Original Paper

Adaptive Behavior

Adaptive Behavior
2016, Vol. 24(1) 27–51

DOI: 10.1177/1059712315609412
adb.sagepub.com
SAGE

Learning programs is better than learning dynamics: A programmable neural network hierarchical architecture in a multi-task scenario

Francesco Donnarumma[1], Roberto Prevete[2], Andrea de Giorgio[3], Guglielmo Montone[4] and Giovanni Pezzulo[1]

Abstract

Distributed and hierarchical models of control are nowadays popular in computational modeling and robotics. In the artificial neural network literature, complex behaviors can be produced by composing elementary building blocks or motor primitives, possibly organized in a layered structure. However, it is still unknown how the brain learns and encodes multiple motor primitives, and how it rapidly reassembles, sequences and switches them by exerting cognitive control. In this paper we advance a novel proposal, a hierarchical programmable neural network architecture, based on the notion of *programmability* and an interpreter-programmer computational scheme. In this approach, complex (and novel) behaviors can be acquired by embedding multiple modules (motor primitives) in a single, *multi-purpose* neural network. This is supported by recent theories of brain functioning in which skilled behaviors can be generated by combining functional different primitives embedded in "reusable" areas of "recycled" neurons. Such neuronal substrate supports flexible cognitive control, too. Modules are seen as *interpreters* of behaviors having controlling input parameters, or *programs* that encode structures of networks to be interpreted. Flexible cognitive control can be exerted by a *programmer* module feeding the interpreters with appropriate input parameters, without modifying connectivity. Our results in a multiple *T*-maze robotic scenario show how this computational framework provides a robust, scalable and flexible scheme that can be iterated at different hierarchical layers permitting to learn, encode and control multiple qualitatively different behaviors.

Keywords

Programming neural networks, hierarchical organization, distributed representation, neuronal reuse, cognitive control

1 Introduction

Converging evidences in literature indicate that the capabilities of human and artificial agents to learn and control complex skilled behaviors are grounded on some mechanisms of compositionality of *primitives*. According to this view, almost all behaviors (including complex actions such as playing table tennis) can essentially be generated by combining simpler motor acts (primitives) which are picked out from a predetermined set of primitives by following rules. One can isolate three principles involved in the definition/comprehension of this mechanism as discussed in the following.

Reusable and adaptable primitives. Primitives can flexibly be used and re-used in order to construct different sequences of actions (d'Avella, Portone, Fernandez, & Lacquaniti, 2006; Thoroughman & Shadmehr, 2000). For example, the action of eating an apple can be broken down into a combination of multiple motor primitives. Some motor primitives would be responsible for reaching for the apple, some for grasping it and some for moving the apple toward one's own mouth. In other words, each primitive can be used (and re-used) in different tasks with the ability to adapt to the specific task "on the fly" (Pezzulo, Donnarumma,

[1]Institute of Cognitive Sciences and Technologies, National Research Council of Italy, Italy
[2]Dipartimento di Ingegneria Elettrica e Tecnologie dell'Informazione, Università degli Studi di Napoli Federico, Napoli
[3]KTH Royal Institute of Technology, Sweden
[4]Institut Neuroscience Cognition, Université Paris Descartes, France

Corresponding author:
Francesco Donnarumma, Institute of Cognitive Sciences and Technologies, National Research Council of Italy, Via San Martino della Battaglia, 44 Rome 00185, Italy.
Email: francesco.donnarumma@istc.cnr.it

Iodice, Prevete, & Dindo, 2015) and this mechanism is thought to be at the basis of a shared representation enabling social interaction with conspecifics (Candidi, Curioni, Donnarumma, Sacheli, & Pezzulo, 2015; Pezzulo, Donnarumma, & Dindo, 2013). Consequently, each primitive does not have to be interpreted as a specific sequence of motor commands, but rather as a behavior, i.e. it should be "fluid" and context dependent (Tani, Nishimoto, & Paine, 2008).

Distributed representation. One can individuate two kinds of behavior representation: *local* and *distributed* representation of behavioral modules (Paine & Tani, 2005). In the first case, the acquisition of novel behaviors consists of adding novel, *single-purpose* modules, each implementing one specific behavior (Haruno, Wolpert, & Kawato, 2003; Igari & Tani, 2009). This strong modularization of the behaviors (i.e. the fact that each behavior is essentially represented in a different module) provides flexible control and avoids problems of interference. By contrast, the distributed representation consists of generalizing the function of (at least some of) the modules. This idea is developed, for example, in a number of Artificial Neural Network (ANN) models (see e.g. Agmon & Beer, 2013; Araújo, Diniz, Passos, & Davids, 2014; Paine & Tani, 2005; Tani, Ito, & Sugita, 2004; Woodman, Perdikis, Pillai, Dodel, Huys, Bressler, & Jirsa, 2011), which can embed multiple behaviors in a single, *multi-purpose* neural network module. Importantly, recent theories of brain functioning provide support for the existence of a neural substrate which could implement such functionalities; they suggest that skilled behaviors can be generated by combining functional different primitives embedded in "reusable" (see Anderson, 2010) areas of "recycled" (Dehaene, 2005) neurons. In addition, a number of neurophysiological evidence suggests the presence of neural modules which exhibit drastic, rapid and reversible changes of behaviors (Bargmann, 2012; Park & Friston, 2013).

Hierarchical organization. Converging evidence in neuroscience indicates that the behavioral repertoire of living organisms is hierarchically organized and includes multiple levels of control, spanning from simple spinal reflexes, spinal cord and brainstem, up to sensory motor cortex and prefrontal cortex (Donnarumma, Prevete, Chersi, & Pezzulo, 2015b; Graziano, 2006; Hamilton & Grafton, 2007; Kelly, 1991). This hierarchical organization is widely believed to support the flexible selection, sequencing and recombination of the primitives (Flash & Hochner, 2005). For example, concerning the premotor area of monkey (and human) brains, they are (partially) organized in hierarchies, with multiple levels of representations (e.g. effector-dependent and effector-independent actions) (Fogassi, Ferrari, Chersi, Gesierich, Rozzi, & Rizzolatti, 2005). From a theoretical perspective, hierarchical control has been described in terms of (hierarchical) Bayesian systems and predictive coding (Friston, 2003; Haruno et al., 2003). In the Artificial Neural Network (ANN) literature, a somewhat simpler control scheme is usually adopted, in which hierarchies are implemented as two-level ANNs. Numerous studies have addressed the learning of action primitives (Hioki, Miyazaki, & Nishii, 2013; Paine & Tani, 2004; Tani, Nishimoto, & Paine, 2008; Yamauchi & Beer, 1994), the acquisition of multi-level control hierarchies for robot navigation (Chersi, Donnarumma, & Pezzulo, 2013; Tani, 2003; Tani, Nishimoto, & Paine, 2008; Tani & Nolfi, 1999) and the acquisition of sub-goals (Bakker & Schmidhuber, 2004; Dindo, Donnarumma, Chersi, & Pezzulo, 2015; Maisto, Donnarumma, & Pezzulo, 2015; Mcgovern & Barto, 2001; Mussa-Ivaldi & Bizzi, 2000; Thoroughman & Shadmehr, 2000).

Despite progress in understanding and defining the capabilities of human and artificial agents to learn and control complex skilled behaviors, still many aspects remain unclear, such as for instance how multiple motor primitives are encoded in the (same areas of the) brain, what neural substrate permits their learning without catastrophic forgetting, what the organizing principle of control hierarchies is, how cognitive control is exerted, or how parts of the brain can control other parts of the brain and permit to rapidly (i.e. without re-learning) change behavior and follow rule-like regularities. The novel hypothesis gaining ground on the neural realization of motor primitives envisages neural circuits capable of changing their behaviors rapidly and reversibly, thus without modifying their structure or modifying (re-learning) synaptic connectivity (Bargmann, 2012). Building on this, a control theory of ANN modules was developed that could express different dynamical behaviors by switching among them by means of a set of controlling input parameters (Donnarumma, Prevete, Chersi, & Pezzulo, 2015b; Donnarumma, Prevete, & Trautteur, 2010, 2012; Eliasmith, 2005; Eliasmith & Anderson, 2004; Montone, Donnarumma, & Prevete, 2011). In particular, in Donnarumma et al. (2015b, 2012) this type of control is interpreted in terms of the concept of programming as it is defined in the context of computer science.

