Towards Reinforcement Learning-based Aggregate Computing

Gianluca Aguzzi[1][0000−0002−1553−4561], Roberto Casadei[1][0000−0001−9149−949X], and Mirko Viroli[1][0000−0003−2702−5702]

Alma Mater Studiorum – Università di Bologna
{gianluca.aguzzi,roby.casadei,mirko.viroli}@unibo.it

Abstract. Recent trends in pervasive computing promote the vision of Collective Adaptive Systems (CASs): large-scale collections of relatively simple agents that act and coordinate with no central orchestrator to support distributed applications. Engineering global behaviour out of local activity and interaction, however, is a difficult task, typically addressed by try-and-error approaches in simulation environments. In the context of Aggregate Computing (AC), a prominent functional programming approach for CASs based on field-based coordination, this difficulty is reflected in the design of versatile algorithms preserving efficiency in a variety of environments. To deal with this complexity, in this work we propose to apply Machine Learning techniques to automatically devise local actions to improve over manually-defined AC algorithms specifications. Most specifically, we adopt a Reinforcement Learning-based approach to let a collective learn local policies to improve over the standard gradient algorithm—a cornerstone brick of several higher-level self-organisation algorithms. Our evaluation shows that the learned policies can speed up the self-stabilisation of the gradient to external perturbations.

Keywords: Collective Adaptive System · Aggregate Computing · Reinforcement Learning · Collective Intelligence.

1 Introduction

The pervasiveness of computing and networking fosters applications backed by large-scale cyber-physical collectives—cf. edge-fog-cloud infrastructures, robot swarms, and smart ecosystems. Combined with the *autonomic computing* vision [18], which promotes autonomy and self-* capabilities in engineered systems, there is an increasing trend towards Collective Adaptive Systems (CASs) and their engineering [31,9]. CASs are characterised by a multitude of agents that can produce globally coherent results (*emergents* [43]), and collective-level adaptivity to environment change via local decision-making and decentralised interaction. The *engineering of CASs* is an open research problem [31,19] of significance, tightly linked with the problems of "steering" self-organisation and "controlling" emergence to promote desired while avoiding undesired emergents [29]. In general, when dealing with CASs, there are two distinct problems: (i) given an

initial system state and local behavioural rules, predicting what global outcomes will be produced (*forward*, *prediction*, or *local-to-global problem*); and (ii) what local behavioural rules must be assigned to the system devices to achieve certain global outcomes (*inverse*, *control*, or *global-to-local problem*). These two problems provide corresponding perspectives for *designing* CASs. In particular, the latter perspective has promoted research on *spatial* and *macro-programming* [7,10] aiming at expressing programs in terms of the desired global outcome and leaving the underlying platform to deal with the global-to-local mapping.

In this work, we consider *Aggregate Computing (AC)* [8], a prominent *field-based coordination* approach [41] promoting macro-programming by capturing CAS behaviours as functions operating on *computational fields* [41], in a system model of neighbour-interacting devices operating in asynchronous sense-compute-interact rounds. A computational field is a macro-abstraction that maps a set of devices over time to computational values. AC is based on the *Field Calculus (FC)* [41], or variants thereof, that define constructs for manipulating and evolving fields. So, CAS behaviour can be expressed by a single *aggregate program* (global perspective) that also defines what processing and communication activities must be performed by each individual device (local perspective).

Besides the programming model and its implications, a significant portion of research on AC [41] has focussed on design and analysis of *coordination algorithms* expressed in FC for efficiently carrying out self-organising behaviours like, e.g., computing fields of minimum distances from sources (*gradients*) [30,24,4], electing leaders [27], or distributed summarisation [3]. However, devising self-organising coordination algorithms is not easy; especially difficult is identifying solutions that are efficient across environment assumptions, configurations, and perturbations. The difficulty lies in determining, for a current context, the local decisions of each device, in terms e.g. of processing steps and communication acts, producing output fields that quickly converge to the correct solution.

In this work, we start what we consider a key future research thread of field-based coordination, i.e., the study of Machine Learning techniques to improve existing AC coordination algorithms. Specifically, we adopt a *Reinforcement Learning (RL)-based approach*—where an agent learns from experience how to behave in order to maximise delayed cumulative rewards [38]. We devise a general methodology that somewhat resembles the notion of *sketching* in program synthesis [36]: a template program is given and *holes* are filled with actions determined through search. In our case, the program is the AC specification of a coordination algorithm, and holes are filled with actions of a policy learnt through Hysteretic Q-Learning [25]. We consider the case of the classic gradient algorithm, a paradigmatic and key building block of self-organising coordination [41,7,10]: we show via simulations that the system, after sufficient training, learns an improved way to compute and adjust gradient fields to network perturbations.

In the rest of the paper: Section 2 offers background on AC and RL; Section 3 discusses the integration of RL and AC, and the use of the approach to improve the basic gradient algorithm; Section 4 provides an experimental evaluation of the proposed approach; Section 5 summarises results and future research.

2 Background

In this work, we contribute to *field-based coordination* [45,24,40,23,41], a well-known nature-inspired approach [32] to coordination that exploits computational mechanisms inspired by the force fields of physics. Use of *artificial force fields* for navigation and obstacle avoidance has been explored since the 90s [45]. In *co-fields* [24], fields produced by agents or the environment are used to drive the activities of agent swarms. In TOTA (Tuples on the air) [23], tuples spread over the network to model dynamic distributed data. In Aggregate Computing [8,41], the field-based coordination approach we consider in this work, fields are used as an abstraction for functionally expressing CAS behaviour.

We recap Aggregate Computing in Section 2.1. Then, we review RL in Section 2.3, to prepare the ground for our contribution, *Reinforcement Learning-based Aggregate Computing* (Section 3).

2.1 Aggregate Computing

Aggregate Computing [8,41] is a paradigm for CAS programming. The approach generally assumes a system model of *neighbour-interacting devices* that work at asynchronous *rounds* of *sense-compute-act* steps. On such an execution model, self-organising collective behaviour is expressed in terms of functional manipulations of *computational fields* [41]: maps from devices to values. A field can denote, e.g., what different devices sense from the environment, or the outputs of their computations. The *Field Calculus (FC)* [41] is a core functional language that captures the key constructs needed to properly manipulate fields in order to express collective adaptive computations; they cover state evolution, communication, and computation branching. The main benefit of AC/FC is its *compositionality*: the ability to abstract collective adaptive behaviours into reusable functions that can be composed together to build more complex behaviours.

