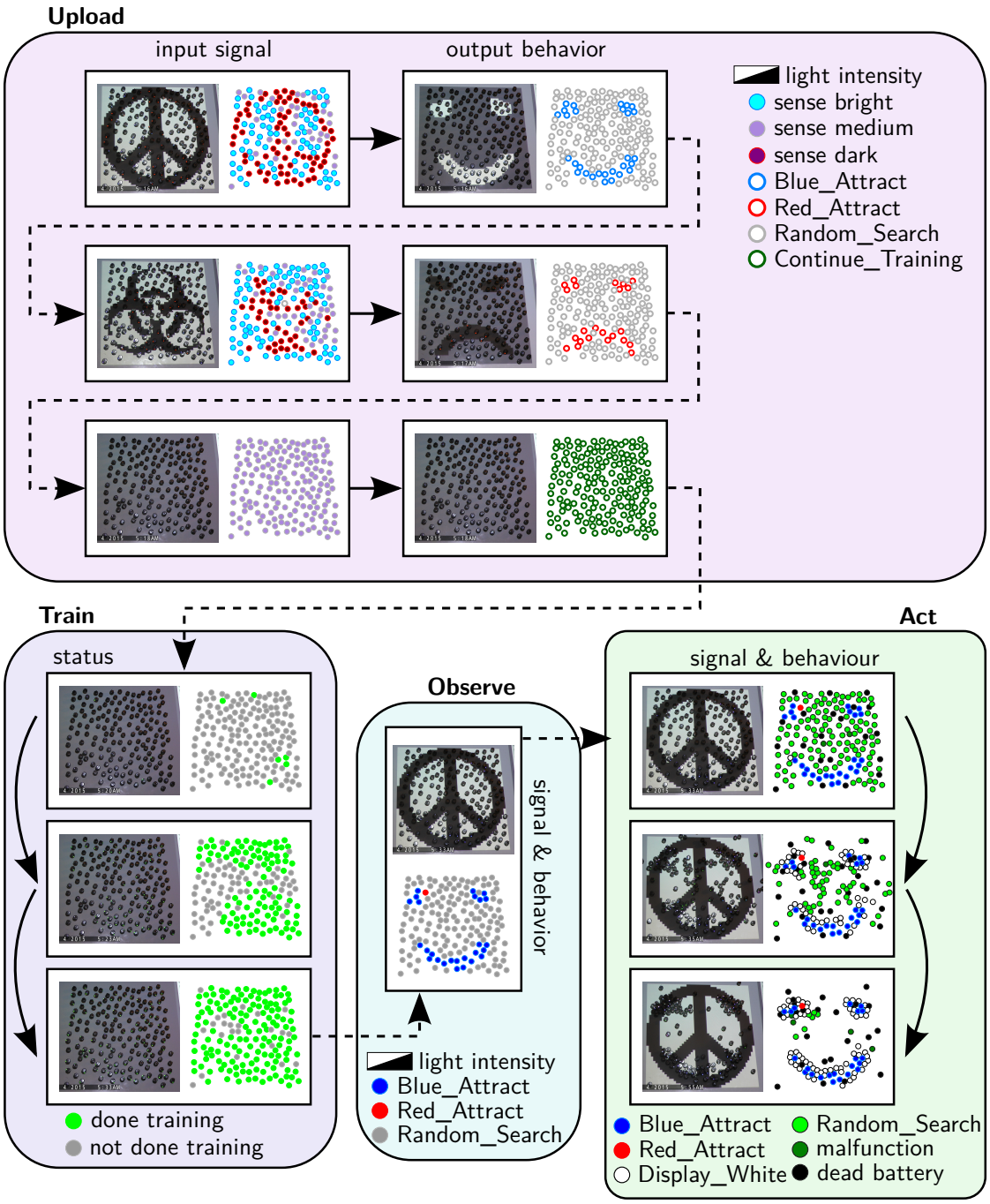


Figure 15. Diagram of group mind interaction and movement. All images appear as pairs with the raw light intensity on the left/top and robot status on the right/bottom. Data upload shows light intensity patterns and behavior profile that the group mind believes it should learn to recognize and perform. During training robots display a green LED once they finish training their portion of the group mind. When a non-blank pattern is projected onto the swarm, the group mind is used to recognize which pattern it is, and which behavior each robot should perform in response. In this case, a peace symbol is detected so robots move to create a blue smiley face outlined in white.

The communication range of the Kilobot robots are limited to about 10cm in practice. Therefore, in our experiments, each robot aggregates discriminative data from about a 30 cm radius by the final output layer depth.

Training time vs. training iteration: The convergence guarantees that we proved in the Analysis Section assume that neighboring robots do not get too many *training iterations* out of sync. If a robot falls too far behind its neighbors (in training iterations) then training is paused on its neighbors until the lagging robot catches up. This ensures that communication is never lost for more than a user-defined number of *training iterations*; however, it also means that dropped messages may cause a small number of training iterations to be stretched over a very long period of *time*.

Of critical importance is that we ***do not*** assume a particular message will be transported within any particular amount of ***time***. However, given our other assumptions, the amount of *time* over which any particular *training iteration* will be stretched is finite with probability 1.

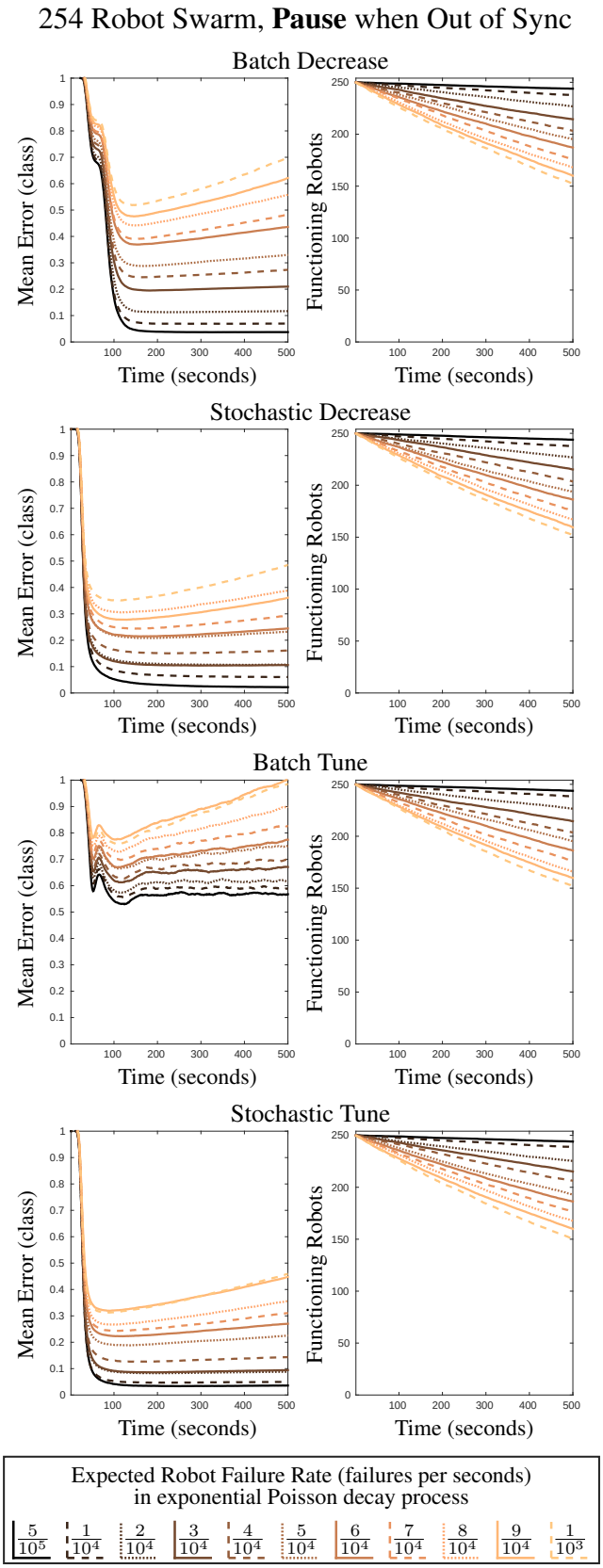


Figure 16. Mean classification error (left) and the corresponding number of functioning robots (right) when different methods (top to bottom) experience various rates of robot failure (different colors). This figure shows performance when the training iterations are paused on robots whenever a neighbor is more than 100 raining iterations behind. Data points represent the mean over 50 trials.

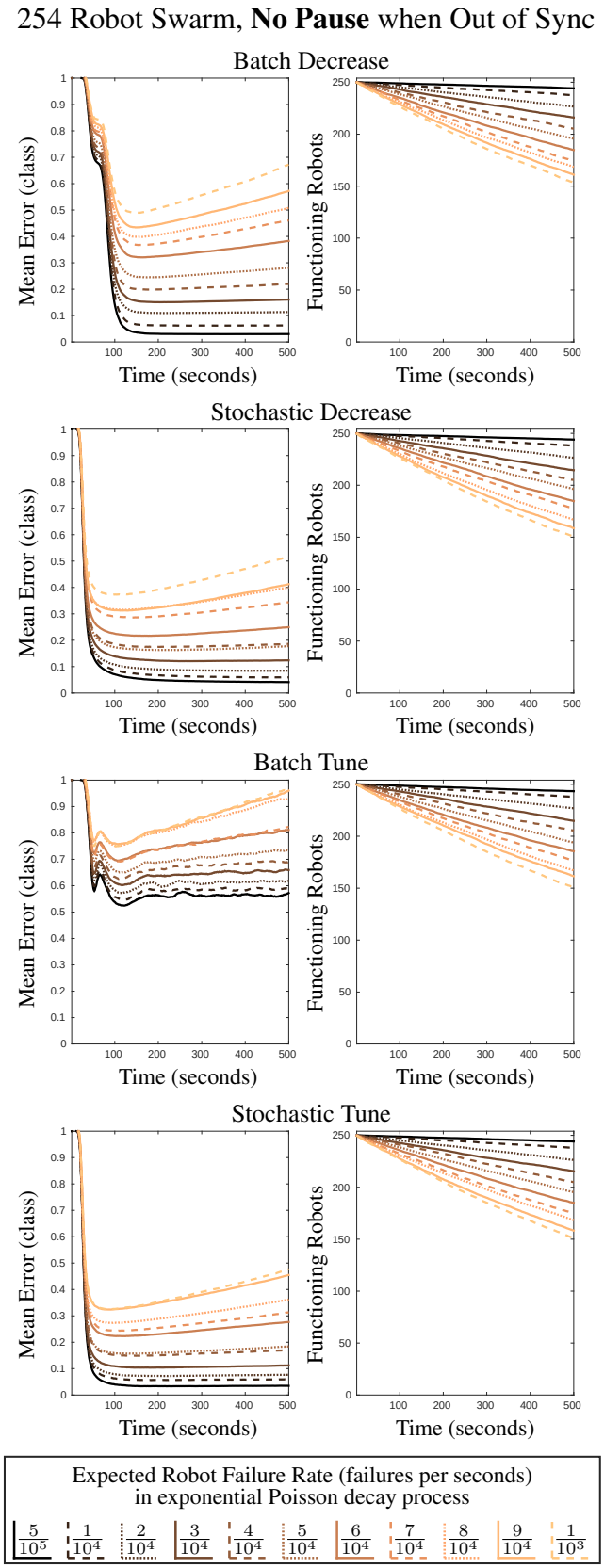


Figure 17. Mean classification error (left) and the corresponding number of functioning robots (right) when different methods (top to bottom) experience various rates of robot failure (different colors). This figure shows performance when the training iterations are never paused (and neighbors' iterations are allowed to become increasingly out of sync). Data points represent the mean over 50 trials.