In this paper, we take a computational perspective and propose a novel view on hierarchical organization and control in the brain including all three properties discussed previously. Our starting point is the approach proposed in Donnarumma et al. (2015b, 2012), on the basis of which we propose a Hierarchical Programmable Neural Network Architecture (HPNNA). In particular, we expected that learning and switching among behavior codes is a "simpler" task if compared with learning behavior dynamics as a whole. To this aim, here, we extensively test the learning ability of this architecture with respect to standard neural network approaches, and deeply investigate the possibility to obtain multiple programmable levels in a hierarchical fashion. HPNNA

is based on fixed-weight Continuous Time Recurrent Neural Networks (CTRNNs), which are plausible (though highly simplified) computational models of biological neuronal networks. In keeping with the neural evidence reviewed so far, we assume that multiple primitives could be encoded in the same neural structures, with higher neural levels exerting control over behavior by biasing the selection among these primitives. Compared to existing theoretical and computational proposals, our work elaborates on the concept of *programmability* of neural networks, which entails two novel proposals: a novel way to encode multiple motor primitives in multi-purpose and reusable neural networks, and a novel control scheme for exerting cognitive control. We test our approach in a Robotic scenario: a multiple T -maze with eight possible different goals. Firstly, our experimental scenario starts with a comparison of the learning capability of standard non-programmable approach versus the HPNNA in an idealization of eight different sub-tasks. Then we test the overall HPNNA, implementing on the lower layer an *interpreter* of motor primitives, receiving commands from a higher level interpreter layer capable of sequencing the low-level primitives in order to achieve the proper task. The proposed computational scheme is compared with a non-organized architecture (NOA), i.e. a neural network without a structure explicitly subsuming neither program nor a hierarchical organization, and results in a robust, scalable and flexible scheme able to successfully decompose the desired task. Because of these features, ours results in an appealing proposal to explain brain function and hierarchical control organization. Furthermore, this computational scheme provides many advantages from a learning perspective, including the possibility to learn novel primitives incrementally without disrupting the existing functionalities, to flexibly reassemble and off-line learning novel behavioral sequences using feedback signals generated by the existing motor primitives and to build modular networks (*interpreters*) splitting the task space into more manageable (learnable) parts.

2 Hierarchical programmable neural network architecture

Our proposed architecture takes as its starting point the *programmable neural network* (PNN) architecture introduced by Donnarumma et al. (2012). This neural model is endowed with a *programming* capability. The term programmability is not intended in a metaphorical sense but in a precise computational sense, as a generalization of the concept of programming to dynamical systems (Trautteur & Tamburrini, 2007). Following this work, a system can be considered endowed with programmability if three conditions are satisfied:

(a) there exists an effective encoding of the structure of the single systems into patterns of input, output, and internal variables;
(b) the codes provided by such encoding can be applied to specific systems of the class, interpreters, realizing the behavior of the coded system;
(c) the codes can be processed by the systems of the class on a par with the input, output, and internal variables.

Ensuring those requirements, a PNN realizes a virtual machine resulting in an interpreter of a finite set of neural networks, or in other words it can simulate a well-defined (finite) set of neural networks. In other words, a PNN realizes a neural sub-system *fully* controllable (*programmable*) behavior without changing connectivity and efficacies associated with the synaptic connections. Moreover, a distributed representation scheme is also ensured, as multiple motor primitives can be embedded in the same (*fixed-structured*) neural population.

At a computational level, this can be achieved by the presence of multiplicative sub-networks that enable a first (programmer) network to provide input values to a second (interpreter) network through auxiliary input lines. More in detail, the dynamic behavior of an artificial neural network can be defined as an output y_i based on the sums of the products between connection weights w_{ij} and neuron output signals x_j

$$y_i = f\left(\sum_j w_{ij} \cdot x_j\right)$$

From a mathematical point of view, one can "pull out" the multiplication operation $w_{ij} \cdot x_j$ by means of a *multiplication* (*mul*) sub-network that can compute the result of the multiplication between the output and the weight, inputs to the *mul* sub-network

$$y_i = f\left(\sum_j mul\left(w_{ij}, x_j\right)\right)$$

This procedure (called *w-substitution* in Donnarumma et al. (2012)) is at the basis of the construction of a PNN with a line of auxiliary inputs capable of modulating its behavior "as if" the synaptic efficacies were varied (see Figure 1). As a consequence, a PNN gets the results of receiving two kinds of input lines: *auxiliary* (or *programming*) input lines and *standard* data input lines. The newly introduced programming inputs are meant to be fed with a code, or program, describing the network to be "simulated".

In principle, a PNN architecture can be implemented using several kinds of recurrent neural networks. Here we introduce an implementation of PNN using Continuous Time Recurrent Neural Networks

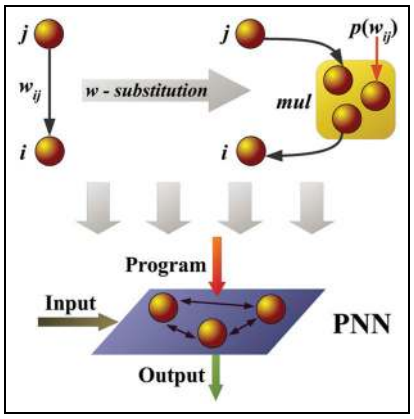


Figure 1. The "pulling out" of the multiplication (on top) performed by means of the *w-substitution* procedure. Distinct *mul* networks "break" weights in order to effectively implement a PNN that acts as an interpreter of neural programs.

(CTRNNs), which are generally considered to be biologically plausible networks of neurons and are described by equation (1) (Beer, 1995; Hopfield & Tank, 1986)

$$\tau_i \frac{dy_i}{dt} = -y_i + \sigma\left(\sum_{j=1}^{N} w_{ij} y_j + \theta_i + I_i^e(t)\right) \quad (1)$$

where $i \in \{1, \ldots, N\}$ and N is the total number of the neurons in the network. Thus, for each neuron i:

- τ_i is the *membrane time constant*;
- y_i is the *mean firing rate*;
- θ_i is the *threshold* (or bias);
- $\sigma(x)$ is the standard logistic activation function, i.e. $\sigma(x) = \frac{1}{1+e^{-x}}$;
- $I_i^e = \sum_{j=1}^{Q} w_{i,j+N} \cdot x_j$ is a weighted *external input current* coming from Q external sources x_j;
- w_{ij} is the *synaptic efficacy* (*weight*) of the connection coming from the neuron j or external sources x_j to the neuron i.

The equation (1) has a solution of $\mathbf{y}(t) = (y_1(t), \ldots, y_N(t))$ describing the dynamics of the neurons of the network. In general such a solution cannot be found exactly and an approximation of it can be computed by numerical integration.

Following Donnarumma et al. (2015b, 2012) it is possible to build a PNN, in the CTRNN framework, that is able to simulate (behaving like an *interpreter*) the behavior of the encoded CTRNN networks on the data coming from the standard input lines when varying the *programming* input line.

To this aim, the first step is to build a *mul* network in the CTRNN framework. A *mul* can be written as

$$mul \equiv \left\{ \theta_m \dot{\mu}_m = mul(a,b) = -\mu_m + \sigma\left(C_m a + \sum_{j=1}^{M} C'_{mj}\mu_j + C''_m b\right). \right. \quad (2)$$

with $m \in \{1, \ldots, M\}$. Equation (2) describes a network of M neurons receiving two inputs, a and b; the m-th neuron has time constant $theta_m$, mean firing rate μ_m, C'_{mj} is the weight value of the connection coming from the j-th neuron. C_m and C''_m weight the input a and b, respectively. The connection of the *mul* network, C_m, C'_{mj} and C''_m are tuned in order that solutions of equation (2), $\boldsymbol{\mu}(t) = (\mu(t), \ldots, \mu_M(t))$ are constrained to satisfy

$$\lim_{t\to\infty} \mu_M(t) \approx a \cdot b \quad (3)$$

$\forall a, b \in (0,1)$. In Donnarumma et al. (2012) an approximated *mul* solution is found for $M = 3$, (the same that we used in Section 3) by means of an evolutionary approach. Note that larger M values, while improving the behavior of *mul* networks, on the other hand increase the cost of computational simulation. By means of *mul* networks it is possible to construct a hierarchy of PNN layers with the higher level programming the lower in an increasing complexity of programs. In a first approximation, each layer is composed of the interaction of slower neurons (with higher time constants) whose activity is denoted with y_n, and faster neurons (with smaller time constants) with activity μ_m, belonging to the *mul* networks. To show how this is achieved, we first construct a layer $L_l(1, 1)$, i.e. with one slow neuron and one input source. It is described by the following system

$$L_l(1,1) \equiv \begin{cases} \tau_1 \dot{y}_1^{(l)} = -y_1^{(l)} + \sigma\left(\hat{w} \cdot \left(\mu^1_{M^{(l)}} + \mu^2_{M^{(l)}}\right) + \tilde{w} \cdot \left(y_1^{(l)} + x_1^{(l)}\right)\right) & \\ \theta^1_m \dot{\mu}^1_m = mul\left(y_1^{(l)}, y_1^{(l+1)}\right) & m \in \{1, \ldots, M^{(l)}\} \\ \theta^2_m \dot{\mu}^2_m = mul\left(x_1^{(l)}, y_2^{(l+1)}\right) & m \in \{1, \ldots, M^{(l)}\} \end{cases} \quad (4)$$