The FC is implemented by aggregate programming languages such as *ScaFi* (**Sca**la **Fi**elds) [14]. So, in practice, developing a CAS using this paradigm amounts to: (i) writing an *aggregate program* using e.g. ScaFi; (ii) setting up an AC middleware (for simulation or concrete distributed systems) to handle the scheduling of computations and communications; (iii) deploying and configuring the middleware and the program on a network of nodes. The approach has proven effective to implement various kinds of coordination services [15] and self-* applications in domains like crowd management [8], swarm robotics [11], and smart cities [12]—see [41] for a recent review.

In the following, we summarise the computation model and the FC/ScaFi language, which are essential to understand the contribution and case study.

System Model For an aggregate program to yield collective adaptive behaviour, *ongoing* computation and communication are needed. An individual atomic step of a device is called a *round* and consists of the following:

1. *context acquisition*: the device collects information from the sensors, and the most recent messages received from each neighbour (including the device itself—to model state);
2. *program evaluation*: the aggregate program is evaluated against the acquired context, yielding an *export*, namely a message to be sent to neighbours for coordination purposes and that is implied by the use of communication constructs in the aggregate program;
3. *export sharing*: the export is sent to the neighbours;
4. *actuations*: the export also includes information that can be used to drive actuators in the local node.

Rounds execution is completely asynchronous: there is no global clock or barrier to coordinate the aggregate. Scheduling of rounds might be periodic or reactive [34], and messages from neighbours are assumed to be retained for some configurable amount time. Such asynchrony, combined with local interaction, promotes scalability. The combination of the aggregate program logic and such a collective and periodical execution promotes the emergence of globally coherent results.

Field Calculus The main constructs that capture the essential aspects for programming self-organising systems with FC are:

- *Stateful field evolution* — expression `rep(`e_1`) {(`x`) =>` e_2`}` describes a field evolving in time. e_1 is the initial field value and the function $(x) => e_2$ defines how the field changes round by round substituting (x) with the value of the previous computed field (at the beginning $(x) = e_1$).
- *Neighbour interaction* — expression `nbr{`e`}` involves evaluation of e, sharing of the corresponding local value with neighbours, and observation of the neighbours' evaluations of e. Then, `*hood` operators can be used to locally reduce such neighbouring fields to values. For instance, for a device, `minHood` returns the minimum value of e found in its neighbourhood.
- *Domain partitioning* — expression `branch(`e_0`){`e_1`}{`e_2`}` splits the computational field into two non-communicating domains hosting isolated subcomputations: e_1 where e_0 is true, and e_2 where e_0 is false.

Full coverage of FC/ScaFi is beyond the scope of this paper. A more comprehensive presentation is given in [41]. As a key example, we introduce the gradient algorithm, which will be considered Sections 3 and 4.

2.2 The gradient building block

A *gradient* [30,24,4] is a field mapping each device in the system with its minimum distance from the closest *source* device. A *(self-healing) gradient algorithm* is one that computes a gradient field and automatically adjusts it after changes in the source set and the connectivity network. This algorithm is important as it often recurs as part of higher-level self-organising algorithms, such as information

flows [44], distributed data collection [3], and regional network partitioning [13]. A simple implementation, which we call *classic gradient*, can be expressed in FC as follows:

```
def gradient(source, metric) { // source is a Boolean field
  rep(infinity) {
    g => mux(source) { 0 } { minHoodPlus(nbr(g) + metric())}
  }
}
```

where `metric` is 0-ary function that evaluates the distance between two neighbours; `mux(c){e1}{e2}` is a conditional expression selector which evaluates all its arguments and selects `e1` if `c` is true or `e2` otherwise; and `minHoodPlus` selects the minimum of its argument across the neighbourhood without considering the contribution of the device itself. A repeated evaluation of this program will self-stabilise to the field of minimum distances from the sources, i.e., it will eventually converge to the ideal gradient field once the inputs and topology stop changing.

Though working and self-stabilising, this algorithm suffers some problems [4]. One is the *rising value problem* (also known as *count to infinity*): due to the repeated minimisation of all the contributions, the system handles well the situations when the output needs to drop (e.g. a new source enters the system) but it instead reacts slowly when the output needs to rise (e.g. a source node turns off). In literature, several heuristics are proposed to tackle the problems of the classical gradient [4]. One of them is CRF (Constraint and Restoring Force) [6]. Its goal is to deal with the problem by enforcing a constant rising speed when nodes recognise a local slow rising of the gradient field. To this end, each node is affected by a set of constraints (i.e. nodes that have a lower gradient value). If a node finds that it is slowly rising (i.e. there are no more constraints) then it increases its output at a fixed velocity, ignoring its neighbours. Otherwise, the output of the gradient follows the classical formula.

2.3 Reinforcement Learning

Reinforcement Learning [38] is a generic framework used to structure *control problems*. The focus is on *sequential* interactions between *agents* (i.e. entity able to act) and an *environment* (i.e. anything outside the control of agents). At each discrete time step t, an agent observes the current environment *state* s_t (i.e. the information perceivable by an agent) and selects an *action* a_t through a *policy* π (i.e. a probabilistic mapping between state and action). Partly as a consequence of its action, at the next time step $t+1$ the agent finds itself in state s_{t+1} and receives a *reward* r_{t+1}, i.e. a scalar signal quantifying how good the action was against the given environment configuration. The goal of RL is to *learn* a policy π^* that maximises the long-term return G (i.e. the cumulative reward) through a *trial-and-error* process. Different problems can be devised in these settings, like video games [2], robotics [20], routing [22], etc.

This general framework is supported by Markov Decision Process (MDP), a mathematical model that describes the environment evolution in sequential decision problems. A MDP consists of a tuple $< \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R} >$ in which:

- $\mathcal{S}$ denotes the set of states;
- $\mathcal{A}$ is the set of actions;
- $\mathcal{P}(s_{t+1}|s_t, a_t)$ define the probability to reach some state s_{t+1} starting from s_t and performing a_t (i.e. transition probability function);
- $\mathcal{R}(s_t, a_t, s_{t+1})$ devise a probabilistic reward function.