Discussion regarding malfunctioning robots

Drained batteries were an unexpected difficulty that we encountered in our experiments. The chances of experiencing drained batteries can be minimized by replacing batteries prior to an experiment and/or modifying the hardware or output behaviors to be more power efficient. However, one reason for using swarms is their robustness to partial loss. In experiments where color LED images were the desired output, a dead battery simply meant that a particular robot's LED was dark. In experiments with movement, the desired output shape was discernible despite moderate (up to 26%) loss. Thus, we did not observe dead or malfunctioning robots leading to problems in practice.

However, in terms of algorithmic performance, if a robot loses power during the training process then its neural signals freeze from its neighbors' point of view. Because each robot n pauses training after becoming c_{wait} iterations out of sync with any neighbor m, power loss on one robot can potentially pause training across the entire swarm. Although a full-scale failure was not observed in the experiments, this is clearly a weakness of our algorithm.

We believe that we did not observe this type of failure in our experiments because robots were allowed to be up to 100 training iterations out of sync, which gave most of the swarm adequate time to train before any robots lost power. Another reason may be because if such a frozen signal is detrimental to a neighbor's neural performance, then that signal will be weighted less and less over time as part of the training procedure.

Modifying our algorithm such that robots cease interaction with uncommunicative robots may be able to eliminate this problem. Such a modification may still require treating the last known signals of pruned robots as if they remain fixed values. Although this would technically break the asymptotic theoretical convergence guarantees, these guarantees are also broken whenever a robot becomes permanently uncommunicative (and thus forfeit in the event of power loss anyway).

Discussion regarding training the group mind

Comparison between training algorithms: The most reliable training algorithm variant tested resulted from batch link weight updates (Rumelhart et al. 1986) combined with learning rates proportional to the inverse iteration number (Tsitsiklis et al. 1986). Stochastic updates (Bottou 1991) and/or heuristic Hessian tuning (Silva and Almeida 1990) often yielded quicker convergence but suffered from increased tuning parameter sensitivity. Heuristic Hessian methods also required an order of magnitude more pre-computation for parameter selection, and were prohibitively expensive for use on large swarms.

One difference between batch and stochastic versions of the modified backpropagation algorithm is that the batch version must wait until it has data regarding all training examples before performing a training iteration. As a result, the batch method is expected to train more slowly than the stochastic method, but it will tend to weight each training example more equally over short periods of time.

Transforming parameters from one problem to another: We find that tuning parameters learned on one (tuning) problem can successfully be used on a different (test) problem. This is obviously a prerequisite for the group mind to be useful in practice, given that we may not always know what a swarm will be required to learn during any *a priori* training phase.

Using simulation to expedite parameter tuning: Our results strongly suggest that simulation can and should be used to guide/bootstrap the search for optimal tuning parameters on the physical swarm. On the other hand, applying tuning parameters learned in simulation directly to the physical system does not always work (although batch methods appear to be more robust than the stochastic methods in this regard). We attribute this discrepancy to the fact that there are many aspects of the physical swarm that the simulation fails to capture. We suspect that more accurate simulations will yield tuning parameters closer to those optimal for the physical system.

Summary of convergence properties: The convergence properties that we derive in the Analysis Section show that, if robots pause training whenever their neighbors fall too many training iterations behind (and no robots malfunction), then the group mind will converge in the limit, at the number if iterations increases, with probability one.

This guarantee is technically valid only for the batch and stochastic variants of our modified backpropagation algorithm that use a decreasing learning rate. It is not valid for the heuristic Hessian variants; even though the heuristic Hessian variants sometimes have quicker convergence in practice.

The ability to pause the neighbors of a robot whenever that robot falls too far behind is necessary to ensure almost sure convergence in the limit. An implicit assumption is that the lagging robot continues to train and communicate so that it can eventually catch back up. This assumption is obviously broken if robots fail or otherwise malfunction.

In practice, it seems reasonable that we may decide to sacrifice theoretical convergence in the interest of possible practical gains, by simply letting robots go out of sync. The results of experiments comparing our original algorithm to this idea appear in Figures 16 and 17, respectively. These figures show that, although performance decreases as more robots fail, the observed difference in performance between pausing and not pausing is relatively small in the experiments that we run. This was an unexpected result and may indicate that the loss of connectivity that happens due to a robot's failure or malfunction is detrimental to that robot's neighbors' chances of training well.

Discussion regarding robot actions, swarm behaviors, and emergent swarm behaviors

Homogeneous swarm behavior: After the group mind detects and classifies the environmental state pattern that is present in the environment, the swarm performs a specific heterogeneous behavior that the user has designed specifically for that environmental state pattern.

It is worth emphasizing that it is only the swarm's behavior that is heterogeneous and not the robot hardware that we use in our experiments. The behavior is heterogeneous because different groups of robots run different action programs. In

contrast, much of the existing work in the swarm robotics community assumes that all robots run identical single robot actions (e.g., all robots run the same distributed control algorithm, or all robots forage, or all robots run a consensus algorithm, etc.) such that both the robots' behaviors and hardware are homogeneous. We believe that having heterogeneous and not homogeneous robot actions can lead to more interesting emergent swarm behaviors.

Nothing prevents our method from being used on a swarm consisting of heterogeneous hardware. However, each *type* of robot would need to be told which of its own possible actions it should run in response to a particular feature set class being active in the environment.

Designing behaviors: We define a swarm behavior tupel to contain both a set of robot actions and a corresponding set of robot positions. Actions are essentially "any program a robot might care to run". A robot does not need to know its own position explicitly, as long as it performs the correct action for its position. This definition of robot swarm behavior is simple and concise, but it does not answer the (broader) question of how to assign a set of robots actions such that a desired *emergent* behavior is the long-term output product. How to design useful emergent behavior is not a question that our work answers. Rather, we provide a tool that can be used to generate a rich set of finely-tuned emergent behaviors; a tool that also enables these emergent behaviors to be selected by the swarm as a function of the global environmental state it detects at runtime.

This tool potentially increases the number of problems that swarms can be used to solve; however, it is up to the robot swarm programmer to figure out which actions performed by which subsets of robots are likely to result in an emergent behavior that solves a particular problem.

Behavioral complexity: The simple actions used by the robots in our experiments could easily be replaced by more sophisticated actions with no change to the training and decision making algorithms. Other environmental features (chemical, temperature, acoustic, etc.) and their combinations could easily be used in place of light intensity.

Discussion regarding the group mind concept

Proof of concept: We have performed a variety of experiments on real robot swarms containing from 4 to 316 robots that demonstrate a group mind can be created within a swarm of robots and trained to react differently depending which complex environmental pattern it senses in the environment. Our work shows that an artificial group mind can emerge as the result of distributed computation over an ad hoc wireless network that emerges at runtime as robots discover and form wireless neural links with their neighbors. The ad hoc process in which neural connections form in the group mind is a departure from traditional ANNs, and echoes similar emergent neural linking in the animal brain.

Our work also demonstrates that an artificial group mind is a useful tool for solving the "trained at runtime heterogeneous swarm response" human-swarm interaction problem. That is, a swarm already deployed in the environment can be programmed to perform different swarm behaviors in response to different inputs that it detects. This enables fine-grained heterogeneous swarm behavior to be programmed at runtime and at a high level by a human user.

Group minds vs. swarms: The group mind is one of many different swarm algorithms, where a "swarm algorithm" is arguably any algorithm that runs on a group of robots that is able to scale, without major difficulty, if the number of robots in the group is increased by a few orders of magnitude.

That said, the swarm community is often interested in focusing on a handful of specific algorithmic traits that tend to be part of many algorithms that scale well. We now discuss the group mind's possession of some of these traits, ending with a discussion of scalability, in general.

Self-configuring: The connections between neurons on different robots emerge as a property of robot positions.

Self-optimizing: Given the emergent neural structure, the swarm trains itself to differentiate between user-provided patterns. This training involves the swarm autonomously tuning the weights between neurons using a distributed backpropagation algorithm that runs over the emergent ad hoc wireless network.

Self-healing: The group mind is self-healing in the sense that the swarm adjusts for dropped messages between neurons. While total robot failures during the group mind training process are not explicitly handled, each robot remembers the last signal sent from all robots Thus, further training will mitigate (or eliminate) the effects of the stagnant signals from failed robots as the weights of the uninformative signals are decreased.

Scalability: The group mind itself scales well because it relies on local communication and assumes all robots received the same programming *a priori*. Once the group mind has dissolved, then the swarm inherits the swarm properties of whatever distributed behavior is being performed by the swarm. Any standard swarm algorithms can be run, as well as variations that can benefit from: (A) different subsets of robots performing different actions and/or (B) knowledge of the global environmental pattern that has been detected by the group mind.

Why call it a group mind?: We believe that "artificial group mind" is an appropriate name for a distributed neural network that spans across a swarm of robots and uses wireless communication to transmit neural data between robots. In popular culture and science fiction there is a well established history of calling an autonomous-agent-spanning computation system a "group mind". Thus, we did not invent the term; we engineered and tested an *artificial* system that does what the term "group mind" already describes.