A depiction of the system of equations (4) is given in Figure 2. The Layer is a PNN composed of a *slow* neuron of activity $y_1^{(l)}$ and receiving an input $x_1^{(l)}$ and two programming inputs $y_1^{(l+1)}$ and $y_2^{(l+1)}$ from the higher level (the superscript indicates the belonging to l-th level). The first *mul* network $mul_1^{(l)} = mul(y_1^{(l)}, y_1^{(l+1)})$ connects $y_1^{(l)}$ with the programming input $y_1^{(l+1)}$ coming from the layer $l+1$. In the same way, the second *mul* network $mul_2^{(l)} = mul(x_1^{(l)}, y_2^{(l+1)})$ modulates by means

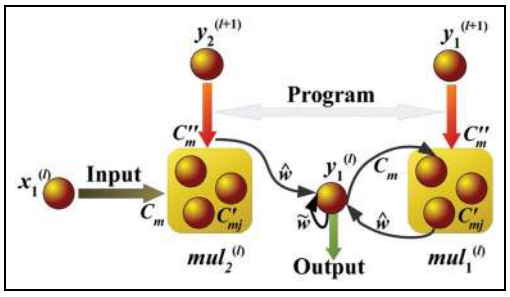


Figure 2. Depiction of a Layer $L_l(1, 1)$ described in the system of equations (4). The Layer is a PNN composed of one *slow* neuron y_1^l and two *mul* networks. It receives an input $x_1^{(l)}$ and two programming inputs $y_1^{(l+1)}$ and $y_2^{(l+1)}$ from the higher level. In particular, the network $mul_1^{(l)} = mul(y_1^{(l)}, y_1^{(l+1)})$ connects $y_1^{(l)}$ with the programming input $y_1^{(l+1)}$ and the network $mul_2^{(l)} = mul(x_1^{(l)}, y_2^{(l+1)})$ connects $y_1^{(l)}$ with the input source $x_1^{(l)}$.

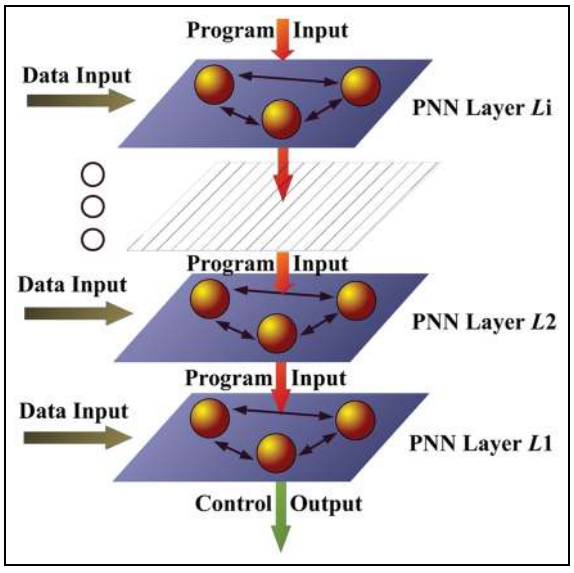


Figure 3. Hierarchical multi-layer architecture of the *programmable neural network* (PNN) architecture. In the proposed scheme, the upper layers send programs to the lower ones that act, in their turn, as a neural interpreter.

of the programming input $y_1^{(l)}$ contribute of the input source $x_1^{(l)}$ to $y_1^{(l)}$ activation. $\hat{w}$ weights the connection from neurons μ_M^1 of the $mul_1^{(l)}$ networks to the slow neuron $y_1^{(l)}$, and $\tilde{w}$ weights the self-connection of $y_1^{(l)}$.

Equation (4) can be straightforwardly generalized to accomplish $N^{(l)}$ slow neurons and $Q^{(l)}$ input sources

$$L_l(N^{(l)}, Q^{(l)}) \equiv \begin{cases} \tau_n \dot{y}_n^{(l)} = -y_n^{(l)} + \sigma\left(\hat{w} \sum_{h=T^{(l)}\cdot(n-1)+1}^{T^{(l)}\cdot n+1} \mu_{M^{(l)}}^h + \tilde{w} \sum_{j=1}^{T^{(l)}} y_j^{(l)}\right) & n \in \{1, \ldots, N^{(l)}\} \\ \theta_m^k \dot{\mu}_m^k = mul\left(y_k^{(l)}, y_k^{(l+1)}\right) & m \in \{1, \ldots, M^{(l)}\}, k \in \{1, \ldots, N^{(l)}(l)\} \end{cases} \quad (5)$$

where

- we set $y_{N^{(l)}+j} = x_j$ and $T^{(l)} = N^{(l)} + Q^{(l)}$;
- μ_m^k is the activation of the m-th (fast) neuron of the k-th *mul* network;
- $y_n^{(l)}$ is the activation of the n-th slow neuron;
- the condition on the time constants $\theta_m^k \ll \min\{\tau_n\}$ is imposed in order that the *mul* networks have faster dynamics with respect to the slow neurons $y_n^{(l)}$;
- $\hat{w}$ weights the connections from neurons μ_M^k the *mul* networks to the slow neurons $y_n^{(l)}$;
- $\tilde{w}$ weights the connections among slow neurons $y_n^{(l)}$.

The output of a PNN can be redirected as an input of a new PNN, i.e. this computational scheme can easily be iterated at multiple hierarchical layers, with the result that a network playing the role of *programmer* relative to a lower-level *interpreter* can also play the role of an *interpreter* relative to a higher-level *programmer*, providing a homogeneous hierarchical organizing principle that extends over an indefinite number of layers (see Figure 3).

Notice that for an ideal *mul*, the solution of $L_l(N^{(l)}, Q^{(l)})$ restricted to the slow neurons

$$\mathbf{y}^{(l)}(t) = \left(y_n^{(l)}(t; \mathbf{x}^{(l)}, \mathbf{y}^{(l+1)}), \ldots, y_{N^{(l)}}^{(l)}(t; \mathbf{x}^{(l)}, \mathbf{y}^{(l+1)})\right) \quad (6)$$

when varying the programming inputs $y_j^{(l+1)}$, can approximate solutions $\mathbf{y}(t) = (\{y_n(t; W, \{x_i\}_{i=1}^{Q}\cdot\}_{n=1}^{N})$ of an "ordinary" CTRNN of equation (1) when $N^{(l)} = N$, $x_i = x^{(l+1)}$. Or more formally, $\forall \epsilon > 0, \forall t$

$$\mathbf{y}(t) = (y_1(t; \mathbf{x}, W), \ldots, y_N(t; \mathbf{x}, W)) \quad (7)$$

In other words, a PNN system performs a substitution of variables which lets the programming inputs vary the system in the same way the changing of weights does in an "ordinary" CTRNN. When an approximated *mul* is given, however, it is a difficult task to formally establish an ε bound for large networks (Donnarumma, Murano, & Prevete, 2015a), thus the fine-tuning of the system relies on experimental considerations and can be improved in a way to satisfy Condition (3): (a) by increasing the "speed" of the *mul* networks tuning the setting of time constants in order to improve the approximation $\theta_m^k \ll \min\{\tau_n\}$ and (b) by refining its output response increasing the size M of the *mul* networks.

Finally, we stress that in our modelization, all the connections of the layers are *fixed* connections and thus, the dynamic behaviors they exhibit are completely due only to the change of their input, i.e. data input $x_i^{(l)}$ and *programs* from the upper layer determined by the activation $y_j^{(l+1)}$. In other words, the layer L_{l+1} may

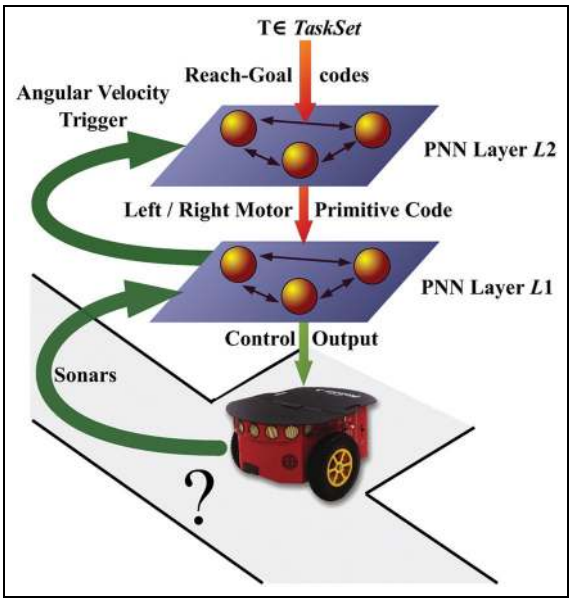


Figure 4. The hierarchical programmable neural network architecture built for the robotic scenario. Two layers of interpreters (PNNs) are present: a (higher) level L_2 and a (lower) level L_1. L_2 receives *reach-goal* codes on the programming input lines and trigger signals on the data input lines. L_1 receives *motor-primitive* codes, from L_2, on the program input lines and sensor data on the data input lines and outputs the control signals which govern the robot in the environment.