In MDP, $\mathcal{R}$ is memory-less, namely the next environment state depends only on the current state. Typically, in RL problems, agents do not have access to $\mathcal{R}$ or $\mathcal{P}$ but they can rely only on the experience (s_t, a_t, r_t) sampled at a time step. Therefore, G is defined as the discounted sum of reward a possible future trajectory τ (i.e. a sequence of time steps):

$$G_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots + \gamma^T r_{t+T} = \sum_{k=t}^{T} \gamma^{k-t} r_k \tag{1}$$

Where $0 \leq \gamma \leq 1$ is the *discount factor*, that is how much the future reward impacts the long-term return. Finally, the RL goal can be expressed as the maximisation of the *expected* long-term return following a policy π:

$$J = \mathbb{E}_\pi \Big[G_t \Big] = \mathbb{E}_\pi \Big[\sum_{k=t}^{T} \gamma^{t-k} r_k \Big] \tag{2}$$

The RL algorithms classification depends on how we derive the π^* (i.e. the optimal policy) according to J. In particular, *value-based* methods learn one further function (Q^π or V^π) to derive π^*. V^π is the value function that evaluates how good (or bad) a *state* is according to the long-term return following the policy π (*expected value*). It is defined as:

$$V(s)^\pi = \mathbb{E}_\pi \Big[G_t | s_t = s \Big] \tag{3}$$

Q^π is the corresponding value fuction that evaluates *state-action* pairs:

$$Q(s, a)^\pi = \mathbb{E}_\pi \Big[G_t | s_t = s, a_t = a \Big] \tag{4}$$

Policies could be defined through value functions. In particular, a greedy policy based on Q function is the one that always chooses the action with the highest value in a certain state: $\pi(s) = \arg\max_a(Q(s, a))$.

Q-Learning [42] is one of the most famous value-based algorithms. It aims at finding the Q^* (i.e. the Q function associated with π^*) by incrementally refining a Q table directly sampling from an unknown environment. Particularly, this is done through a temporal difference update performed at each time step:

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha * [r_t + \gamma * \arg\max_a(Q(s_{t+1}, a)) - Q(s_t, a_t)] \tag{5}$$

Where α is the learning rate (i.e. how much new information will influence the learned Q at each update). The agent typically follows a ϵ-greedy policy (*behavioural* policy, the function chooses a random action with a ϵ probability) to balance the *exploitation* and *exploration* trade-off. Using Q^* we could extract the π^* greedy policy (*target* policy).

Nowadays, Q-Learning is applied in various fields, ranging from robotics to wireless sensor networks and smart grids [17]. However, one of the most challenging settings in which Q-Learning could be applied is when the learning process has to deal with multiple concurrent learners, namely a multi-agent system.

Multi-Agent Reinforcement Learning RL was originally proposed as a framework to control a *single* agent. However, in CASs we are interested in *many* agents that interact in a common environment. The study of learning in these settings is known as Multi-Agent Reinforcement Learning (MARL) [39].

A straightforward way to apply RL algorithms to multi-agent settings, called *independent learning* (IL) approach [39], consists in deploying a learning process *for each* agent and considering other agents as *part* of the environment. However, applying single-agent algorithms as-is would probably lead to bad results, due to the non-stationarity of the environment induced by concurrent learning, and stochasticity [26]. Therefore, different algorithms are proposed to handle those issues in IL settings, such as Hysteretic Q-Learning [25] and Distributed Q-Learning (DQL) [21]. Hysteretic Q-Learning (HQL) aims at managing the stochasticity problems exploiting an *optimistic* heuristic. The idea is to give more importance to good actions than bad actions – more frequent due to the concurrent learning – reducing the policies oscillation during the learning. To this aim, HQL introduces two learning rates (α and β) to weigh the increase and decrease of Q values. The update equation becomes:

$$\delta(t) = r_t + \gamma \arg\max_a(Q(s_{t+1}, a)) - Q(s_t, a_t) \tag{6}$$

$$Q(s_t, a_t) = \begin{cases} Q(s_t, a_t) + \alpha\,\delta(t) & \text{if } \delta \geq 0 \\ Q(s_t, a_t) + \beta\,\delta(t) & \text{else} \end{cases} \tag{7}$$

In this study (cf. Section 4), we use HQL as reference RL algorithm since it has a strong empirical track for cooperative multi-agent systems [26,46,5]. Moreover, although structurally similar to DQL, it can be used in non-deterministic MDPs—a typical setting for CASs. Indeed, DQL is a particular case of HQL where $\beta = 0$. However, in doing so, the agents tend to overestimate the Q value due to the optimistic settings because they do not consider the environment noise.

RL could also be used in multi-agent settings through a central controller that learns exploiting system-wide information. The learning problem thus becomes a single agent one in which the agent consists of the Cartesian product of all the system nodes. However, this solution cannot be applied to CASs, due to the many-agent settings, openness, and the lack of a central entity. Novel approaches are structured in between independent learners and centralised

learners techniques—the so-called Centralised Training Decentralised Execution (CTDE) [16]. In CTDE, a central agent performs the learning process and generates distributed controllers, one for each agent. CTDE is typically applied in offline learning through the use of simulations. This way, it is possible to leverage global information at simulation time, but then the controllers are completely independent of this central authority. So, when the learner finds a good policy, it can be removed from the system.

Although CTDE allows RL to be used in environments with many agents, the learner must consider a large population of agents at simulation time, leading to sample inefficient algorithms. A solution to this problem in similar CASs (e.g. swarm robotics [37]) is to add another constraint, namely the agents' homogeneous and cooperative behaviour [33]. With this assumption, the learner only has to find a single strategy that applies to the whole system.

3 Reinforcement Learning-based Aggregate Computing

3.1 On Integrating Machine Learning and Aggregate Computing

As anticipated in Section 2.1, the behaviour of an aggregate system depends on the interplay of three main ingredients: (i) the aggregate program, expressing conceptually the global behaviour of the entire system, and concretely the local behaviour of each individual node in terms of local processing and data exchange with neighbours; and (ii) the aggregate execution model, promoting a certain dynamics of the system in terms of topology management (e.g., via neighbour discovery), execution of rounds, scheduling of communications; and (iii) the environment dynamics. While the latter cannot be controlled, the importance of the first element is reflected by research on the design of novel algorithms (cf. [41,4]), while the second element is studied w.r.t. the possibility of tuning and adaptivity according to available technology and infrastructure or the dynamics of the environment. Since tuning programs or execution details to specific environments or adapting those to changing environments can be burdensome, it makes sense to consider the use of Machine Learning techniques to let a system *learn* effective strategies for unfolding collective adaptive behaviour.

3.2 Aggregate Programs Improvement through RL

In this work, we focus on *improving aggregate programs* by learning effective local actions within a given algorithmic schema—an approach similar to the *sketching* technique for program synthesis [36]. As a learning framework, we use RL as we deal with systems of agents performing ongoing activities by interacting with one another and the environment. From a local viewpoint, an AC device is an agent acting based on the local context it perceives (sensor readings, device state, and messages from neighbours), and this matches the RL execution model very well.