Conclusions and Summary

We pose the "trained at runtime heterogeneous swarm response problem" in which a swarm of robots must: (1) learn to distinguish between different environmental state pattern classes that it senses using the swarm's distributed sensors; (2) perform a particular swarm response behavior prescribed for the class of the state pattern that is observed. Each swarm behavior is defined by different subsets of robots performing different actions (single robot programs). Robots come pre-loaded with a library of individual single robot actions; however, (3) the specific environmental state pattern

classes the swarm must distinguish between and the mapping function from class index to swarm behaviors are provided by a user at runtime, after the swarm is already deployed in the environment.

To solve this problem we propose a new form of emergent distributed neural network that we call the "group mind". In the group mind, each robot maintains a set of neurons and forms wireless neural connections between its own neurons and the neurons on neighboring robots. Neighbors are discovered at runtime via local ad hoc wireless communication. The group mind is trained to differentiate between the environmental feature patterns using an asynchronous distributed backpropagation algorithm we have modified especially for the type of unreliable wireless neural connections that are used.

Using swarms of 4 to 316 Kilobot robots, we experimentally demonstrate that the group mind is capable of solving various instances of the "Trained at runtime heterogeneous swarm response problem". We also compare four different variations of the backpropagation training algorithm when used within a group mind, and prove that two of them will almost surely converge to a solution, in the limit, as the number of training iterations increases.

In order to guarantee this convergence property, despite dropped wireless messages, robots must pause their own training whenever a neighbor falls to far behind. A practical disadvantage of this pausing is that a malfunction or dead battery on a single robot can potentially cause the entire swarm to pause training (although we did not observe this in our experiments). In additional experiments designed to evaluate performance when robots fail, we find that dropping the pausing requirement does not appear to affect practical performance in the event of robot malfunctions.

We find the group mind is a powerful tool for human-swarm interaction, enabling new types of heterogeneous swarm behavior, and enabling swarm behavior to be a function of the global environment state that is observed at runtime.

Acknowledgments

This work would not have been possible without Michael Rubenstein and Radhika Nagpal, who designed and built the Kilobot platform. Michael Rubenstein taught the author how to use the Kilobots, and provided invaluable feedback on this work. The author is grateful for the time, knowledge, resources, encouragement, and advice provided by Radhika Nagpal and Melvin Gauci. The author is also grateful to Derek Kingston for providing the time, space, and freedom to pursue this problem. This work was funded by the Control Science Center of Excellence at the Air Force Research Laboratory (CSCE AFRL), the National Science Foundation (NSF) grant IIP-1161029, and the Center for Unmanned Aircraft Systems. This work was performed while Michael Otte was "in residence" at CSCE AFRL.

References

Abelson H, Allen D, Coore D, Hanson C, Homsy G, Knight Jr TF, Nagpal R, Rauch E, Sussman GJ and Weiss R (2000) Amorphous computing. *Communications of the ACM* 43(5): 74–82.

Agarwal A and Duchi JC (2011) Distributed delayed stochastic optimization. In: *Advances in Neural Information Processing Systems*. pp. 873–881.

Amkraut S, Girard M and G GK (1985) "motion studies for a work in progress entitled eurnythmy. *SIGGRAPH Video Review* 21.

Bain A (1894) *Mind and body: the theories of their relation*. International scientific series. D. Appleton & Company. URL `http://books.google.com/books?id=5bPwE5NeNNMC`.

Balch T and Arkin RC (1998) Behavior-based formation control for multirobot teams. *IEEE TRANSACTIONS ON ROBOTICS AND AUTOMATION* 14(6).

Barca JC and Sekercioglu A (2011) Generating formations with a template based multi-robot system. In: *Proceedings of Australasian Conference on Robotics and Automation*.

Beal J (2003) A robust amorphous hierarchy from persistent nodes .

Beal J (2005) Programming an amorphous computational medium. In: *Unconventional programming paradigms*. Springer, pp. 121–136.

Becker A, Habibi G, Werfel J, Rubenstein M and McLurkin J (2013) Massive uniform manipulation: Controlling large populations of simple robots with a common input signal. In: *Intelligent Robots and Systems (IROS), 2013 IEEE/RSJ International Conference on*. pp. 520–527. DOI:10.1109/IROS.2013.6696401.

Beni G (1988) The concept of cellular robotic system. In: *Intelligent Control, 1988. Proceedings., IEEE International Symposium on*. IEEE, pp. 57–62.

Berridge K and Seitzer J (2005) Extending the shortest-path swarm algorithm to cycle detection. In: *Circuits and Systems, 2005. 48th Midwest Symposium on*. pp. 931–934 Vol. 2.

Bonabeau E, Dorigo M and Theraulaz G (1999) *Swarm intelligence: from natural to artificial systems*. 1. Oxford university press.

Bottou L (1991) Stochastic gradient learning in neural networks. *Proceedings of Neuro-Nımes* 91(8).

Brogan DC and Hodgins JK (1997) Group behaviors for systems with significant dynamics. *Autonomous Robots* 4: 137–153.

Buckley SJ (1989) Fast motion planning for multiple moving robots. In: *Robotics and Automation, 1989. Proceedings., 1989 IEEE International Conference on*. IEEE, pp. 322–326.

Butera WJ (2002) Programming a paintable computer .

Çelikkanat H and Şahin E (2010) Steering self-organized robot flocks through externally guided individuals. *Neural Computing & Applications* 19(6): 849–865.

Chen J, Low KH and Tan CKY (2013) Gaussian process-based decentralized data fusion and active sensing for mobility-on-demand system. *arXiv preprint arXiv:1306.1491* .

Chen J, Low KH, Tan CKY, Oran A, Jaillet P, Dolan JM and Sukhatme GS (2012a) Decentralized data fusion and active sensing with mobile sensors for modeling and predicting spatiotemporal traffic phenomena. *arXiv preprint arXiv:1206.6230* .

Chen J, Low KH, Yao Y and Jaillet P (2015) Gaussian process decentralized data fusion and active sensing for

spatiotemporal traffic modeling and prediction in mobility-on-demand systems. *IEEE Transactions on Automation Science and Engineering* 12(3): 901–921.

Chen X, Eversole A, Li G, Yu D and Seide F (2012b) Pipelined back-propagation for context-dependent deep neural networks. In: *Interspeech*. ISCA. URL http://research.microsoft.com/apps/pubs/default.aspx?id=173312.

Cireşan D, Meier U and Schmidhuber J (2012) Multi-column deep neural networks for image classification. In: *Proceedings of the 2012 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, CVPR '12. Washington, DC, USA: IEEE Computer Society. ISBN 978-1-4673-1226-4, pp. 3642–3649.

Cireşan D, Meierand U, Gambardella L and Schmidhuber J (2010) Deep, big, simple neural nets for handwritten digit recognition. *Neural Computation* 22(12): 3207–3220.

Clark CM, Rock SM and Latombe JC (2003) Motion planning for multiple mobile robots using dynamic networks. In: *Robotics and Automation, 2003. Proceedings. IEEE International Conference on*, volume 3. IEEE, pp. 4222–4227.

Dahl G, McAvinney A and Newhall T (2008) Parallelizing neural network training for cluster systems. In: *Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Networks*, PDCN '08. Anaheim, CA, USA: ACTA Press. ISBN 978-0-88986-714-7, pp. 220–225. URL http://dl.acm.org/citation.cfm?id=1722252.1722292.

De Nardi R and Holland O (2007) Ultraswarm: A further step towards a flock of miniature helicopters. In: *Swarm Robotics*. Springer, pp. 116–128.

De Nardi R, Holland O, Woods J and Clarck A (2006) Swarmav: A swarm of miniature aerial vehicles .

de Oca MAM, Ferrante E, Scheidler A, Pinciroli C, Birattari M and Dorigo M (2011) Majority-rule opinion dynamics with differential latency: a mechanism for self-organized collective decision-making. *Swarm Intelligence* 5(3-4): 305–327.

Dean J, Corrado G, Monga R, Chen K, Devin M, Mao M, aurelio Ranzato M, Senior A, Tucker P, Yang K, Le QV and Ng AY (2012) Large scale distributed deep networks. In: Pereira F, Burges C, Bottou L and Weinberger K (eds.) *Advances in Neural Information Processing Systems 25*. Curran Associates, Inc., pp. 1223–1231.

Di Caro GA, Giusti A, Nagi J and Gambardella LM (2013) A simple and efficient approach for cooperative incremental learning in robot swarms. In: *Advanced Robotics (ICAR), 2013 16th International Conference on*. IEEE, pp. 1–8.

Dorigo M, Floreano D, Gambardella LM, Mondada F, Nolfi S, Baaboura T, Birattari M, Bonani M, Brambilla M, Brutschy A, Burnier D, Campo A, Christensen AL, Decugniere A, Caro GD, Ducatelle F, Ferrante E, Forster A, Gonzales JM, Guzzi J, Longchamp V, Magnenat S, Mathews N, de Oca MM, O'Grady R, Pinciroli C, Pini G, Retornaz P, Roberts J, Sperati V, Stirling T, Stranieri A, Stutzle T, Trianni V, Tuci E, Turgut AE and Vaussard F (2013) Swarmanoid: A novel concept for the study of heterogeneous robotic swarms. *IEEE Robotics Automation Magazine* 20(4): 60–71. DOI:10.1109/MRA.2013.2252996.

Émile Borel M (1909) Les probabilités dénombrables et leurs applications arithmétiques. *Rendiconti del Circolo Matematico di Palermo (1884-1940)* 27(1): 247–271.

Farber P and Asanovic K (1997) Parallel neural network training on multi-spert. In: *Algorithms and Architectures for Parallel Processing, 1997. ICAPP 97., 1997 3rd International Conference on*. pp. 659–666.

Farley BG and Clark WA (1954) Simulation of self-organizing systems by digital computer. institute of radio engineers. *Transactions on Information Theory* (4).

Ferrante E, Turgut AE, Huepe C, Stranieri A, Pinciroli C and Dorigo M (2012) Self-organized flocking with a mobile robot swarm: a novel motion control method. *Adaptive Behavior* : 1059712312462248.

Ferrante E, Turgut AE, Stranieri A, Pinciroli C, Birattari M and Dorigo M (2014) A self-adaptive communication strategy for flocking in stationary and non-stationary environments. *Natural Computing* 13(2): 225–245.

Gazi V and Passino KM (2003) Stability analysis of swarms. *IEEE TRANSACTIONS ON AUTOMATIC CONTROL* 48(4): 693.

Gilpin K, Knaian A and Rus D (2010) Robot pebbles: One centimeter modules for programmable matter through self-disassembly. In: *Robotics and Automation (ICRA), 2010 IEEE International Conference on*. IEEE, pp. 2485–2492.