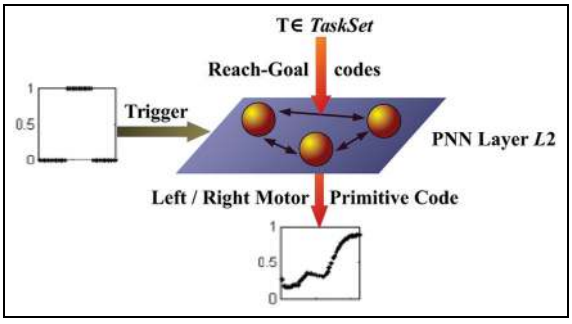


Figure 5. General presentation of sought L_2 module. Its output controls the activation of the proper motor primitive P_l or P_r. It has two inputs, a reach-goal code on which the coded task is presented, and a Trigger input, carrying the information on when the proper primitive should be enacted.

eventually fall into attractor states "readable" on its neurons $y_i^{(l+1)}$. These values form the programming inputs sent to the layer L_l, which consequently rapidly changes its behavior dynamics. This means that the changes of behavior we model are qualitatively different from learning, because they do not involve synaptic weight changes. Moreover they are *reversible*, because previous enacted dynamic behaviors can be elicited whenever suitable programming inputs to the layer are sent.

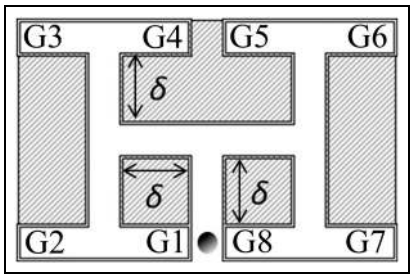


Figure 6. The multiple *T*-maze scenario used in the experiments. The starting position of the agent is indicated by a gray circle at the bottom of the maze. Each corridor of the maze has the same length δ. G_1, …, G_8 are the eight possible goal positions corresponding to the given task T_1,…,T_8.

3 Experiments

We present a Hierarchical Programming Neural Network Architecture (HPNNA) built for a robotic scenario. We considered an agent learning eight different tasks corresponding to eight different goals in a T -maze environment (see Figure 6), starting from a fixed start location. The HPNNA is composed of two *interpreters* of networks (PNNs): a (higher) level L_2 and a (lower) level L_1 (see Figure 4).

L_2 receives *reach-goal* codes on the programming input lines T and *trigger* inputs on the data input lines from L_1. L_1 receives *motor-primitive* codes, from L_2, on the programming input lines and sensor data on the data input lines and outputs the control signals that govern the agent in the environment. L_1 can be programmed to implement different motor primitives when its programming input lines are fed with suitable codes. The trigger signal encodes the completion of a motor primitive. L_2 can be programmed to implement different sequences of motor-primitive codes when its programming input lines are fed with suitable codes.

The programming input learning is achieved by a two-step learning strategy which can be described as follows;

1. In the first step we sought 2^3 reach-goal codes for the programming input lines of L_2. The learning ensures that when L_2 is fed with one of these codes, the agent is able to perform a sequence of three consecutive motor primitive programs constituting the reach-goal program. The switch between the motor primitive programs occurs when L_2 detects a T-intersection by means of the trigger signal coming from L_1.
2. In the second step we sought two different lower level programs, *Right-Wall Follower* (P_r) and *Left-Wall Follower* (P_l), which encode the basic motor primitives of our control architecture. The agent exhibits two different behaviors in the environment according to two different codes P_r and P_l. When L_1

is fed with the programming input P_r or P_l the robot follows the wall to its right or its left, respectively.

In Subsection 3.1 we show the first learning step, preparing a synthetic dataset, in which the stimuli and the programs are simplified in order to study the different learning properties of the proposed architecture versus a non-programmable one. The second step is presented in Subsection 3.2 where the primitives are actually learned in a simulated robotic environment and then the overall architecture is tested in the multiple T-maze simulated robotic scenario.

3.1 Learning motor primitives composition - HPNNA versus NOA

The task of this section is the learning phase of the L_2 PNN module. The aim of the learning is to endow L_2 with the capability of driving the agent in a multiple T-maze environment by sequencing specific motor primitives (see Figure 5).

Following from Yamauchi and Beer (1994), we sought a network capable of changing its state when an external trigger is given. Let us suppose we have a network that selects the two programs P_r and P_l by means of the output of one of its neurons. A high value of this neuron selects the program P_r while a low value selects the program P_l.

In order to test how able the proposed architecture is to learn different programs, in this section we prepare a synthetic dataset in order to capture the stylized different tasks in a multiple T-maze code. We compare our HPNNA versus a traditional non-organized architecture (NOA, see below) showing how *learning multiple behaviors* is computationally more difficult with respect to *learning multiple behavior codes*. In this experimental scenario we imagined an agent exploring a multiple T-maze (see Figure 6). In the considered mazes each corridor has the same length δ. This δ parameter is an environment variable we varied during the experiments.

Accordingly with our strategy, the control module of HPNNA, sequencing primitives, L_2 has two kinds of input lines:

- the data input line is fed with the external trigger;
- the programming input line encodes the different sequences that constitute our high level program.

Thus, given the fixed structure interpreter L_2, we learn the structure of a neural network memorizing the input codes to be sent to L_2, testing our HPNNA approach. As a comparison a similar learning is performed on a CTRNN layer that it is not structured as in equation (5) but follows the ordinary CTRNN equation (1); we refer to this module as a *non-organized architecture* (NOA).

By means of layer L_2, the agent is supposed to control two different low-level primitives:

- *Left-Wall Follower* denoted with P_l, i.e. the behavior "follow the wall on the left";
- *Right-Wall Follower* denoted with P_r, i.e. the behavior "follow the wall on the right".

We assume to learn a control module of an agent, with two inputs, a task-input T and a trigger-input I_D, plus a motor-output U calling the two primitives, P_l or P_r. The agent is supposed to move inside the maze perceiving it with sensors able to detect walls. When it reaches a T-cross, the trigger-input is activated and the agent consequently moves in order to regain the wall performing respectively a left-turn or a right-turn depending on the input task the agent receives.

Each of the eight tasks T_1, …, T_8 corresponds to the successful reaching of one of the goals G_1, …, G_8 in the maze (see Figure 7). Each task of the agent can be decomposed into a sequence of three low-level primitives $[P_1P_2P_3]$ with $P_i = P_r$ or P_l. Each sequence is recalled by the corresponding task-input T sent to L_2, i.e.

$$
\begin{array}{llllll}
[& P_l & P_l & P_l &] \rightarrow & T_1 = [0 \;\; 0 \;\; 0] \\
[& P_l & P_l & P_r &] \rightarrow & T_2 = [0 \;\; 0 \;\; 1] \\
[& P_l & P_r & P_l &] \rightarrow & T_3 = [0 \;\; 1 \;\; 0] \\
[& P_l & P_r & P_r &] \rightarrow & T_4 = [0 \;\; 1 \;\; 1] \\
[& P_r & P_l & P_l &] \rightarrow & T_5 = [1 \;\; 0 \;\; 0] \\
[& P_r & P_l & P_r &] \rightarrow & T_6 = [1 \;\; 0 \;\; 1] \\
[& P_r & P_r & P_l &] \rightarrow & T_7 = [1 \;\; 1 \;\; 0] \\
[& P_r & P_r & P_r &] \rightarrow & T_8 = [1 \;\; 1 \;\; 1]
\end{array}
$$

Therefore each program corresponds to a high level representation of the possible agent's behaviors. Ideally, at the end of the learning phase, by selecting a task-input T_i, the agent is asked to assume the behavior that allows it to reach the corresponding goal G_i in the maze. It is important to stress that this program formalization does not point at any specific trajectory, but at a *sequencing* of low-level primitives.

In this test, the trigger-input $I_D(t) \in \{0, 1\}$ is the idealization of a time varying input signal: it is high ($I_D = 1$) when the agent is turning (i.e. the agent is at the end of the corridor) and low ($I_D = 0$) when the agent moves forward along the corridors of the maze. In other words, the trigger tells the controlling unit when the robot turns left or right and, therefore, when it is necessary to select the next primitive from the program sequence "stored" in T_i.