So, our long-term goal is to integrate RL into the AC "stack" (i.e., across the platform, language, and library levels) to improve the collective behaviour

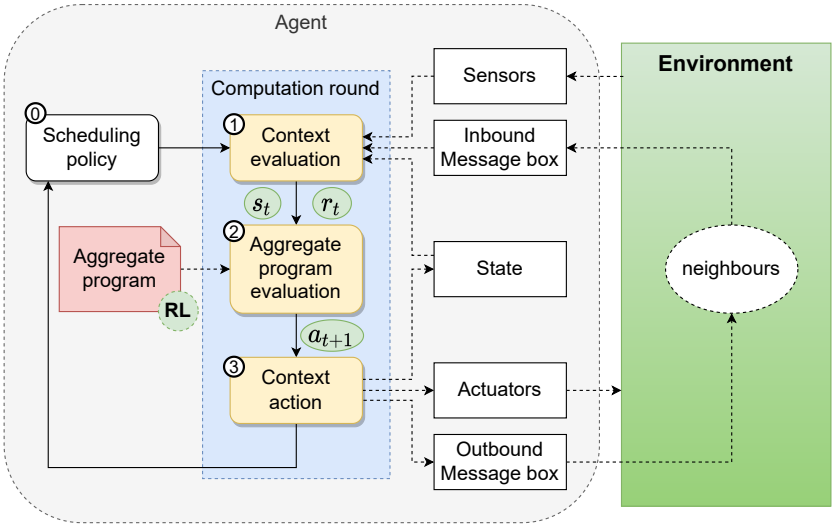


Fig. 1: Integration of RL within the AC control architecture [11]. The RL state and reward concepts build upon the context, given by environment and neighbour data. The designer configures action points where learning can improve the aggregate computation. The actions selected by the learned policies will then affect the environment (via actuators) and neighbours (via outbound messages).

defined by ScaFi aggregate programs in terms of *efficiency* (i.e., reducing the resource usage maintaining the same functional interface), *efficacy* (i.e., synthesising more stable and faster converging behaviours) and *adaptability* (i.e., the same program works against different environments).

As a first contribution, summarised in Figure 1, in this work we integrate RL within the AC control architecture in order to support learning of good collective behaviour sketched by a given aggregate program. Specifically, we focus on improving AC building blocks (such as the gradient algorithm covered in Section 2.2) through learning, leading toward a so-called *Reinforcement Learning-based Aggregate Computing*. Learning, thus, does not replace the AC methodology for defining the programs but it is best understood as a technique that supports and improves the AC algorithm design process.

3.3 Building blocks Refinement

A major advantage of AC as a programming model is its *compositionality*: complex collective behaviours (e.g., the maintenance of a multi-hop connectivity channel) can be expressed through a combination of building blocks capturing simpler collective behaviours (e.g., gradients). Since building blocks are fundamental bricks of behaviour that often recur in programs, their bad and good qualities (cf., convergence speed, stability, etc.) tend to amplify and affect behaviours that depend on them. Therefore, research tends to investigate refined

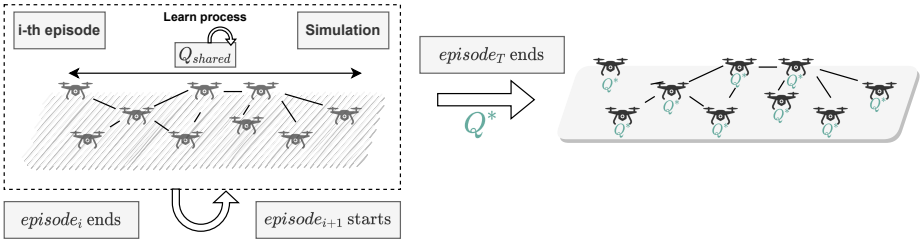


Fig. 2: Reinforcement Learning schema used in our simulations. The learning algorithm is applied at simulation time (for T episodes) improving a shared Q table. At the deployment time then, the agents exploit a local copy of the optimal Q^* table found by learning.

variants of building blocks that provide the same functionality but are more effective or efficient under specific assumptions or contexts (e.g., high mobility, situations where stability is more important than precision, etc.) [4,3]. With a library of algorithms, the designer can choose the best combination of building blocks that are well-fitted for a given environment, and even substitute a building block with a variant implementation without substantially affecting the application logic. In general, a building block can be seen as a black box (function) that takes a set of input fields (e.g. metric, perception fields, constant fields, etc.) and yields an output field. To increase its flexibility, such a function could leverage a *refinement policy* able to affect the behaviour of the building block over time or in a certain situation. This policy could be a feedback loop, hysteresis, or custom logic to solve a specific problem. We aim at structuring the learning of refinement policies through RL [1]. Our idea is that it should not be the designer who codes a particular block to be used, but that it is the learning algorithm that understands, given a global policy to be optimised following a utility function, what actions need to be activated.

However, using Machine Learning to improve AC programs exposes several non-trivial challenges:

- *scale-free behaviours*: the learned policy should work in small networks as well as in large networks as fostered by the AC abstractions;
- the state typically depends on *continuous* value as computational fields are often associated with continuous data (e.g. temperature or distance fields);
- *multi-agent credit assignment problem*: it is not easy to adequately reward and credit local actions for their contribution to the eventual convergence to the "target" field denoting the desired, emergent, collective result.

The AC system model is based on *cooperative* and *homogeneous* behaviour. Indeed, when a node participates in an aggregate system, it has to execute an aggregate program shared within the whole system, which yields different outcomes according to the contexts on which it gets evaluated. This leads to handling homogeneous MARL, hence our goal will be to find one policy for the

entire ensemble, partially solving the scale-free behaviour problem, since the learned policy will not depend on the system's size.

The continuous and variable state problems are typically tackled using Deep Learning to learn the right state representation for a given problem [28]. In this case, instead, we used a handcraft feature engineering process since we are more interested in devising a general design process.

Concerning the *multi-agent credit assignment problem*, we decided to use offline learning in a typical CTDE setting (Figure 2). This way, we assess the influence of an individual in comparison to the entire system, which cannot be done in online learning due to the difficulty of achieving global snapshots of the system in a reasonable time.

Learning Schema The learning algorithm is seen as a state (s_t) evolution function in which the nodes try to apply a correction factor (`update`) following a policy (π^Q_{target} or $\pi^Q_{behavioral}$) refined by learning. The state is built from a neighbourhood field of the building block generic output (o_t) passed as input. Listing 1.1 exemplifies the general program structure used to combine RL with AC for improving building blocks. The branching operator (`branch`) on `learn` condition makes it possible to use the CTDE schema since when the `learn` is `false` there is no need for a central entity (`simulation`). The Q table is gathered using `sense`, a ScaFi operator used to query sensors and collect data from them. At simulation time, Q is a shared object, but at runtime each agent owns a local table.

```
def optBlock(o_{t-1}) { // learning as a field that evolves in time
  rep((s_0, a_0, o_0)) { // s_0, a_0 context dependent
    case (s_{t-1}, a_{t-1}, _) => {
      val Q = sense("Q") // global during training, local during execution
      val o_t = update(o_{t-1}, a_{t-1}) // local action
      // state from the neighbourhood field program output
      val s_t = state(nbr(o_t))
      val a_t = branch(learn) { // actions depends on learn condition
        val r_{t-1} = reward(o_t, simulation) // simulation is a global object
        simulation.updateQ(Q, s_{t-1}, a_{t-1}, r_{t-1}, s_t) // Q update
        ~ π^Q_{behavioural}(s_t) // sample from a probabilistic distribution
      } {
        π^Q_{target}(s_t) // greedy policy, no sampling is needed
      }
    }
    (s_t, a_t, o_t)
  }._3 // select the output from the tuple
}
```

Listing 1.1: ScaFi-like pseudocode description (implemented in the simulation) for value-based RL algorithm applied AC. `state`, `update`, `reward` are block specific.