Giusti A, Nagi J, Gambardella L and Di Caro G (2012) Cooperative sensing and recognition by a swarm of mobile robots. In: *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*. pp. 551–558.

Groß R, Bonani M, Mondada F and Dorigo M (2006) Autonomous self-assembly in swarm-bots. *Robotics, IEEE Transactions on* 22(6): 1115–1130.

Gruau F, Lhuillier Y, Reitz P and Temam O (2004) Blob computing. In: *Proceedings of the 1st conference on Computing frontiers*. ACM, pp. 125–139.

Guillot A (2008) Biologically inspired robots. In: *Springer Handbook of Robotics*. Springer, pp. 1395–1422.

Hamann H and Wörn H (2007) Embodied computation. *Parallel Processing Letters* 17(03): 287–298.

Heaton KB (2000) *Physical pixels*. PhD Thesis, Massachusetts Institute of Technology.

Heigold G, McDermott E, Vanhoucke V, Senior A and Bacchiani M (2014) Asynchronous stochastic optimization for sequence training of deep neural networks. In: *Acoustics, Speech and Signal Processing (ICASSP), 2014 IEEE International Conference on*. pp. 5587–5591.

Holland O, Woods J, De Nardi R and Clarck A (2005) Beyond swarm intelligence: the ultraswarm .

Hosseinmardi H, Mysore A, Farrow N, Correll N and Han R (2015) Distributed spatiotemporal gesture recognition in sensor arrays. *ACM Trans. Auton. Adapt. Syst.* 10(3): 17:1–17:19. DOI: 10.1145/2744203.

Kato S, Nishiyama S and Takeno J (1992) Coordinating mobile robots by applying traffic rules. In: *Proc. IEEE International Conference on Intelligent Robots and Systems*. pp. 1535–1541.

Komendera E, Reishus D and Correll N (2011) Assembly by intelligent scaffolding .

Lane C (1999) Stigmergy, self-organization, and sorting in collective robotics. *Artificial Life* 5(2): 173–202.

Langford J, , Smola AJ and Zinkevich M (2009) Slow learners are fast. In: Bengio Y, Schuurmans D, Lafferty J, Williams C and Culotta A (eds.) *Advances in Neural Information Processing Systems 22*. pp. 2331–2339.

LeCun YA, Bottou L, Orr GB and Müller KR (2012) Efficient backprop. In: *Neural networks: Tricks of the trade*. Springer Berlin Heidelberg, pp. 9–48.

Levine S, Pastor P, Krizhevsky A and Quillen D (2016) Learning hand-eye coordination for robotic grasping with large-scale data collection. In: *International Symposium on Experimental Robotics (ISER)*. Tokyo, Japan.

Low KH, Leow WK and Ang MH (2006) Autonomic mobile sensor network with self-coordinated task allocation and execution. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 36(3): 315–327.

Lumelsky VJ and Harinarayan K (1997) Decentralized motion planning for multiple mobile robots: The cocktail party model. In: *Robot colonies*. Springer, pp. 121–135.

Ma J, Zhao Q, Chaudhary V, Cheng J, Yang LT, Huang R and Jin Q (2006) Ubisafe computing: vision and challenges (i). In: *Autonomic and Trusted Computing*. Springer, pp. 386–397.

Matarić MJ (1992) Designing emergent behaviors: From local interactions to collective intelligence : 432 – 441.

Matsumoto A, Asama H, Ishida Y, Ozaki K and Endo I (1990) Communication in the autonomous and decentralized robot system actress. In: *Intelligent Robots and Systems '90. 'Towards a New Frontier of Applications', Proceedings. IROS '90. IEEE International Workshop on*. pp. 835–840 vol.2.

Nagi J, Di Caro GA, Giusti A, Nagi F and Gambardella LM (2012a) Convolutional neural support vector machines: Hybrid visual pattern classifiers for multi-robot systems. In: *Machine Learning and Applications (ICMLA), 2012 11th International Conference on*, volume 1. IEEE, pp. 27–32.

Nagi J, Ngo H, Giusti A, Gambardella L, Schmidhuber J and Di Caro G (2012b) Incremental learning using partial feedback for gesture-based human-swarm interaction. In: *RO-MAN, 2012 IEEE*. pp. 898–905.

Noel C and Osindero S (2014) Dogwild! – distributed hogwild for cpu & gpu. In: *Neural Information Processing Systems Conference: Distributed Machine Learning and Matrix Computations*.

OrHai M and Teuscher C (2011) Spatial sorting algorithms for parallel computing in networks. In: *Self-Adaptive and Self-Organizing Systems Workshops (SASOW), 2011 Fifth IEEE Conference on*. IEEE, pp. 73–78.

Otte M (2016a) Collective cognition & sensing in robotic swarms via an emergent group mind. In: *International Symposium on Experimental Robotics (ISER)*. Tokyo, Japan.

Otte M (2016b) Videos of experiments. `https://www.youtube.com/playlist?list=PLZ7pj8IjxuVUZS8UKLKceN6hqkMZ5rw35`.

Otte M, Bialkowski J and Frazzoli E (2014) Any-com collision checking: Sharing certificates in decentralized multi-robot teams. In: *IEEE International Conference on Robotics and Automation (ICRA)*. Hong Kong, China.

Otte M and Correll N (2010a) Any-com multi-robot path-planning: Maximizing collaboration for variable bandwidth. In: *Proc. 10th International Symposium on Distributed Autonomous Robotics Systems*.

Otte M and Correll N (2010b) Any-com multi-robot path-planning with dynamic teams: Multi-robot coordination under communication constraints. In: *Proc. 12th International Symposium on Experimental Robotics*.

Paine T, Jin H, Yang J, Lin Z and Huang TS (2013) GPU asynchronous stochastic gradient descent to speed up neural network training. *CoRR* abs/1312.6186. URL `http://arxiv.org/abs/1312.6186`.

Petrowski A, Dreyfus G and Girault C (1993) Performance analysis of a pipelined backpropagation parallel algorithm. *Neural Networks, IEEE Transactions on* 4(6): 970–981.

Pini G, Brutschy A, Frison M, Roli A, Dorigo M and Birattari M (2011) Task partitioning in swarms of robots: An adaptive method for strategy selection. *Swarm Intelligence* 5(3-4): 283–304.

Profita H, Farrow N and Correll N (2015) Flutter: An exploration of an assistive garment using distributed sensing, computation and actuation. In: *Proc. of the ACM Conference on Tangible Embodied Interaction*. Stanford, CA. To appear.

Recht B, Re C, Wright S and Niu F (2011) Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In: Shawe-taylor J, Zemel R, Bartlett P, Pereira F and Weinberger K (eds.) *Advances in Neural Information Processing Systems 24*. pp. 693–701.

Reynolds CW (1987) Flocks, herds, and schools: A distributed behavioral model. *Computer Graphics* 21(4).

Rogers R and Skillicorn D (1998) Using the bsp cost model to optimise parallel neural network training. In: Rolim J (ed.) *Parallel and Distributed Processing*, *Lecture Notes in Computer Science*, volume 1388. Springer Berlin Heidelberg. ISBN 978-3-540-64359-3, pp. 297–305. DOI:10.1007/3-540-64359-1_700.

Rubenstein M, Ahler C and Nagpal R (2012) Kilobot: A low cost scalable robot system for collective behaviors. In: *Robotics and Automation (ICRA), 2012 IEEE International Conference on*. pp. 3293–3298.

Rubenstein M, Cornejo A and Nagpal R (2014) Programmable self-assembly in a thousand-robot swarm. *Science* 345.

Rumelhart DE, Hinton GE and Williams RJ (1986) Learning representations by back-propagating errors. *Nature* 323: 533 – 536.

Rus D and Vona M (2001) Crystalline robots: Self-reconfiguration with compressible unit modules. *Autonomous Robots* 10(1): 107–124.

Schwager M, Rus D and Slotine JJ (2011) Unifying geometric, probabilistic, and potential field approaches to multi-robot deployment. *International Journal of Robotics Research* 30(3): 371–383.

Schwager M, Vitus MP, Powers S, Rus D and Tomlin CJ (2015) Robust adaptive coverage control for robotic sensor networks. *IEEE Transactions on Control of Network Systems* .

Schwartz JT and Sharir M (1983) On the piano movers problem. ii. general techniques for computing topological properties of real algebraic manifolds. *Advances in applied Mathematics* 4(3): 298–351.

Seide F, Fu H, Droppo J, Li G and Yu D (2014) On parallelizability of stochastic gradient descent for speech dnns. In: *ICASSP*. IEEE SPS. URL `http://research.microsoft.com/apps/pubs/default.aspx?id=217322`.

Servat D and Drogoul A (2002) Combining amorphous computing and reactive agent-based systems: a paradigm for pervasive intelligence? In: *Proceedings of the first international joint conference on Autonomous agents and multiagent systems: part 1*. ACM, pp. 441–448.

Silva FM and Almeida LB (1990) Acceleration techniques for the backpropagation algorithm. In: *Neural Networks*. Springer, pp. 110–119.

Soltero DE, Schwager M and Rus D (2014) Decentralized path planning for coverage tasks using gradient descent adaptive control. *International Journal of Robotics Research* 33(3): 401–425.

Someya T, Sekitani T, Iba S, Kato Y, Kawaguchi H and Sakurai T (2004) A large-area, flexible pressure sensor matrix with organic field-effect transistors for artificial skin applications. *Proceedings of the National Academy of Sciences of the United States of America* 101(27): 9966–9970.

Srivastava N, Hinton G, Krizhevsky A, Sutskever I and Salakhutdinov R (2014) Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research* 15(1): 1929–1958.

Steels L (1990) Cooperation between distributed agents through self-organisation. In: *Intelligent Robots and Systems '90. 'Towards a New Frontier of Applications', Proceedings. IROS '90. IEEE International Workshop on*. pp. 8–14 supl.

Suri NR, Deodhare D and Nagabhushan P (2002) Parallel levenberg-marquardt-based neural network training on linux clusters-a case study .

Tanner HG, Jadbabaie A and Pappas GJ (2007) Flocking in fixed and switching networks. *Automatic Control, IEEE Transactions on* 52(5): 863–868.

Tolksdorf R (2000) Models of coordination. In: *Engineering Societies in the Agents World*. Springer, pp. 78–92.

Trianni V (2006) *On the Evolution of Self-Organising Behaviours in a Swarm of Autonomous Robots*. PhD Thesis.

Tsitsiklis J, Bertsekas D and Athans M (1986) Distributed asynchronous deterministic and stochastic gradient optimization algorithms. *Automatic Control, IEEE Transactions on* 31(9): 803–812.