In this first experiment we assume the agent moves at constant velocity v_A. Consequently, the duration ΔT_{low} of the low trigger-input can be considered proportional to the length δ of any of the corridors, while the duration ΔT_{high} of the high trigger-input is considered proportional to the time spent in the turning at

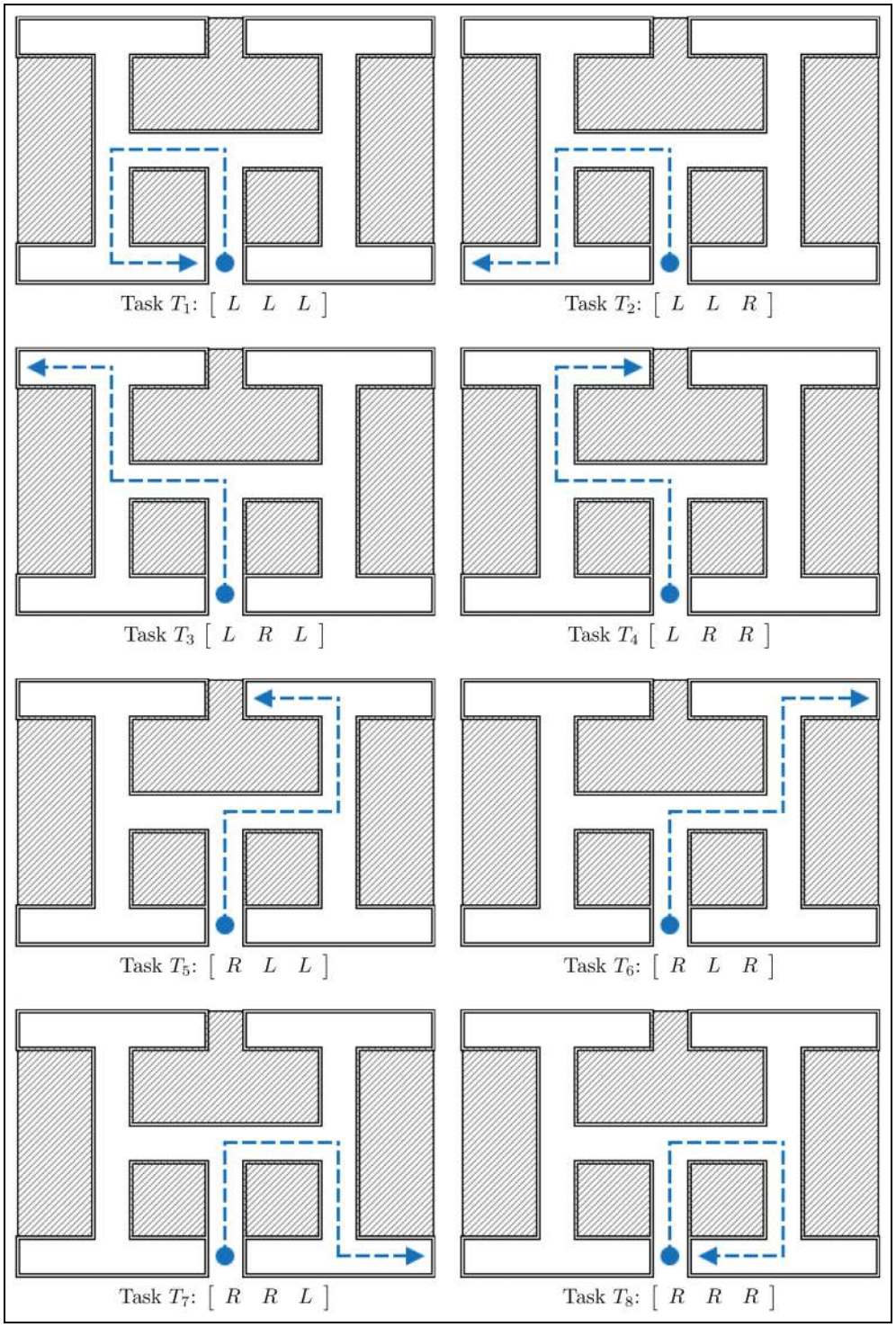


Figure 7. A depiction of the eight goal-reach tasks T_1, …, T_8 defined for the experimental scenario. They are composed of a series of three motor primitives and correspond to the reaching of the corresponding goals G_1, …, G_8 in the maze.

each T-cross and is assumed constant across maze dimensions.

Given these conditions, it is possible to define a parameter λ

$$\lambda = \frac{\Delta T_{low}}{\Delta T_{high}} = \frac{\delta}{v_A \cdot \Delta T_{high}} \Rightarrow \lambda \propto \delta \qquad (8)$$

which corresponds to the size of a chosen labyrinth, with respect to the trigger inputs to the controlling unit.

We build two different synthetic datasets, D_1 and D_2:

- D_1represents a single multiple T-maze with $\lambda = 2$, i.e. a maze with a single δ length of the corridor;

- D_2 represents three different multiple T-mazes, with $\lambda = 2$, $\lambda = 3$ and $\lambda = 4$, respectively, i.e. three mazes with three different δ lengths of the corridor.

By means of the parameter λ this information is implicitly stored in the trigger-input signal. The corresponding Target output $O(t)$ is consequently created in order to create datasets of input-output couples. The aim of learning is to replicate this target on the network output $U(t)$ at each time step. A number of ten target sequences for programs has been created for each maze. A reference sequence is about 40 time steps $t = (1/5) \cdot \tau$, where τ is the time constant unit used for the neural network modules. This means that dataset D_1 has 80 sample input-target sequences, while D_2 has 240 sequences.

3.1.1 Learning a control module by differential evolution. We adopt a learning algorithm based on an evolutionary approach, Differential Evolution (DE) Algorithm (De Falco, Della Cioppa, Donnarumma, Maisto, Prevete, & Tarantino, 2008; Price, Storn, & Lampinen, 2005). DE is an evolutionary population based algorithm proved to be very efficient in the continuous domain, fitting the case of learning of parameters of neural networks (De Falco et al., 2008). DE addresses a generic optimization problem with m real parameters by starting with a randomly initialized population consisting of n individuals, each made up of m real values. The population is updated from one generation to the next by means of many different transformation schemes commonly named as strategies (Price et al., 2005). In all of these strategies DE generates new individuals by adding to an individual a number of weighted difference vectors between couples of population individuals.

In this experiment the HPNNA learning is assigned an architecture based on equation (5), which is a fixed structure neural network (no synaptic connections are learned for this module), and an input module, that is a neural network that has to "memorize" the different codes allowing the different task. The aim of the learning is to find the programming inputs able to let HPNNA solve the task. The NOA architecture is a full-connected CTRNN of equation (1), without any particular internal structure. In this case, the aim of the learning is to find suitable CTRNN weights able to let NOA solve the presented task. To keep the comparison fair, we keep a similar number of parameters during the experiments. Thus, for both the compared architectures, DE performs a search for solutions in a parameter space $S \subseteq \mathbb{R}^{24}$ (see Table 1).

The learning procedure is described in detail in Algorithm 1. There is an outer loop which iterates the procedure for I_{max}. Each program is evaluated separately with a fitness function which is proportional to the *distance* between the target output O_k of the corresponding program selected by the task input T_k, from the motor output of the control module $U_k(t) = U(T_k(t), I_D(t))$. Thus the fitness value is computed on the set of sequences relative to the program T_k

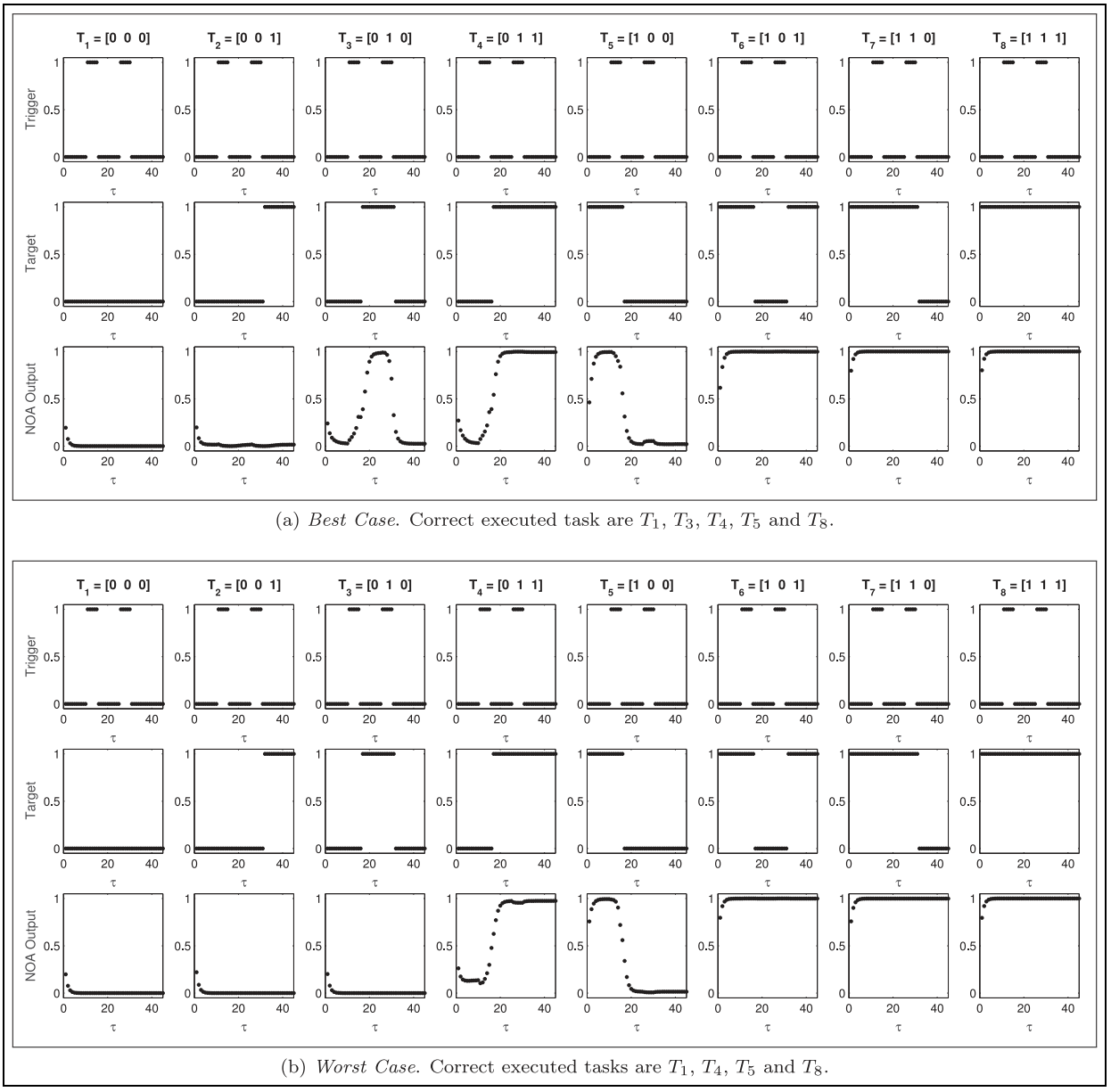
$$F_k = \sum_{T_k} \sum_{t} (U_k(t) - O_k(t))^2 \tag{9}$$