Finally, the produced o_t is returned to the caller.

In this case, we encoded the learning process with AC itself. Though we could have extracted the learning process from the AC program, we took this decision because: (i) it allows us to extend learning by exploiting neighbourhood Q table fields – so we can think of doing non-homogeneous learning in different areas of the system; (ii) the scheme for taking state and choosing actions is the same as the one we would need for learning, so the only difference is in the branch; and (iii) it can simply be extended to online learning.

3.4 Reinforcement Learning-based gradient block

The gradient block (cf. Section 2.2) could be generalised as follows:

```
def gradientOpt(source, metric, opt) {
  rep(infinity) { g => mux(source) { 0 } { opt(g, metric)) } }
}
```

where `opt` is a function that determines how to evolve the gradient based on the field of current gradient values `g` and current `metric` (estimation of distances to neighbours). In this article, we consider `opt` as a *hole* that a RL algorithm will fill through raw experience with actions aiming at incrementally constructing a gradient field, hopefully mitigating the *rising-value* issue (cf. Section 2.2). To frame our learning problem, we adopt the above-described general schema (Figure 2). The state and action functions are inspired by the CRF algorithm. The `state` function must hold enough information to know when agents should speed up the local rising of values. In this case, we encode the state as the difference between the output perceived by a node and the minimum and maximum gradient received by its neighbours: $s_t = (|min_t - o_t|, |max_t - o_t|)$. These data have to be discretised; otherwise, the search space would be too big and the solution could suffer from overfitting. The discretisation is ruled by two variables, $maxBound$ and $buckets$. The former constrains the output to be between $-radius * maxBound$ and $radius * maxBound$ (where $radius$ is the maximum communication range of the nodes). The values outside that range will be considered as the same state. The $buckets$ variable rules the division count of the given range. Finally, we stack two time steps in order to encode history inside the agent state: $h_t = [(s_{t-1}, s_t)]$. h_t is used as the state function for our RL algorithm. Hence, the cardinality of the state space of $|s_t| * |s_t| = buckets^4$. The action space is divided into two action types: `ConsiderNeighborhood` is the action that will produce the classic gradient evaluation, while `Ignore(velocity)` ignores the neighbour data and increases the gradient at a given `velocity`. So, the `update` function is defined as:

```
def update(o_{t-1}, a_{t-1}, metric) { // o_{t-1} is the previous gradient output
  val g_classic = minHoodPlus(nbr(o_{t-1}) + metric())
  match a_{t-1} { // scala-like pattern matching
    case ConsiderNeighborhood => g_classic
    case Ignore(velocity) => o_t + velocity * deltaTime()
  }
}
```

Name	Values
(γ)	[0.4 – 0.7 – 0.9]
(ϵ_0, θ)	[(0.5,200) – (0.01,1000) – (0.05,400) – (0.02,500)]
(β, α)	[(0.5,0.01) – (0.5,0.1) – (0.3,0.05) – (0.2,0.03) – (0.1,0.01)]
(buckets, maxBound)	[(16,4) – (32,4) – (64,4)]

Table 1: Summary of the simulation variables. A simulation is identified by a quadruple (i, j, k, l) of indexes for each variable.

Finally, the `reward` function is described as follows:

```
def reward(o_t, simulation) {
  if(o_t - simulation.rightValueFor(mid()) ~= 0) { 0 } { -1 }
}
```

where `mid` returns the field of node identifiers. The idea is to push the nodes to produce an output that is very close to the ideal, correct gradient value as provided by an oracle (`simulation.rightValuefor()`). When this is the case, we provide a value equals to 0; instead, when the value is different from the expected one, we provide a small negative reward, -1, in order to push the nodes to quickly seek the situation where the actual and ideal value match.

4 Evaluation

To evaluate our approach, we run a set of simulated experiments and verify that an aggregate system can successfully learn an improved way to compute a gradient field (cf. the gradient block described in Section 2.2). To this purpose, we adopt Alchemist [35], a flexible simulator for pervasive and networked systems that comes with a support for aggregate systems programmed with ScaFi [14]. The source code, data, and instructions for running the experiments have been made fully available at a public repository[1], to promote reproducibility of results.

4.1 Simulation setup

The simulated system consists of N devices deployed in a grid. We use two kinds of grid-like environments. They both have the same width (200 m) and distance between nodes (5 m), but they differ in the row count. In one case, only one row exists (so the nodes are placed in a line). In the other case, there are five rows. The total number (N) of agents is then defined as $200/5 * rows$. So in the first case, we have a total of 40 agents, in the second case we have 200 agents. Each node asynchronously fires the round evaluation at 1 Hz. The leftmost and the rightmost agents are marked as source nodes. Each simulated episode lasts 85 s (T). For simulating the slow rising problem, we drop the left source at 35 s (C_s), and so the left part of the system starts to rise until eventually stabilising to the

[1] https://github.com/cric96/experiment-2022-coordination

correct computational field. An entire simulation lasts $N_E = 1200$ episodes, in which in the first $N_L = 1000$ the system uses RL to improve a shared Q table, and then in the last $N_T = 200$, the system deploys the Q table found in each agent. In these last runs, the agents act following the greedy policy.

As learning algorithm, we used Hysteretic Q-Learning (cf. Section 2.3). As behavioural policy, we use an ϵ-greedy with an exponential decay to balance the exploration-exploitation. We make this choice because without using the decay the policy found tends to be not optimised (i.e. the exploration is preferred w.r.t exploitation). At each episode i, the ϵ value is updated as $\epsilon_i = \epsilon_0 \cdot e^{i/\theta}$.