Tumer K and Wolpert D (2004) A survey of collectives. In: *Collectives and the design of complex systems*. Springer, pp. 1–42.

Ueyama T and Fukuda T (1993) Self-organization of cellular robots using random walk with simple rules. In: *Robotics and Automation, 1993. Proceedings., 1993 IEEE International Conference on*. pp. 595–600 vol.3.

Utete SW, Barshan B and Ayrulu B (1999) Voting as validation in robot programming. *The International Journal of Robotics Research* 18(4): 401–413.

Vincent P, Larochelle H, Lajoie I, Bengio Y and Manzagol PA (2010) Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion. *The Journal of Machine Learning Research* 11(1): 3371–3408.

Werfel J, Petersen K and Nagpal R (2014) Designing collective behavior in a termite-inspired robot construction team. *Science* 343(6172): 754–758.

Wilson S, Pavlic TP, Kumar GP, Buffin A, Pratt SC and Berman S (2014) Design of ant-inspired stochastic control policies for collective transport by robotic swarms. *Swarm Intelligence* 8(4): 303–327.

Witbrock M and Zagha M (1990) An implementation of back-propagation learning on gf11, a large simd parallel computer. *Parallel Computing* 14: 329–346.

Yamaguchi H (1997) Adaptive formation control for distributed autonomous mobile robot groups. In: *Robotics and Automation, 1997. Proceedings., 1997 IEEE International Conference on*, volume 3. pp. 2300–2305 vol.3.

Yim M, Shen WM, Salemi B, Rus D, Moll M, Lipson H, Klavins E and Chirikjian GS (2007a) Modular self-reconfigurable robot systems [grand challenges of robotics]. *Robotics & Automation Magazine, IEEE* 14(1): 43–52.

Yim M, Shirmohammadi B, Sastra J, Park M, Dugan M and Taylor CJ (2007b) Towards robotic self-reassembly after explosion. *Departmental Papers (MEAM)* : 147.

Yoshida E, Arai T, Ota J and Miki T (1994) Effect of grouping in local communication system of multiple mobile robots. In: *Intelligent Robots and Systems '94. 'Advanced Robotic Systems and the Real World', IROS '94. Proceedings of the IEEE/RSJ/GI International Conference on*, volume 2. pp. 808–815 vol.2.

Zambonelli F, Gleizes MP, Mamei M and Tolksdorf R (2004) Spray computers: Frontiers of self-organization. In: *Autonomic Computing, International Conference on*. IEEE Computer Society, pp. 268–269.

Zhang S, Zhang C, You Z, Zheng R and Xu B (2013) Asynchronous stochastic gradient descent for dnn training. In: *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*. pp. 6660–6663.

Appendix A: Additional Algorithms

In Appendix A we provide lower level details about selected subroutines that were called within the algorithms presented in the main body of the paper. These are included to help practitioners and others that may wish to replicate our work. We break this section into four subsections. In the first we provide pseudocode for the send message thread and the receive message callback, both of these run concurrently to the main state machine in order to facilitate message-passing between robots.

The second subsection covers the lower level neural network implementation details used in our experiments. These are provided to document our particular implementation; it is likely that other implementations will also work. The third subsection provides pseudocode for the various learning rate tuning methods that were used in our experiments. As with the previous algorithmic details, these are provided to document our approach and to assist practitioners that may wish to duplicate our work.

The fourth subsection presents pseudocode for the low-level message packing and unpacking that we used with the Kilobots. Because Kilobots messages are limited to a maximum of 10 bytes, we thought it fair to give practitioners some idea of how we packed all necessary data for training and using the group mind neural network (indeed, we actually break the messages for a single training iteration into multiple packets containing independent information, in an effort to minimize the negative effects of dropped message). In the fourth and final subsection we provide the high level behaviors and actions that are used in the small robot experiments without movement.

Algorithm 7: group mind Send Message Thread

```
1  loop
2      if state ∈ {Train, Observe} then
3          neural_data ← get_neural_data()
4          message ← {state, local_id, neural_data}
5      else if state = Act then
6          act_data ← get_act_data()
7          message ← {state, local_id, h, act_data}
8      else
9          populate message as required for calibration,
           initialization, etc.
10     broadcast(message)
```

Algorithm 8: group mind Receive Message Callback

```
1  if state ∈ {Train, Observe} then
2      sender_state ← message
3      if sender_state ∈ {Train, Observe} then
4          {sender_local_id, neural_data} ← message{2 : 3}
5          update_group_mind(sender_local_id, neural_data)
6      else if sender_state = Act then
7          state ← Act
8          h ← message{3}
9          B̂ ← f̂(h)
10         B ∈ B̂          // pick swarm behavior in B̂
11         α_n ∈ B        // α_n is this robot's action in B
12         if α_n = train then
13             run state Train
14         else
15             run state Act with α_n
16 else if state = Act then
17     if sender_state = Act then
18         α_n ← modify_action(α_n, message)
19 else
20     use message for calibration, initialization, etc.
```

High Level Send and Receive

Algorithm 7 depicts the message broadcast thread that runs concurrently to the main state machine (in an independent thread) and is responsible for broadcasting data from robot n to its neighbors.

Function get_neural_data() retrieves the neural network data that resides on this robot's portion of the group mind (line 3). For each training example as well as the real-time environmental sensor input, this includes both the forward neural signals and backpropagation messages (including training iteration number and, for each backpropagation message, the destination ID). Neural data is broadcast, along with this robot's state and ID (line 4). In practice, due to the Kilobots' small message payload size (10 bytes), we must divide each batch of neural network data across multiple messages (not shown). If there is movement behavior such that state Act is used, then the robot sends an identifier for this state, its own local ID, and the swarm behavior class h output of the neural network vs. real-time environmental data (lines 5-6). To save space we omit the other message-passing details necessary to run the standard distributed algorithms that we employ as subroutines during the start-up phase and the various actions used in state Act (represented by lines 8-9).

The receive message callback function appears in Algorithm 8. Normal training data is received on lines 2-5. If a neighbor has decided to act then this robot will join it (lines 6-15); making sure to perform its own prescribed behavior *behaviour* relevant to the overall swarm behavior h (line 15). The function modify_action(α_n, *message*) is used to modify the specific output behavior of this robot during the Act phase, as a function of interaction with neighboring robots (lines 16-18). This enables more complex swarm behaviors to emerge out of the interactions between robots. For example, the smiley faces in our experiments are created as randomly searching robots stop moving in the vicinity of attracting robots. Lines 19-20 represent other message processing that is used for the distributed subroutines within the start-up phase.

Standard Neural Network Implementation Details

Algorithm 9 describes the subroutine that initializes the slice of the neural network residing on this robot. The global robot index n is assumed to be unique within each neighborhood as discussed in the High Level Algorithm Section. Henceforth we shall refer to this robot as robot n.

Algorithm 9: init_neural_network()

```
1  n ← local_id
2  τ ← 0
3  for m ∈ N do
4      for ℓ ← 0, ..., L do
5          for h ← 0, ..., H do
6              s_ℓ^{m,h} ← 0
7      for ℓ ← 0, ..., L − 1 do
8          w_ℓ^{mn} ← random_weight()
9          w_ℓ^{nm} ← 1
10         for h ← 1, ..., H do
11             forward τ_ℓ^{m,h} ← 0
12     for ℓ ← 1, ..., L do
13         for h ← 1, ..., H do
14             u_ℓ^{m,h} ← 0
15     for ℓ ← 1, ..., L − 1 do
16         for h ← 1, ..., H do
17             backward τ_ℓ^{m,h} ← 0
18 for ℓ ← 0, ..., L − 1 do
19     ξ_ℓ ← NaN              // (batch version only)
20     for h ← 1, ..., H do
21         ξ_ℓ^h ← NaN        // (stochastic version only)
```

The iteration counters for robot n and those that it maintains for its neighbors are initialized to 0 (lines 2,

10-11, 16-18). The signal values are initialized to 0 for all neurons on all robots in the local neighborhood, and with respect to all training examples $(\mathbf{F}_i, h) \in \mathscr{T}$ as well as the real-time environmental data $h = 0$ (line 6). Edge weights are initialized to a small random number (line 8), and edge weights maintained by neighbors are set to 1 (line 9). The latter are reset to their actual values when messages from those neighbors are received; however, the two are equivalent for robot n (line 15). ξ, which holds squared error calculated during the previous training iteration, is/are initialized to NaN (lines 18-21); however, the structure of ξ depends on the variant of the backpropagation used. Batch backpropagation maintains one value per neuron (line 19), while stochastic backpropagation maintains one value per neuron per training example (lines 21).

We now provide details for the remaining nontrivial low level functions that are used in our experiments. This includes our implementation of the activation function, the update based on its derivative, learning rate update variants, group mind utilization, training error calculation, and message packing/unpacking functions.

Algorithm 10: activation(v)

1 **return** $1.7159 \tanh(2v/3)$

The activation function (Algorithm 10) maps total weighted neural input signal v to neural output. We use the hyperbolic activation function recommended by LeCun et al. (2012), which has been shown to work well in practice, is easy to calculate, and has a derivative that is also easy to calculate.

Algorithm 11: calc_update_parameter(A, B)

1 **return** $1.14393(1 - A^2)B$

Algorithm 11 provides the derivative of the activation function, evaluated at v, multiplied by B. Note that activation(v) is assumed to have been previously calculated. The derivative of the activation function in Algorithm 11 simplifies to this form because the derivative of $\tanh(v)$ is $1 - \tanh^2(v)$.

Neural Network Tuning Method Implementation Details

Algorithm 12: tune_learning_rate(τ) (decreasing rate variant)

1 **return** c_γ/τ

Algorithm 12 shows the "decreasing rate" version of tune_learning_rate(), the function used to update the relative size of gradient descent updates vs. training iteration number. This version (vs. the alternative methods in Algorithms 13 and 14) leads to the most stable, but also the slowest convergence. Learning rates for all neurons start at a user-defined constant tuning parameter c_γ and decrease proportionally to the inverse of training iteration number. It can be used with both the batch and stochastic training methods.