The function selectSamples is a function that selects sequences corresponding to which a program is going to be evaluated, allowing DE to move population towards a better solution of the parameter space. Samples related to programs that have been correctly learned can be excluded from learning. However, this option can be selected only for HPNNA architecture, because for NOA architecture, if this option is selected, the change in the synaptic weights would cause the well-known effect of catastrophic forgetting of previous learned behaviors, so that the learned module would at last collapse to learn only the last program selected.

The results of the tests are evaluated by comparing 20 learning-runs for each architecture, HPNNA and NOA. It is possible to see that HPNNA is able to achieve solutions that correctly perform eight out of eight programs. We show:

Algorithm 1 Control Module Learning $(\mathcal{D}, M, opt, \beta, I_{max})$.

Require: Dataset $\mathcal{D}$, Architecture Model M, optimum fitness threshold opt,
DE parameters β, Maximum number of total iterations I_{max}.
Ensure: Model Parameters $\boldsymbol{\theta}$
initialize Model Parameters $\boldsymbol{\theta}(M)$
set Fitness Values $F_k = +\infty$ for each Task $T_k \in TaskSet$
set iteration $i = 0$
while $F_k > opt$ for all Tasks $P_k \in TaskSet$ **and** $i < I_{max}$ **do**
  select samples to learn `selectSamples` $\mathcal{D}_T = (\mathcal{D})$
  execute Differential Evolution step DE $(\mathcal{D}_T, \beta)$
  update Architecture Model best parameters $\boldsymbol{\theta}$
  update Fitness Values F_k
  update iteration number i
end while



(a) *Best Case.* Correct executed task are T_1, T_3, T_4, T_5 and T_8.

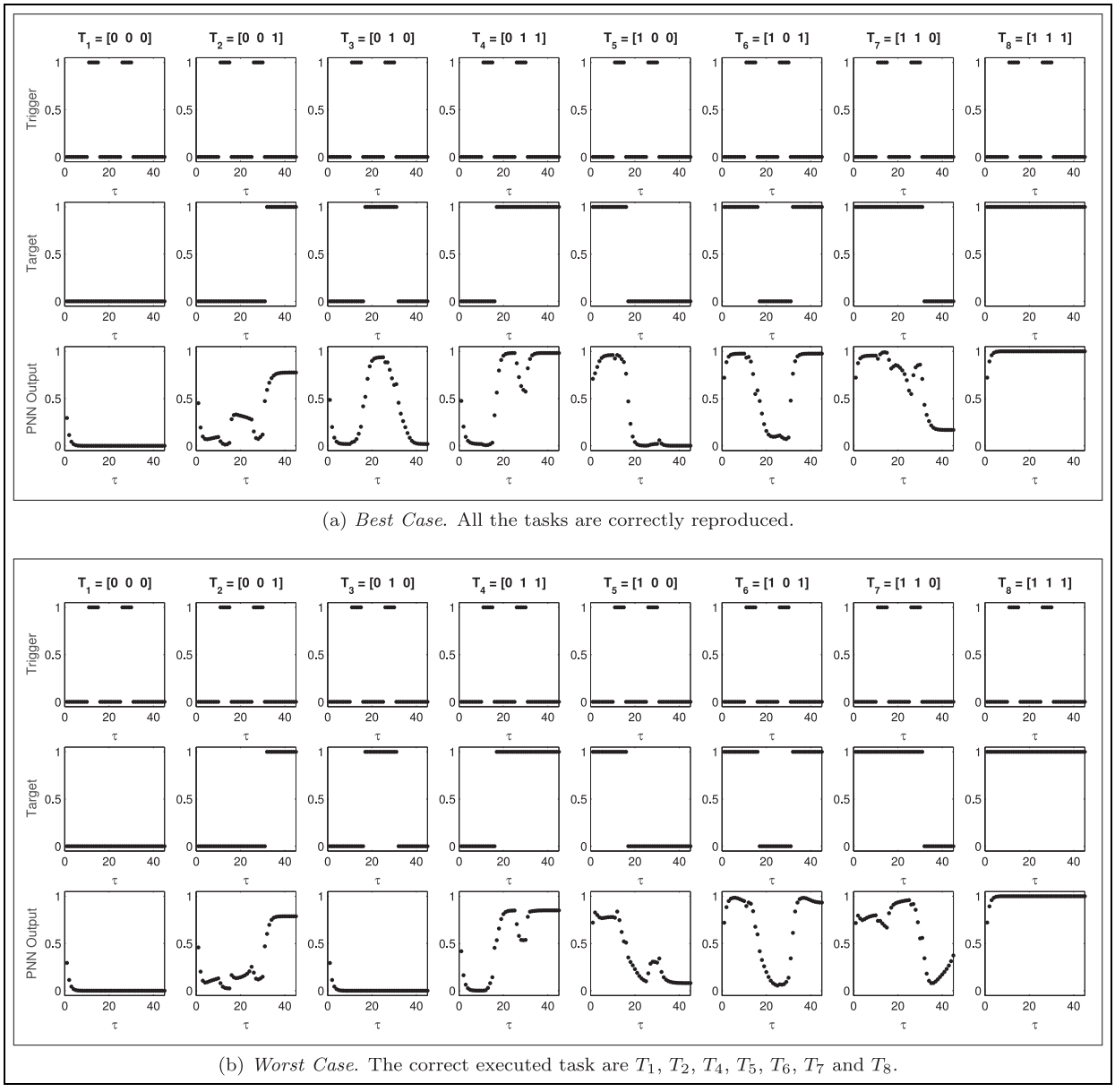
(b) *Worst Case.* Correct executed tasks are T_1, T_4, T_5 and T_8.

Figure 8. NOA sample outputs for Dataset $\mathcal{D}_1$.

- results in learning D_1 dataset, with samples from a maze with single δ length (see Paragraph 3.1.2);
- results in learning D_2 dataset, with samples from three mazes with three different δ lengths (see Paragraph 3.1.3);
- testing in maze of δ lengths different from the one seen during the learning phase (see Paragraph 3.1.4).

3.1.2 Learning D_1 dataset – single maze size. Table 2 and Table 3 detail the final fitness value for NOA and HPNNA. For dataset D_1 HPNNA is able to learn all programs in the 40% of learning-runs. In the remaining 60% of learning-runs, HPNNA learns at least seven out of eight programs. On the other hand NOA architecture is able to learn only a maximum of five out of eight programs (see Table 5). Notice that a NOA with the selectSamples catastrophically forgets previous programs and is able to learn only the last seen program (see Table 5). On the other hand, learning HPNNA is more computationally efficient if counting the number of total steps required while it is able to learn all programs with a smaller number of iterations (see Table 4). In Figures 8 and 9 sample executions for NOA and HPNNA are shown.

3.1.3 D_2 dataset – three different maze sizes. The dataset D_2 is built with sequences by different δ lengths, varied



(a) *Best Case.* All the tasks are correctly reproduced.