Several variables are used, summarised in Table 1, so we perform a grid search to find the optimal combination. To evaluate a particular configuration, we verified the total error performed in the last N_T episodes. This is calculated from the error performed by each node i at each time step t:

$$error_i^t = |gradient_i^t - simulated_i^t| \tag{8}$$

Then for each time step t, we evaluate the average system error as:

$$error_t = \frac{1}{n} \sum_{i=0}^{N} error_i^t \tag{9}$$

And finally, the error of each episode is evaluated as:

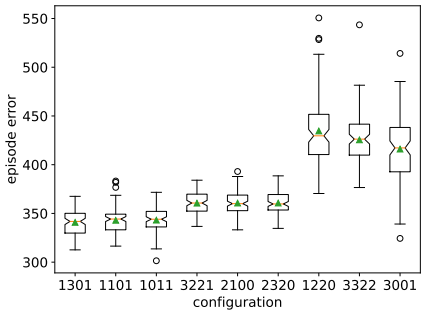
$$error_{episode} = \sum_{t=0}^{T} error_t \tag{10}$$

We choose the configuration by observing the box plots (Figure 3a) and taking the lowest average error in the last N_T episodes.

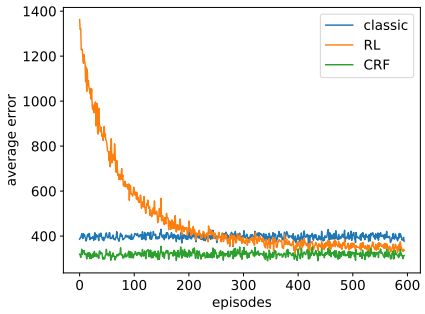
4.2 Results and Discussion

Figure 3 shows the performance of our Reinforcement Learning-based gradient algorithm. Figure 3a was used to choose which configuration was the best. Figure 3b shows the error trend as the episodes change. The second row shows the trend of the mean error over the last N_T episodes. The coloured area under the curve defines the standard deviation. The dashed vertical line is the time at which the source change occurs. Finally, the last row shows the average output of the various algorithms. In the following we discuss the result reached with the best configuration, that has $\gamma = 0.9$, $\epsilon_0 = 0.5$, $\theta = 200$), $\alpha = 0.3$ and $\beta = 0.05$.

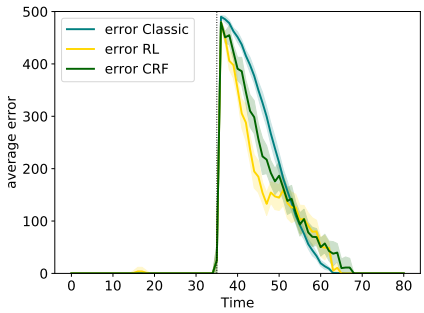
Our goal was to create a solution that outperforms the classic gradient against the rising-value problem. In fact, the system eventually learns how to speed up the gradient rising: observing the Figure 3b the $error_{episode}$ of the new algorithm is lower than the error produced by the standard solution. In particular, this means that the agents learn the moment when they should ignore their neighbourhood and increase the output with a certain velocity (i.e. using the `Ignore` action).



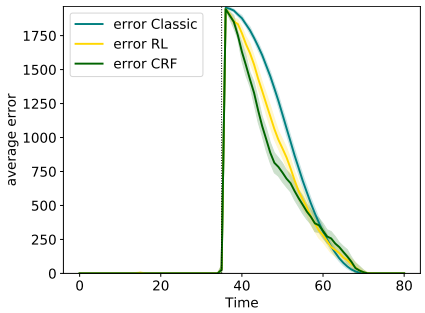
(a) Box plots of last G_s episode.



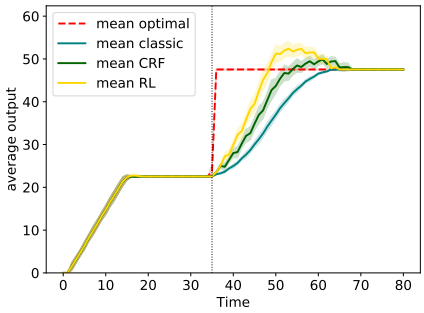
(b) Learning progress of the best result.



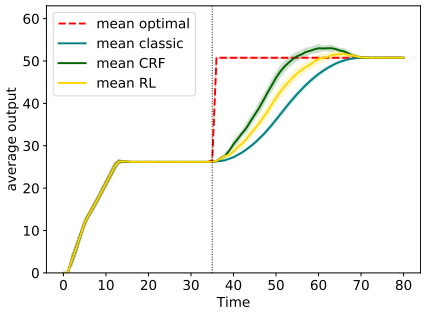
(c) Error evolution with 40 agents



(d) Error evolution with 200 agents



(e) Output evolution with 40 agents



(f) Output evolution with 200 agents

Fig. 3: Performance of our RL-based gradient algorithm with `velocity` = 20.

This intuition is enforced by Figures 3c to 3f. In particular, in Figures 3c and 3e the behaviour is more evident due to the reduced number of agents: when the error is maximised (due to the source that disappears), the error decreased faster than the naive gradient (and, consequently, the output is faster growing). Furthermore, we can also observe that the overall algorithm behaviour is comparable with the CRF handcrafted solution for the rising problem. Indeed,

both have a phase of speed up followed by another phase of overestimation (i.e. the system-wide output is greater than the true gradient field output) that eventually brings the system to slowly reaches the correct value. Moreover, not only the RL-based solution has similar behaviour to CRF, but also it has comparable performance – it means that the aggregate reaches a near-optimal policy with these state-action-reward settings.

We want also to underline that the policy learned is *one* and shared with the whole system. Thereby, the policy can be easily scaled in deployments with different node counts. In this case, the *same* function outperforms our baseline in a system with different nodes and deployment configurations.

Finally, we want to stress that the nodes do not fire synchronously, and thus there is not any kind of global and shared clock: any node round evaluation order reaches the same behaviour in our test deployments. This, again, makes it possible to use the same policy in different scenarios due to the unknown local aggregate programs evaluation order.

5 Conclusion

This paper discusses the integration of Aggregate Computing – a programming approach for Collective Adaptive Systems – and Reinforcement Learning, with the goal of fostering the design of collective adaptive behaviour. In particular, we propose to use RL as a means to improve building block AC algorithms. Our approach is applied to improving the gradient algorithm, one of the key AC algorithms, where learning is performed through Hysteretic Q-Learning. We evaluate the approach through synthetic experiments comparing the reactivity of different gradient algorithms in dealing with the rising value problem.

This work is the first effort towards Reinforcement Learning-based Aggregate Computing. In fact, there are still many aspects that need to be analysed in detail both at the conceptual and practical levels. First of all, the approach could be tuned to learn gradient strategies for smoothness or maximal reactivity in highly variable scenarios, and compared with state-of-the-art algorithms like BIS and ULT [4]. Secondly, the approach could be systematically applied to other building blocks as well [41]. Very interesting would also be the application of Machine Learning at the aggregate execution platform level, e.g. to improve the round frequency to reduce power consumption, reduce the amount of data exchanged between neighbours, or support opportunistic re-configuration of aggregate system deployments.