Algorithm 13: tune_learning_rate(γ, ϵ, ℓ) (batch heuristic Hessian variant)

1 $\zeta \leftarrow \sum_{(\mathbf{F}_i,h)\in\mathscr{T}} (\epsilon_\ell^h)^2$
2 $\Delta \leftarrow \zeta - \xi_\ell$
3 **if** $\Delta = \text{NaN}$ **then**
4 $\quad \gamma_\ell^{\text{new}} \leftarrow c_\gamma$
5 **else if** $\Delta > 0$ **then**
6 $\quad \gamma_\ell^{\text{new}} \leftarrow c_{up}\gamma_\ell$
7 **else**
8 $\quad \gamma_\ell^{\text{new}} \leftarrow c_{down}\gamma_\ell$
9 $\gamma_\ell^{\text{new}} \leftarrow \min(c_{max}, \min(c_{min}, \gamma_\ell^{\text{new}}))$
10 **return** γ_ℓ^{new}

Using cost gradient Hessian information to update the learning rate can often enable faster convergence but comes at the price of requiring additional tuning parameters. We use a simple heuristic method by Silva and Almeida (1990). The batch version of this method appears in Algorithm 13. The basic idea is to track if summed squared error over all examples (line 1) is increasing or decreasing (lines 1-2) by subtracting the previous squared error ξ_ℓ from the current squared error ζ. If the error is decreasing, then we increase the learning rate by a factor of c_{up}, while if error is decreasing, then we decrease the learning rate by a factor of c_{down} (lines 6 and 8, respectively). Bad numerical values cause the learning rate to be re-initialized to c_γ. We also bound the maximum and minimum values that the learning rate is allowed to take (line 9). In our experiments we set $c_{max} = c_\gamma * 10$ and $c_{min} = c_\gamma/100$ in order to reduce the number of tuning parameters to three (c_γ, c_{min}, and c_{max}).

Algorithm 14: tune_learning_rate$(\gamma, \epsilon, \ell, h)$ (stochastic heuristic Hessian version)

1 $\Delta \leftarrow \sum_{(\mathbf{F}_i,h)\in\mathscr{T}} (\epsilon_\ell^h)^2 - \sum_{(\mathbf{F}_i,h)\in\mathscr{T}} \xi_\ell^h$
2 $\xi_\ell^h \leftarrow (\epsilon_\ell^h)^2$
3 **if** $\Delta = \text{NaN}$ **then**
4 $\quad \gamma_\ell^{\text{new}} \leftarrow c_\gamma$
5 **else if** $\Delta > 0$ **then**
6 $\quad \gamma_\ell^{\text{new}} \leftarrow c_{up}\gamma_\ell$
7 **else**
8 $\quad \gamma_\ell^{\text{new}} \leftarrow c_{down}\gamma_\ell$
9 $\gamma_\ell^{\text{new}} \leftarrow \min(c_{max}, \min(c_{min}, \gamma_\ell^{\text{new}}))$
10 **return** $(1/H)\gamma_\ell^{\text{new}} + (1 - 1/H)\gamma_\ell$

Algorithm 14 shows the heuristic Hessian learning rate method that is designed for use with stochastic backpropagation. It is very similar to the batch version (Algorithm 13), except that it is designed to work given data about a single training example h and layer ℓ instead of accounting for all examples and layers at the same time. This difference means that we must store the previous squared error for each neuron and example (line 2). A running average based on H is used to combine updates

from different training examples over multiple calls to the subroutine.

Algorithm 15: training_error()

1 **return** $\frac{1}{LH}\sum_{(\mathbf{F}_i,h)\in\mathscr{T}}\sum_{\ell=1}^{L}\epsilon_\ell^h$

Algorithm 15 shows the calculation used to determine the current training error. In practice a robot stops training when this value falls below the user-defined TARGET_ERROR.

Low Level Data Packaging used with Kilobots

Algorithm 16: get_neural_data()

1 $payload\{1\} \leftarrow \tau$
2 $c_{\text{total}} \leftarrow L + LH + (L-1)H|\mathcal{N}|$
3 $payload\{2\} \leftarrow (\eta+1) \mod c_{\text{total}}$
4 **for** $k \leftarrow 3,\ldots,payload\_size$ **do**
5 $\quad\eta \leftarrow (\eta+1) \mod c_{\text{total}}$
6 $\quad$**if** $\eta < L$ **then**
7 $\quad\quad h \leftarrow 0$
8 $\quad\quad \ell \leftarrow \eta$ // $\ell \in [0, L-1]$
9 $\quad\quad payload\{k\} \leftarrow s_\ell^{n,h}$
10 $\quad$**else if** $\eta < L + LH$ **then**
11 $\quad\quad h \leftarrow 1 + (\eta - L) \mod H$ // $h \in [1, H]$
12 $\quad\quad \ell \leftarrow \lfloor(\eta-L)/H\rfloor$ // $\ell \in [0, L-1]$
13 $\quad\quad payload\{k\} \leftarrow s_\ell^{n,h}$
14 $\quad$**else**
15 $\quad\quad c \leftarrow \lfloor(\eta - L - LH)/|\mathcal{N}|\rfloor$
16 $\quad\quad m \leftarrow (\eta - L - LH) \mod |\mathcal{N}|$ // $m \in [0, |\mathcal{N}|-1]$
17 $\quad\quad h \leftarrow 1 + c \mod H$ // $h \in [1, H]$
18 $\quad\quad \ell \leftarrow \lfloor c/H\rfloor$ // $\ell \in [1, L-1]$
19 $\quad\quad payload\{k\} \leftarrow \{m\,,\,h\,,\,\ell\,,\,u_\ell^{m,h}\,w_\ell^{nm}\}$
20 **return** $payload$

Due to practical hardware constraints of the Kilobot platform, message payload size is limited to 9 bytes and a message ID provides 1 additional byte of information capacity. As a result, we must break signal and training data into pieces and send multiple messages in order to get data from robot n to its neighbors and vice versa. The pseudocode presented in Algorithms 16 and 17 outline the basic technique that we use to respectively pack and unpack neural signal and training data to/from a message $payload$. In order to keep our presentation at a high level, we use the notation $payload\{k\}$ to denote the k-th piece of data contained in the $payload$, ignoring the number of bytes required.

In general, robot n cycles between broadcasting real-time signal data, forward messages for training, and backward messages for training (Algorithm 16, lines 6-9, lines 10-13, and lines 14-19, respectively. There are $L + LH + (L-1)H|\mathcal{N}|$ different pieces of information that need to be sent. L real-time signal messages and LH forward messages need to be sent to all neighbors of robot n. Although each of the $(L-1)H|\mathcal{N}|$ backward messages is destined for (only) a single neighbor, we are constrained such that broadcasting is the only way to transmit data. Therefore, the destination robot is included along with backward messaging data (line 19). We use a single index η to cycle through all possible messages, sending each in turn (lines 5,6,10,14). We have included comments in the algorithm to show how incriminating η cycles through all combinations of h, n, and ℓ.

Algorithm 17: update_group_mind(m, $payload$)

1 $\tau_{\text{sender}} \leftarrow payload\{1\}$
2 $c_{\text{total}} \leftarrow L + LH + (L-1)H|\mathcal{N}|$
3 $\eta \leftarrow payload\{2\}$
4 **for** $k \leftarrow 3 : payload\_size$ **do**
5 $\quad$**if** $\eta < L$ **then**
6 $\quad\quad h \leftarrow 0$
7 $\quad\quad \ell \leftarrow \eta$ // $\ell \in [0, L-1]$
8 $\quad\quad s_\ell^{m,h} \leftarrow payload\{k\}$
9 $\quad$**else if** $\eta < L + LH$ **then**
10 $\quad\quad h \leftarrow 1 + (\eta - L) \mod H$ // $h \in [1, H]$
11 $\quad\quad \ell \leftarrow \lfloor(\eta-L)/H\rfloor$ // $\ell \in [0, L-1]$
12 $\quad\quad s_\ell^{m,h} \leftarrow payload\{k\}$
13 $\quad\quad {}^{\text{forward}}\tau_\ell^{m,h} \leftarrow \tau_{\text{sender}}$
14 $\quad$**else**
15 $\quad\quad intended\_destination \leftarrow payload\{k\}\{1\}$
16 $\quad\quad$**if** $n = intended\_destination$ **then**
17 $\quad\quad\quad h \leftarrow payload\{k\}\{2\}$
18 $\quad\quad\quad \ell \leftarrow payload\{k\}\{3\}$
19 $\quad\quad\quad u_\ell^{m,h}\,w_\ell^{nm} \leftarrow payload\{k\}\{4\}$
20 $\quad\quad\quad {}^{\text{backward}}\tau_\ell^{m,h} \leftarrow \tau_{\text{sender}}$
21 $\quad\eta \leftarrow (\eta+1) \mod c_{\text{total}}$

Algorithm 17 is the receiving counterpart to Algorithm 16 and uses the same form of index compression. Data is extracted from the message $payload$ and saved locally to facilitate model use and training. Backward messages are only saved when they are relevant to the receiving robot (lines 16-20). If a message contains forward or backward data, then we store the sender's current iteration number τ_{sender} in ${}^{\text{forward}}\tau_\ell^{m,h}$ or ${}^{\text{backward}}\tau_\ell^{m,h}$, respectively (lines 1, 13, 20).

The pseudocode in Algorithms 17 and 16 ignores a few tricks that we use to further compress the data that is sent. These tricks amount to low-level bit toggling that allows the compression of more than one piece of information in a byte (e.g., two different numbers between 0 and 15 can be stored in the upper and lower bits of a byte). Finally, in our presentation we have left $u_\ell^{m,h}\,w_\ell^{nm}$ as the product of two separate quantities to keep the notation in Algorithms 9-11 consistent with standard implementation of backpropagation; however, in practice this multiplication is performed on the sending robot and the product sent (the receiving robot always uses the product $u_\ell^{m,h}\,w_\ell^{nm}$, and not $u_\ell^{m,h}$ or w_ℓ^{nm} individually).

Finally, we implement a custom floating point data structure that fits in a signal byte and represents numbers of the form $\pm 1.25^z$ and 0, for integer $z \in [-64, 64]$; i.e., numbers in the set $[-1.25^{64}, -1.25^{-64}] \cup \{0\} \cup [1.25^{-64} 1.25^{64}]$.

This means that numbers in the ranges $[-1.25^{64}, -1.25^{-64}]$ and $[1.25^{-64}1.25^{64}]$ are represented with at most 25% relative error. This single byte float is used only to store and send the products $u_\ell^{m,h}\, w_\ell^{nm}$, with other floating point values such as signals sent using standard 32 bit (4 byte) data structures. This compression was deemed necessary for backward messages due to the fact that they are bandwidth expensive[††], and because we have found the quantity $u_\ell^{m,h}\, w_\ell^{nm}$ to be tolerant to small changes, in practice. We attribute the latter robustness to the fact that the most important piece of information, with respect to the intended use of $u_\ell^{m,h}\, w_\ell^{nm}$, is its sign (positive vs. negative). Rounding occurs toward 0 and thus using this compression amounts to learning at slightly a slower rate.