(b) *Worst Case.* The correct executed task are T_1, T_2, T_4, T_5, T_6, T_7 and T_8.

Figure 9. HPNNA sample outputs for Dataset $\mathcal{D}_1$.

by means of the parameter λ. The parameter values are $\lambda = 2$, $\lambda = 3$ and $\lambda = 4$. HPNNA was able to learn all eight programs (see Table 9), while NOA was not able to learn more than five programs. Table 4 also shows that a smaller number of iterations is needed to learn HPNNA. In Figures 13 to 15 sample HPNNA executions are shown while in Figures 10 to 12 sample NOA executions are shown. Overall the results speak about better performances for HPNNA capable of learning programs in different maze sizes.

3.1.4 Testing on unknown maze sizes. We test all the learned instances of the previous subsections in sequences subsuming mazes of never seen size. This is to verify the generalization capabilities of the architectures. We choose test mazes with $\lambda = 2.4$ and $\lambda = 3.6$ (see equation (8)). For both the architectures we test best and worst cases learned in datasets D_1 and D_2. Though, as expected, modules learned in dataset D_1 perform worse (HPNNA could not execute all programs (see Table 10)), modules learned in dataset D_2 generalize very well on new unseen dimensions (HPNNA can successfully execute all eight programs (see Table 10)).

3.2 HPNNA in a simulated robotic environment

In the previous section we made the hypothesis of having ideal motor primitives, in order to build a control module L_2 with suitable inputs to guide the agent

Table 1. Experimental parameters for the *Primitive Sequencing* learning task.

Network parameters β	Time constants	5τ
	Minimum weight value w_{min}	-5
	Maximum weight value w_{max}	$+5$
	Integration step Δt	0.2
	Fitness optimum value *opt*	0.99
	Maximum number of iterations I_{max}	20000
DE parameters θ	Population number	100
	Parameter space size	24
	Step size	0.7
	Cross − over probability	0.8
	Strategy	DE/RAND/1/BIN

Table 2. $\mathcal{D}_1$ Dataset results for NOA on 20 learning-runs. Fitness value F_k (mean, standard deviation, maximum and minimum) and success rate are shown for each Task T_k. The best NOA was not able to learn all programs. The low standard deviation suggests that similar results are expected if further runs were made.

	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8
Mean F_k	0.004	8.212	13.945	2.956	1.280	15.205	11.125	0.005
Standard deviation	0.001	5.919	3.739	2.447	0.971	0.225	5.375	0.003
Maximum	0.009	14.149	15.168	10.119	3.043	16.160	14.148	0.016
Minimum	0.003	1.028	2.050	0.028	0.022	15.153	1.110	0.003
Success rate	100%	55%	10%	95%	100%	0%	25%	100%

Table 3. $\mathcal{D}_1$ Dataset results for HPNNA on 20 learning-runs. Fitness value F_k (mean, standard deviation, maximum and minimum) and success rate are shown for each task T_k. The best HPNNA was able to execute eight out of eight programs. A high standard deviation in some cases underlines that the respective programs are more difficult to be learned than others (T_3 and T_7); this can also be noticed from the corresponding success rate values.

	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8
Mean F_k	0.005	0.080	8.210	0.100	0.038	0.058	0.773	0.005
Standard deviation	$< 10^{-3}$	0.014	7.379	0.217	0.019	0.013	3.147	$< 10^{-3}$
Maximum	0.005	0.100	15.155	1.020	0.092	0.088	14.145	0.005
Minimum	0.005	0.051	0.059	0.029	0.018	0.040	0.025	0.005
Success rate	100%	100%	45%	100%	100%	100%	95%	100%

Table 4. Iterations for HPNNA learning-runs in $\mathcal{D}_1$. The table shows mean, standard deviation, maximum and minimum of the number of iterations in the 20 runs. The avoiding of catastrophic forgetting effect allows to skip learning of programs for which F_k is less than the *opt* value. We stress that for NOA, it is always necessary to execute a number of iterations equal to I_{max} = 20 000.

HPNNA	Learning iterations per task								Total iterations
	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	
Mean	50	192	3760	390	72	780	860	50	6155
Standard deviation	3	78	1946	1087	25	199	1037	3	2269
Maximum	52	400	5000	5000	100	1100	5000	52	12 500
Minimum	48	100	600	50	50	450	250	49	2450

towards the desidered goal. In this section we actually implement a lower level interpreter L_1 of motor primitives (see Figure 16) in order to complete the HPNNA architecture and show its performance in a simulated robotic environment. Robot simulations were carried out using the open source software project **Player-Stage** (Gerkey, Vaughan, & Howard, 2003) to simulate a *Pioneer 3DX* robot (see Figure 17). The robot is

Table 5. Dataset $\mathcal{D}_1$ success percentage. The comparison summarizes successful results for the architectures HPNNA and NOA. In this table we show also the NOA learning results, when excluding samples from the dataset by the selectSamples procedure. In this case NOA meets the well-known *catastrophic forgetting* effect.

Learned programs	HPNNA (% on 20 runs)	NOA (% on 20 runs)	NOA *Catastrophic forgetting* (% on 20 runs)
1/8	100	100	100
2/8	100	100	0
3/8	100	100	0
4/8	100	100	0
5/8	100	85	0
6/8	100	0	0
7/8	100	0	0
8/8	40	0	0

Table 6. $\mathcal{D}_2$ Dataset results for NOA on 20 learning-runs. Fitness value F_k (mean, standard deviation, maximum and minimum) and success rate are shown for each Task T_k. The best NOA was not able to learn all programs. The low standard deviation suggests that similar results are expected if further runs were made.

	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8
Mean F_k	0.016	36.360	45.518	9.205	2.661	60.918	57.583	0.012
Standard deviation	0.013	23.197	21.472	5.500	2.975	0.934	0.004	0.002
Maximum	0.056	57.585	60.675	17.422	9.110	63.657	57.591	0.017
Maximum	0.010	2.484	6.158	0.087	0.076	60.610	57.580	0.010
Success rate	100%	35%	30%	95%	100%	0%	0%	100%

Table 7. $\mathcal{D}_2$ Dataset results for HPNNA on 20 learning-runs. Fitness value F_k (mean, standard deviation, maximum and minimum) and success rate are shown for each Task T_k. The best HPNNA was able to execute eight out of eight programs. A high standard deviation in some cases underlines that the respective programs are more difficult to be learned than others (T_2, T_3 and T_7); this can also be noticed from the corresponding success rate values.

	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8
Mean F_k	0.014	3.493	39.818	0.195	0.178	0.203	11.945	0.014
Standard deviation	$< 10^{-3}$	12.759	29.087	0.057	0.097	0.076	23.423	$< 10^{-3}$
Maximum	0.014	57.584	60.614	0.329	0.351	0.412	57.584	0.014
Minimum	0.014	0.177	0.277	0.093	0.055	0.125	0.102	0.014
Success rate	100%	95%	35%	100%	100%	100%	80%	100%

Table 8. Iterations for HPNNA learning-runs in $\mathcal{D}_2$. The table shows mean, standard deviation, maximum and minimum of the number of iterations in the 20 runs. The avoiding of catastrophic forgetting effect allows to skip learning of programs for which F_k is less than the *opt* value. We stress that for NOA, it is always necessary to execute a number of iterations equal to I_{max} = 20 000.

HPNNA	Learning iterations per task								Total iterations
	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	
Mean	50	693	2070	148	65	953	1218	50	5245
Standard deviation	4	825	769	73	24	352	864	3	1472
Maximum	52	2500	2500	400	100	1600	2500	53	7700
Minimum	47	150	550	50	50	450	150	48	2800

equipped with ten *sonars* placed on the frontal and the lateral parts of the robot ($s1$, …,$s10$ in Figure 17b.)

Note that L_1 governs the robot by setting its angular and linear velocity corresponding to the output of two neurons belonging to L_1. During this learning phase, the environment is a single T-maze consisting of corridors of fixed length and three times as wide as the robot size.