Acknowledgements This work has been supported by the MIUR FSE REACT-EU PON R&I 2014-2022 (CCI2014IT16M2OP005).

References

1. Aguzzi, G.: Research directions for aggregate computing with machine learning. In: IEEE International Conference on Autonomic Computing and Self-Organizing

Systems, ACSOS 2021, Companion Volume. pp. 310–312. IEEE (2021). https://doi.org/10.1109/ACSOS-C52956.2021.00078
2. Arulkumaran, K., Deisenroth, M.P., Brundage, M., Bharath, A.A.: Deep reinforcement learning: A brief survey. IEEE Signal Process. Mag. **34**(6), 26–38 (2017). https://doi.org/10.1109/MSP.2017.2743240
3. Audrito, G., Casadei, R., Damiani, F., Pianini, D., Viroli, M.: Optimal resilient distributed data collection in mobile edge environments. Comput. Electr. Eng. **96**(Part), 107580 (2021). https://doi.org/10.1016/j.compeleceng.2021.107580
4. Audrito, G., Casadei, R., Damiani, F., Viroli, M.: Compositional blocks for optimal self-healing gradients. In: 11th IEEE International Conference on Self-Adaptive and Self-Organizing Systems, SASO 2017. pp. 91–100. IEEE Comp. Soc. (2017). https://doi.org/10.1109/SASO.2017.18
5. Barbalios, N., Tzionas, P.: A robust approach for multi-agent natural resource allocation based on stochastic optimization algorithms. Appl. Soft Comput. **18**, 12–24 (2014). https://doi.org/10.1016/j.asoc.2014.01.004, https://doi.org/10.1016/j.asoc.2014.01.004
6. Beal, J., Bachrach, J., Vickery, D., Tobenkin, M.M.: Fast self-healing gradients. In: Proceedings of the 2008 ACM Symposium on Applied Computing (SAC). pp. 1969–1975. ACM (2008). https://doi.org/10.1145/1363686.1364163
7. Beal, J., Dulman, S., Usbeck, K., Viroli, M., Correll, N.: Organizing the aggregate: Languages for spatial computing. In: Formal and Practical Aspects of Domain-Specific Languages: Recent Developments, pp. 436–501. IGI Global (2013)
8. Beal, J., Pianini, D., Viroli, M.: Aggregate programming for the internet of things. Computer **48**(9), 22–30 (2015). https://doi.org/10.1109/MC.2015.261
9. Bucchiarone, A., D'Angelo, M., Pianini, D., Cabri, G., De Sanctis, M., Viroli, M., Casadei, R., Dobson, S.: On the social implications of collective adaptive systems. IEEE Technol. Soc. Mag. **39**(3), 36–46 (2020). https://doi.org/10.1109/MTS.2020.3012324
10. Casadei, R.: Macroprogramming: Concepts, state of the art, and opportunities of macroscopic behaviour modelling. CoRR **abs/2201.03473** (2022)
11. Casadei, R., Aguzzi, G., Viroli, M.: A programming approach to collective autonomy. J. Sens. Actuator Networks **10**(2), 27 (2021). https://doi.org/10.3390/jsan10020027
12. Casadei, R., Pianini, D., Placuzzi, A., Viroli, M., Weyns, D.: Pulverization in cyber-physical systems: Engineering the self-organizing logic separated from deployment. Future Internet **12**(11), 203 (2020). https://doi.org/10.3390/fi12110203
13. Casadei, R., Pianini, D., Viroli, M., Natali, A.: Self-organising coordination regions: A pattern for edge computing. In: Coordination Models and Languages - 21st IFIP WG 6.1 International Conference, COORDINATION 2019, Proceedings. Lecture Notes in Computer Science, vol. 11533, pp. 182–199. Springer (2019). https://doi.org/10.1007/978-3-030-22397-7_11
14. Casadei, R., Viroli, M., Audrito, G., Damiani, F.: FScaFi: A core calculus for collective adaptive systems programming. In: Leveraging Applications of Formal Methods, Verification and Validation: Engineering Principles - 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Proceedings, Part II. Lecture Notes in Computer Science, vol. 12477, pp. 344–360. Springer (2020). https://doi.org/10.1007/978-3-030-61470-6_21
15. Casadei, R., Viroli, M., Ricci, A., Audrito, G.: Tuple-based coordination in large-scale situated systems. In: Coordination Models and Languages - 23rd IFIP WG

6.1 International Conference, COORDINATION 2021, Proceedings. Lecture Notes in Computer Science, vol. 12717, pp. 149–167. Springer (2021). https://doi.org/10.1007/978-3-030-78142-2_10
16. Foerster, J.N.: Deep multi-agent reinforcement learning. Ph.D. thesis, University of Oxford, UK (2018)
17. Jang, B., Kim, M., Harerimana, G., Kim, J.W.: Q-learning algorithms: A comprehensive classification and applications. IEEE Access **7**, 133653–133667 (2019). https://doi.org/10.1109/ACCESS.2019.2941229
18. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. Computer **36**(1), 41–50 (2003). https://doi.org/10.1109/MC.2003.1160055
19. Kernbach, S., Schmickl, T., Timmis, J.: Collective adaptive systems: Challenges beyond evolvability. CoRR **abs/1108.5643** (2011)
20. Kober, J., Bagnell, J.A., Peters, J.: Reinforcement learning in robotics: A survey. Int. J. Robotics Res. **32**(11), 1238–1274 (2013). https://doi.org/10.1177/0278364913495721
21. Lauer, M., Riedmiller, M.A.: An algorithm for distributed reinforcement learning in cooperative multi-agent systems. In: Langley, P. (ed.) Proceedings of the Seventeenth International Conference on Machine Learning (ICML 2000), Stanford University, Stanford, CA, USA, June 29 - July 2, 2000. pp. 535–542. Morgan Kaufmann (2000)
22. Luong, N.C., Hoang, D.T., Gong, S., Niyato, D., Wang, P., Liang, Y., Kim, D.I.: Applications of deep reinforcement learning in communications and networking: A survey. IEEE Commun. Surv. Tutorials **21**(4), 3133–3174 (2019). https://doi.org/10.1109/COMST.2019.2916583
23. Mamei, M., Zambonelli, F.: Programming pervasive and mobile computing applications: The TOTA approach. ACM Trans. Softw. Eng. Methodol. **18**(4), 15:1–15:56 (2009). https://doi.org/10.1145/1538942.1538945, https://doi.org/10.1145/1538942.1538945
24. Mamei, M., Zambonelli, F., Leonardi, L.: Co-fields: A physically inspired approach to motion coordination. IEEE Pervasive Comput. **3**(2), 52–61 (2004). https://doi.org/10.1109/MPRV.2004.1316820
25. Matignon, L., Laurent, G.J., Fort-Piat, N.L.: Hysteretic q-learning : an algorithm for decentralized reinforcement learning in cooperative multi-agent teams. In: 2007 IEEE/RSJ International Conference on Intelligent Robots and Systems. pp. 64–69. IEEE (2007). https://doi.org/10.1109/IROS.2007.4399095
26. Matignon, L., Laurent, G.J., Fort-Piat, N.L.: Independent reinforcement learners in cooperative markov games: a survey regarding coordination problems. Knowl. Eng. Rev. **27**(1), 1–31 (2012). https://doi.org/10.1017/S0269888912000057
27. Mo, Y., Beal, J., Dasgupta, S.: An aggregate computing approach to self-stabilizing leader election. In: 2018 IEEE 3rd International Workshops on Foundations and Applications of Self* Systems (FAS*W), Trento, Italy, September 3-7, 2018. pp. 112–117. IEEE (2018). https://doi.org/10.1109/FAS-W.2018.00034
28. Mousavi, S.S., Schukat, M., Howley, E.: Deep reinforcement learning: An overview. CoRR **abs/1806.08894** (2018)
29. Müller-Schloer, C., Sick, B.: Controlled emergence and self-organization. In: Würtz, R.P. (ed.) Organic Computing, pp. 81–103. Understanding Complex Systems, Springer (2008). https://doi.org/10.1007/978-3-540-77657-4_4
30. Nagpal, R., Shrobe, H.E., Bachrach, J.: Organizing a global coordinate system from local information on an ad hoc sensor network. In: Information Processing in Sensor Networks, 2nd International Workshop, IPSN 2003, , Proceedings. Lecture