Appendix B: Swarm behaviors and Actions used in experiments with stationary robots (small swarms)

Here we outline the different swarm behaviors and robot actions used in the repeated experiments with small swarms. The overall state machine used in state Act for this experiment is shown if Figure 18.

Different behavior sets are defined by different LED light patterns, as follows:

- $\mathcal{B}_0$ = collectively display red.
- $\mathcal{B}_1$ = collectively display blue.
- $\mathcal{B}_2$ = collectively display teal.
- $\mathcal{B}_3$ = collectively display yellow.

The low level robot actions all involve robots outputting different LED colors (or no color):

- z_1 = Red_LED.
- z_2 = Blue_LED.
- z_3 = Teal_LED.
- z_4 = Yellow_LED.

All valid behaviors are of the form $\mathbf{B}_k \in \mathcal{B}_k$ are of the form $\{\alpha_1 = z_k, \ldots, \alpha_n = z_k\} = \mathbf{B}_k$.

Appendix C: Additional Related work from other technical fields

The group mind overlaps with a number of other technical ideas that have previously appeared in the literature. This section is dedicated to describing these other ideas and their relationship to the group mind. Much of this discussion is tabulated in Table 2.

Surveys of related concepts that cut across disciplines include Tumer and Wolpert (2004) and Tolksdorf (2000). Tumer and Wolpert (2004) study the idea of *collectives* defined as "such systems, where each agent aims to optimize its own performance, but where there is a well-defined set of system-level performance criteria." Tolksdorf (2000) investigates different forms of coordination.

Getting multiple robots to work together—or even near each other—necessarily involves some form of coordination. The Group Mind can be used as a tool for coordination. Many coordination strategies have previously been studied, including: centralization (Schwartz and Sharir 1983), voting (Utete et al. 1999), prioritization (Buckley 1989), decentralization (Lumelsky and Harinarayan 1997), traffic rules (Kato et al. 1992), dynamic teams (Clark et al. 2003), and distributed control (Schwager et al. 2011; Soltero et al. 2014; Schwager et al. 2015). Centralization and prioritization seem capable of solving "trained at runtime heterogeneous response problems" that involve tens of robots or less; however, these classes of methods do not scale to large number of robots. Voting and traffic rules are complementary to the group mind and to each other; nothing prevents a swarm from using voting, traffic rules, and the group mind ideas simultaneously. The group mind may be considered a dynamic team and can be used to facilitate distributed control.

Biologically inspired robotics (Guillot 2008) is an area in which the design of robotic hardware and software is motivated by solutions and technologies found in nature. *Artificial swarms* is a sub-field of biologically inspired robotics that focuses on systems with large numbers of simple robots that can communicate with each other and/or interact with their environment. Methods inspired by insect swarm behavior have been studied as far back as Amkraut et al. (1985); Reynolds (1987). The group mind is both biologically inspired and designed to run on an artificial swarm.

For a comprehensive survey of swarm concepts we suggest Trianni (2006). Sub-genres of swarm research include the study of emergent behavior (Matarić 1992), collective intelligence (Bonabeau et al. 1999), task partitioning (Pini et al. 2011), task allocation (Matsumoto et al. 1990), stability and control (Tanner et al. 2007; Becker et al. 2013; Brogan and Hodgins 1997; Gazi and Passino 2003), and heterogeneity (Dorigo et al. 2013). The group mind can be used as a tool to enable emergent behavior, collective intelligence, task partitioning, task allocation, and behavior that is heterogeneous. The group mind can be used to assign robot actions related to the stability and control of a swarm.

Emergent behavior is concerned with complex swarm behavior resulting from the interaction between many robots. A selection of topics includes: flocking (Reynolds 1987; Ferrante et al. 2012), self organization (Beni 1988; Ueyama and Fukuda 1993; Yoshida et al. 1994), adaptive behavior (Yamaguchi 1997), and self assembly (Rubenstein et al. 2014). The group mind uses self-organization and can potentially be used as a tool to enable flocking, adaptive behavior, self assembly, assembly, and other emergent swarm behaviors.

Other topics related to swarm behavior that have previously been studied include: assembly (Werfel et al. 2014), collective transport (Wilson et al. 2014), foraging (Steels 1990), box-pushing (Becker et al. 2013), sorting (Lane 1999; OrHai and Teuscher 2011), convergence (de Oca et al. 2011), path finding (Berridge and Seitzer 2005), and formation (Balch and Arkin 1998; Barca and Sekercioglu 2011). These are some of the potential swarm behaviors that might be selected by the group mind in response to environmental data.

[††]Each backward message is destined for only a single recipient; thus, the local ID of the recipient must be included, and the act of broadcasting a backward message uses bandwidth shared by many robots that will not benefit from receiving that message.

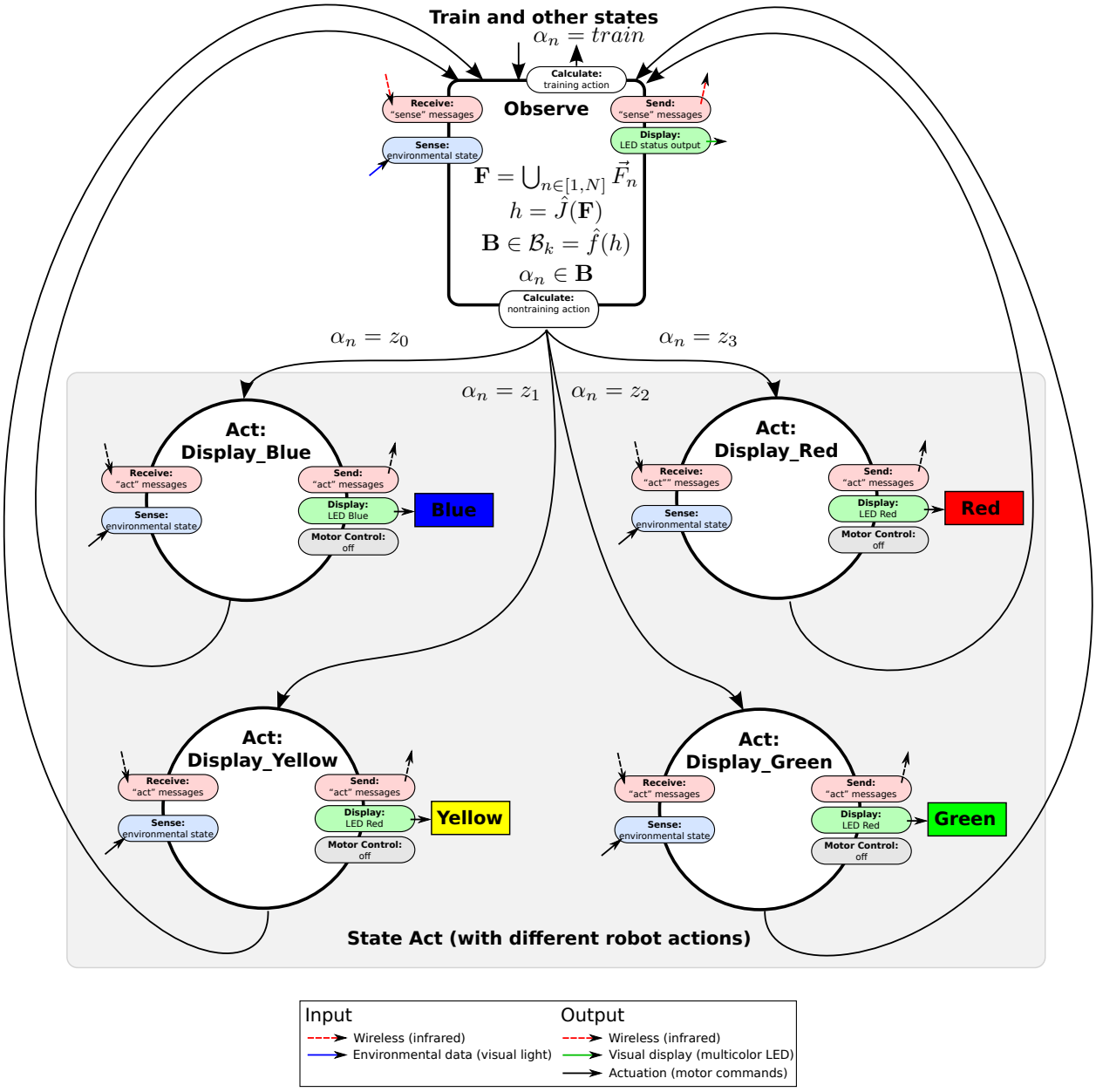


Figure 18. The various actions used in the experiment with a small swarm (4 robots) that does not involve movement (and so the group mind remains intact as long as the swarm is deployed). Depicted is the portion of the high-level state machine that connects robot actions with the overall state-machine described in Figure 7. Each action requires the robot to display a particular color (red, blue, green, or yellow LED). Each state automatically transitions back to the state Observe so that the swarm can update its collective behavior as the group mind detects different environmental feature patterns (e.g., as they change in the environment as the swarm is deployed). The environmental feature patterns used in the experiment are shown in Figure 11.

Other tools that have been used to facilitate collective swarm behavior include: local communication (Ferrante et al. 2014; Yoshida et al. 1994), local sensing (Ferrante et al. 2012), global signaling (Becker et al. 2013), stigmergy (Lane 1999), field-following, predefined schemata (Balch and Arkin 1998), and partial control (Çelikkanat and Şahin 2010). The group mind uses local communication, local (and global) sensing, and global signaling. A swarm that hosts a group mind can potentially use any of these tools as part of an output swarm behavior.

A *self reconfigurable robot* is a single robot composed of multiple smaller robots that can alter their arrangement, e.g., to provide different forms of locomotion (Yim et al. 2007a; Rus and Vona 2001). *Robotic self-assembly* (Groß et al. 2006) happens when either a self reconfigurable robot or a meta-robot spontaneously constructs itself via the unification of smaller robots, and without the help of an outside actor (Rubenstein et al. 2014; Yim et al. 2007b). Self-assembling reconfigurable robots solve the closely related problem of creating a *physical* entity out of a set of robots. The difference between the group mind and robotic self-assembly is analogous to the difference between mind and body. There is nothing preventing a self assembling (or reconfigurable) robot from using a group mind, but this has not yet been done.