Notes in Computer Science, vol. 2634, pp. 333–348. Springer (2003). https://doi.org/10.1007/3-540-36978-3_22
31. Nicola, R.D., Jähnichen, S., Wirsing, M.: Rigorous engineering of collective adaptive systems: special section. Int. J. Softw. Tools Technol. Transf. **22**(4), 389–397 (2020). https://doi.org/10.1007/s10009-020-00565-0
32. Omicini, A.: Nature-inspired coordination for complex distributed systems. In: Intelligent Distributed Computing VI - Proceedings of the 6th International Symposium on Intelligent Distributed Computing - IDC 2012. Studies in Computational Intelligence, vol. 446, pp. 1–6. Springer (2012). https://doi.org/10.1007/978-3-642-32524-3_1
33. Panait, L., Luke, S.: Cooperative multi-agent learning: The state of the art. Auton. Agents Multi Agent Syst. **11**(3), 387–434 (2005). https://doi.org/10.1007/s10458-005-2631-2
34. Pianini, D., Casadei, R., Viroli, M., Mariani, S., Zambonelli, F.: Time-fluid field-based coordination through programmable distributed schedulers. Logical Methods in Computer Science **Volume 17, Issue 4** (Nov 2021). https://doi.org/10.46298/lmcs-17(4:13)2021, https://doi.org/10.46298/lmcs-17(4:13)2021
35. Pianini, D., Montagna, S., Viroli, M.: Chemical-oriented simulation of computational systems with ALCHEMIST. J. Simulation **7**(3), 202–215 (2013). https://doi.org/10.1057/jos.2012.27
36. Solar-Lezama, A.: Program synthesis by sketching. University of California, Berkeley (2008)
37. Sosic, A., KhudaBukhsh, W.R., Zoubir, A.M., Koeppl, H.: Inverse reinforcement learning in swarm systems. In: Larson, K., Winikoff, M., Das, S., Durfee, E.H. (eds.) Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems, AAMAS 2017, São Paulo, Brazil, May 8-12, 2017. pp. 1413–1421. ACM (2017)
38. Sutton, R.S., Barto, A.G.: Reinforcement learning: An introduction. MIT press (2018)
39. Tan, M.: Multi-agent reinforcement learning: Independent versus cooperative agents. In: Utgoff, P.E. (ed.) Machine Learning, Proceedings of the Tenth International Conference, University of Massachusetts, Amherst, MA, USA, June 27-29, 1993. pp. 330–337. Morgan Kaufmann (1993). https://doi.org/10.1016/b978-1-55860-307-3.50049-6
40. Trzec, K., Lovrek, I.: Field-based coordination of mobile intelligent agents: An evolutionary game theoretic analysis. In: Knowledge-Based Intelligent Information and Engineering Systems, 11th International Conference, KES 2007, XVII Italian Workshop on Neural Networks, Proceedings, Part I. Lecture Notes in Computer Science, vol. 4692, pp. 198–205. Springer (2007). https://doi.org/10.1007/978-3-540-74819-9_25
41. Viroli, M., Beal, J., Damiani, F., Audrito, G., Casadei, R., Pianini, D.: From distributed coordination to field calculus and aggregate computing. J. Log. Algebraic Methods Program. **109** (2019). https://doi.org/10.1016/j.jlamp.2019.100486
42. Watkins, C.J.C.H., Dayan, P.: Technical note q-learning. Mach. Learn. **8**, 279–292 (1992). https://doi.org/10.1007/BF00992698
43. Wolf, T.D., Holvoet, T.: Emergence versus self-organisation: Different concepts but promising when combined. In: Engineering Self-Organising Systems, Methodologies and Applications (ESOA 2004). Lecture Notes in Computer Science, vol. 3464, pp. 1–15. Springer (2004). https://doi.org/10.1007/11494676_1

44. Wolf, T.D., Holvoet, T.: Designing self-organising emergent systems based on information flows and feedback-loops. In: Proceedings of the First International Conference on Self-Adaptive and Self-Organizing Systems, SASO 2007, Boston, MA, USA, July 9-11, 2007. pp. 295–298. IEEE Computer Society (2007). https://doi.org/10.1109/SASO.2007.16
45. Xiao, D., Hubbold, R.J.: Navigation guided by artificial force fields. In: Atwood, M.E., Karat, C., Lund, A.M., Coutaz, J., Karat, J. (eds.) Proceeding of the CHI '98 Conference on Human Factors in Computing Systems. pp. 179–186. ACM (1998). https://doi.org/10.1145/274644.274671
46. Xu, Y., Zhang, W., Liu, W., Ferrese, F.T.: Multiagent-based reinforcement learning for optimal reactive power dispatch. IEEE Trans. Syst. Man Cybern. Part C **42**(6), 1742–1751 (2012). https://doi.org/10.1109/TSMCC.2012.2218596, https://doi.org/10.1109/TSMCC.2012.2218596