Smart materials are able to reason about and/or respond to external stimuli. While most common smart materials react to stimuli in a purely analog fashion (e.g., ferrofluids),

	physical structure/organization					Computation						behavior & action	
	product	permanent	temporary	changing	emergent	is an output product	has permanent structure	has temporary structure	program unit	program meta entity	emergent network	emergent	user code
group mind						X	[X]	X	X	X	X		X
group mind + swarm	X	X	X	X	X	X	[X]	X	X	X	X	X	X
self-reconfigurable robot	X			X			Y		Y	Y			
Robotic self-assembly	X	X		[X]	X		Y		Y	[Y]	[Y]		
Amorphous computing	X	X		X	random	X	X		X		X	X	X
Embodied computation	Y	Y	Y	Y	Y	X	[X]	X	X		X	X	
Smart materials (structure)	X		X	X			Y		X				X
Smart materials (actuation)		X		X			Y		X	no meta exists			X
Programmable Matter		X				X	X	X	X	no meta exists		X	X

X Defining characteristics of a method.
[X] Defining characteristics that are occasionally absent in selected works.
Y Common characteristics of a method that are not required.
[Y] Characteristics that occasionally show up in a method.

Table 2. Summary comparison between the group mind and other work from robotics and other related fields.

a growing number also contain digital processing elements that can be used to perform sensing and computation. The latter variety are called *programmable matter* and share similarities with both the group mind and self-reconfigurable robotics (e.g., intelligent scaffolding (Komendera et al. 2011), distributed sensing, etc.). A distinguishing feature of smart materials is that their constituent elements remain mostly assembled once the material is fabricated. A sampling of programmable matter research that is related to the group mind from a computational point of view includes: smart walls (Hosseinmardi et al. 2015), smart sand (Gilpin et al. 2010), smart paint (Butera 2002), sensing skins (Someya et al. 2004), and smart clothes (Profita et al. 2015). Smart materials are another potential host for a group mind.

Amorphous computing "considers the problem of controlling millions of spatially distributed unreliable devices which communicate only with nearby neighbors" (Beal 2005); our group mind is thus a form of amorphous computing. In general the term is used to indicate distributed computation among a network of nodes in which organization emerges organically from local communication and random, haphazard, and/or ubiquitous node placement, and with the assumption that algorithms should scale to millions or billions of nodes. Examples of amorphous computing include: node specialization (Abelson et al. 2000), emergent hierarchy (Beal 2003), visual display/art (Heaton 2000), pervasive/ubiquitous computing (Servat and Drogoul 2002; Ma et al. 2006), spray computers (Zambonelli et al. 2004), and blob computers (Gruau et al. 2004). The major difference between the group mind and the vast majority of this previous work, is that the group mind involves creating a fully-functioning deliberative meta-computer out of a set of fully-functioning deliberative participant robots.

Embodied computation is defined as a multi-level computational model in which the robots at the micro-level are loosely coupled and have limited abilities, while the macro-level behavior emerges from self-organization and environmental interaction/computation (Hamann and Wörn 2007). The group mind swarm we investigate fits this definition (as do all artificial swarms and some types of programmable matter). The main philosophical distinction between embodied computation as defined by Hamann and Wörn (2007) and our group mind swarm is that, in the group mind, the meta-entity fits the (narrow) definition of a deliberative computational entity, e.g., a user can interact with or reprogram the group mind directly. Hamann and Wörn (2007) allows a broader definition of computation that is rooted in the state change (in the robots/environment) that emerges due to the local interaction of the micro-level devices. Our work differs from (Hamann and Wörn 2007) in that we use real robots and teach the group mind to recognize patterns in the environment; in contrast, Hamann and Wörn (2007) use simulation and perform experiments building Steiner trees and two-class density detection.

Appendix D: Additional related work on parallel neural networks

Neural Networks have been studied since the 1870s (Bain 1894), and artificial networks implemented on computers as early as the 1950s (Farley and Clark 1954). More recently, they have experienced at least two major resurgences. The first, in the 1980s after the discovery of the backpropagation algorithm (Rumelhart et al. 1986) enabled better training. The current "deep learning" revolution has arguably been enabled by faster computers and inexpensive distributed computing resources (Cireşan et al. 2010, 2012).

Three fundamentally different approaches have previously been studied for parallelizing neural network computation: (1) model parallelism, (2) data parallelism, (3) numerical parallelism. These ideas are complimentary, and many state of the art methods use multiple forms of parallelism in the

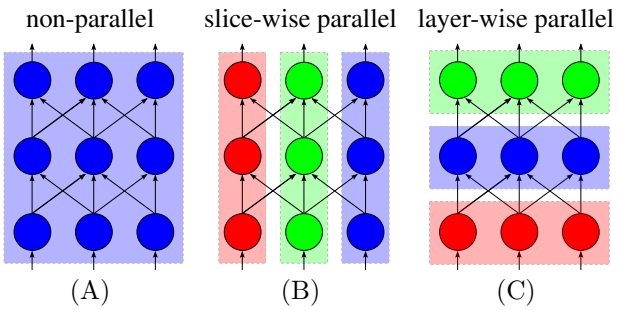


Figure 19. Common computational models in neural networks. Colors illustrate how computation is distributed across multiple CPUs. (A) All computation happens on a single CPU. (B) slice-wise parallelism gives each of N CPUs responsibility for $1/N$ nodes in each layer. (C) layer-wise parallelism gives each of N CPUs responsibility for $1/N$ of the layers.

same algorithm (Suri et al. 2002; Dean et al. 2012; Heigold et al. 2014); Figure 19 illustrates the differences between these types.

A neural network is essentially a graph $G = (V, E)$ of nodes V (neurons) and weighted edges E (signal connections), in which information flows through the edges and is modified by the nodes. The computation at each node depends only on the signals sent along its incoming edges. Model parallelism divides responsibility for the network among N CPUs by making each CPU responsible for $|V|/N$ nodes. Signals are stored in shared memory or sent as messages between CPUs.

Standard multi-layer perception neural networks are most often parallelized 'slice-wise' (see Figure 19-B), where each of N CPUs is responsible for $1/N$ of the neurons in *each* layer ℓ (Farber and Asanovic 1997; Rogers and Skillicorn 1998; Dean et al. 2012). Most methods reduce unnecessary overhead by attempting to minimize the number of messages sent between CPUs by assigning downstream nodes to the same CPU as much as possible (this obviously breaks if the connections between layers are dense).

A less common alternative (Petrowski et al. 1993; Chen et al. 2012b; Dean et al. 2012) is to parallelize the network 'layer-wise' in which each CPU is responsible for all neurons on L/N contiguous layers, where $1 \leq \ell \leq L$ and L is the depth of the network (see Figure 19-C).

Model parallelism without regard to the specific graph structure has also been investigated, wherein each node's computation is farmed out to the next available CPU (Rogers and Skillicorn 1998).

Network topology and computer architecture will largely dictate which form of model parallelism is used in practice. If the standard forward-pass and backpropagation training iteration is used, then the gradients used at layers ℓ will be $\lfloor (L-\ell)/N \rfloor$ iterations out of date when 'layer-wise' propagation is used. Theoretical results show that using delayed updates still converges well (Langford et al. 2009), in general, and experimental results demonstrate that this is not a problem in practice (Chen et al. 2012b).

Training any non-trivial neural network involves a set $\mathscr{T}$ of training examples where $|\mathscr{T}| > 1$. Data parallelism is achieved by duplicating the entire network on N CPUs, and then making each CPU responsible for training the model vs. $|\mathscr{T}|/N$ of the training examples (Witbrock and Zagha 1990; Farber and Asanovic 1997; Rogers and Skillicorn 1998; Dahl et al. 2008; Cireşan et al. 2012; Paine et al. 2013; Zhang et al. 2013; Noel and Osindero 2014). In practice, a parameter server is often used to accumulate and redistribute all updates to the model weights, but broadcast messages have also been used.

Rich theoretical results exist for the practical case that the most recent weights are asynchronously distributed to the CPUs (note that, when the backpropagation algorithm is used then this form of learning is a special case of asynchronous stochastic gradient descent). Tsitsiklis et al. (1986) provide convergence criteria for a variety of asynchronous stochastic computational models, in general, while Bottou (1991) provides the first convergence results of stochastic gradient descent in neural networks. Langford et al. (2009) perform stochastic gradient descent with outdated (by no more than τ iterations) updates, and show that this causes a model to train no more than τ times as slowly. Agarwal and Duchi (2011) show that asynchronous distributed gradient based optimization for stochastic problems scales asymptotically as $O(1/\sqrt{\eta\tau})$, where τ is iteration number and $\eta = |\mathscr{T}|/N$ is the number of training examples given to each CPU. This is similar to the known result for synchronous algorithms. Recht et al. (2011) shows that distributed stochastic gradient descent *without locking* will also converge if each iteration only modifies a small portion of the decision variable. This is an important result because it enables the elimination of a computational bottleneck. Noel and Osindero (2014) extend the work of Recht et al. (2011) to GPUs with asynchronous gradient streaming between nodes. Seide et al. (2014) give bounds on speedup one can expect for data parallelism (as well as model parallelism).

A final type of neural network parallelism involves representing the network and weights in matrix form, and then replacing the necessary linear algebra calls with standard parallel implementation of the same (Suri et al. 2002).

The group mind uses a variant of model slice-wise parallelism that is necessitated by the relatively high cost of passing messages between CPUs (robots). In particular, we pack all current signal messages for a robot's slice (across all training examples plus the real-time sensor data) into a single message[‡‡] that includes both forward and backward propagation data. As a result, the gradient data use by our method is $N - \ell$ iterations outdated at depth ℓ (similar to many model layer-wise parallel algorithms). Thus our method shares both similarities and differences with previous slice-wise (Petrowski et al. 1993; Chen et al. 2012b; Dean et al. 2012) and layer-wise (Farber and Asanovic 1997; Rogers and Skillicorn 1998; Dean et al. 2012) implementations.

Both (Srivastava et al. 2014) and (Vincent et al. 2010) intentionally drop messages at internal layers of the neural network to make the model more robust to noise. While our messages are dropped unintentionally due to packet collisions and environmental interference, this related work shows that moderate message loss (e.g., 50%) can actually

[‡‡] In our experiments, this single message is broken into multiple packets for transmission between Kilobots.

make the learning algorithm more robust. On a somewhat related note, Paine et al. (2013) synchronize weights only every 600 training iterations due to the fact that getting data into the GPU buffer is expensive; demonstrating that even moderate weight update delays do not prevent models from training in practice.