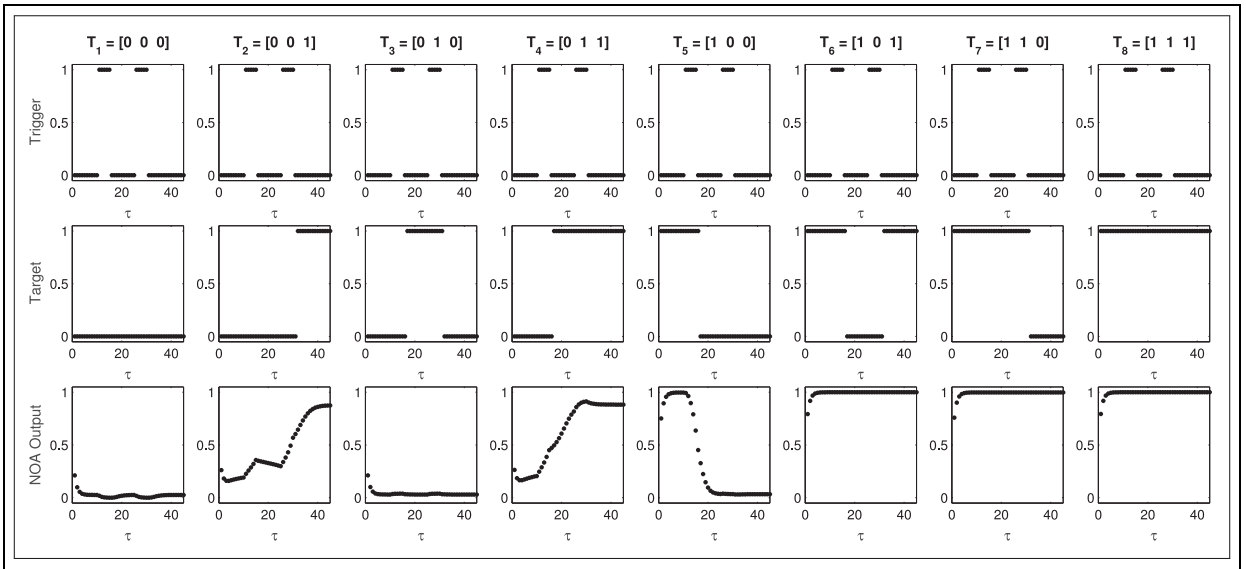
Table 9. Dataset $\mathcal{D}_2$ success percentage. The comparison summarizes successful results for the architectures HPNNA and NOA.

Learned programs	HPNNA (% on 20 runs)	NOA (% on 20 runs)
1/8	100	100
2/8	100	100
3/8	100	100
4/8	100	100
5/8	100	70
6/8	100	0
7/8	85	0
8/8	25	0

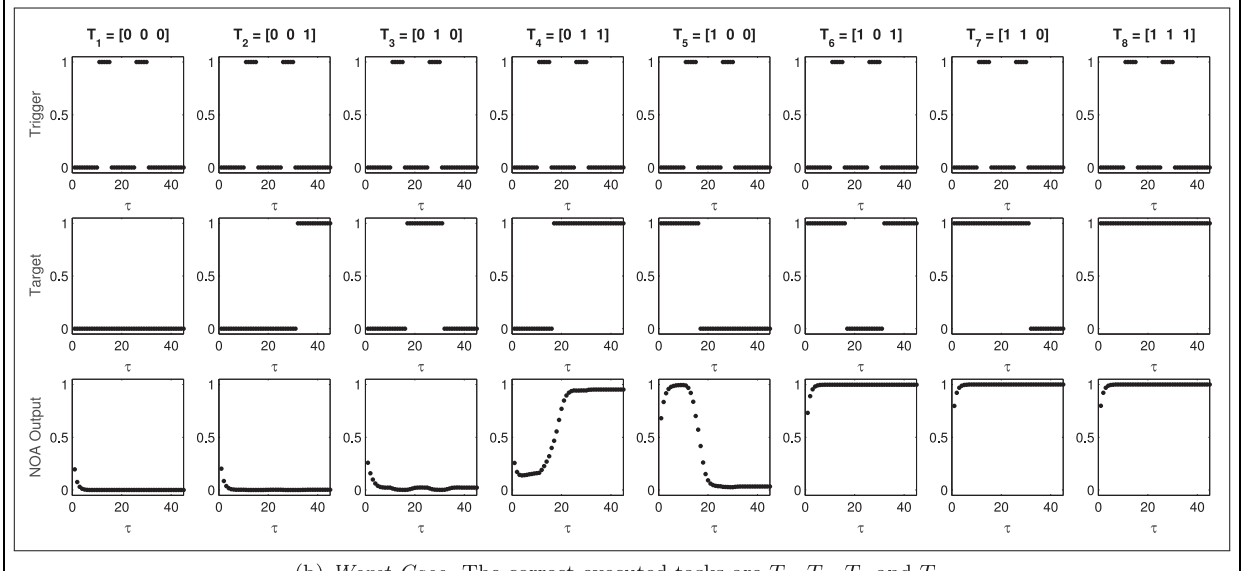
The L_1 module should realize an interpreter on which two motor-primitive programs are learned: Right-Wall Follower (P_r) and Left-Wall Follower (P_l). According to Section 2, this module has two kinds of input lines: a data input line and a programming input line.

The data input line consists of three inputs $\{I_1, I_2, I_3\}$ that are the weighted sum of three sonars facing right, the three in which basic motor-primitives are learned facing left and two frontal sonars, respectively, as in the following equations

$$\begin{aligned} I_1 &= 0.2 \cdot S_2 + 0.4 \cdot S_3 + 0.4 \cdot S_4 \\ I_2 &= 0.2 \cdot S_9 + 0.4 \cdot S_8 + 0.4 \cdot S_7 \\ I_3 &= 0.5 \cdot S_5 + 0.5 \cdot S_6 \end{aligned} \tag{10}$$

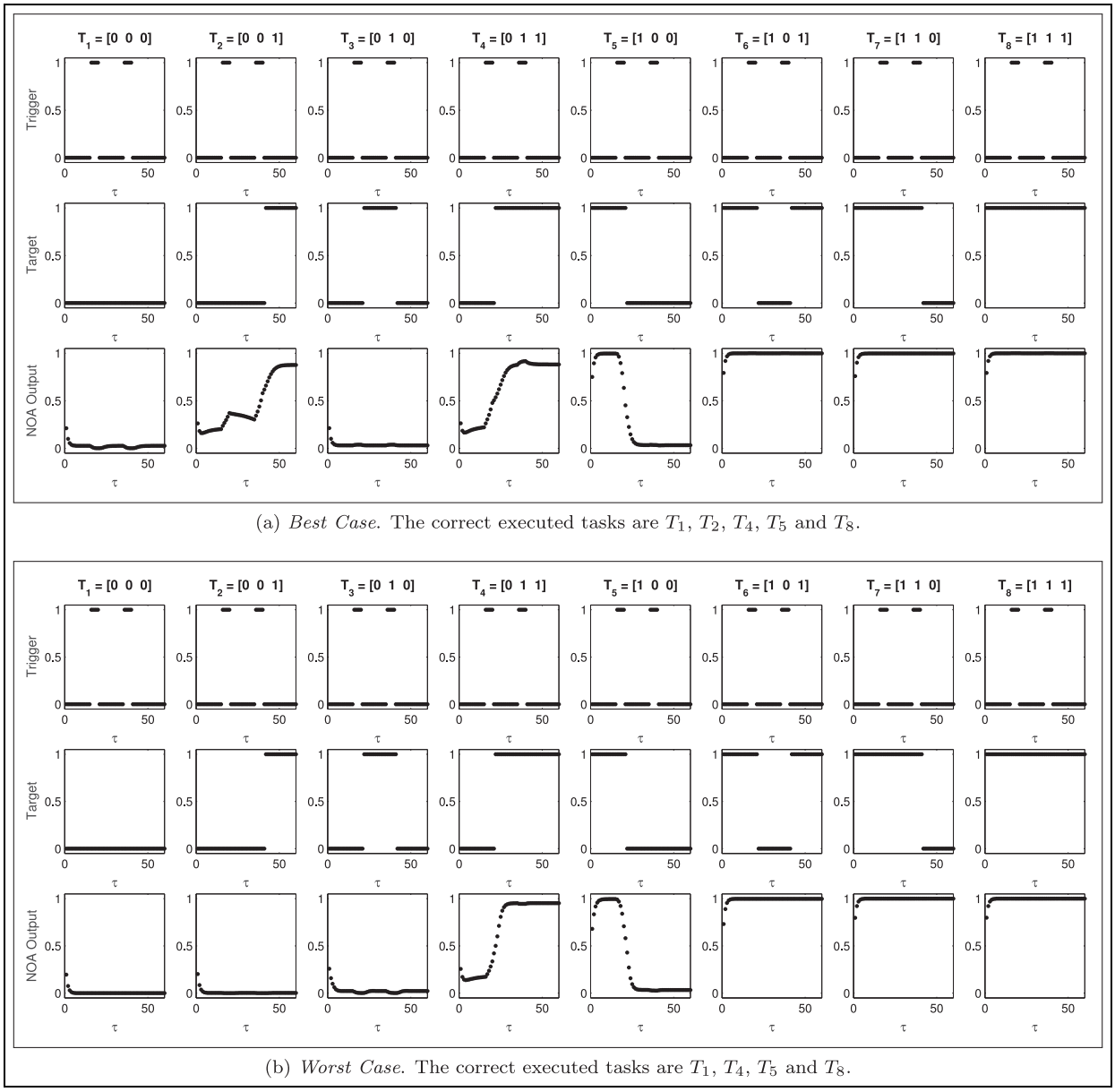


(a) *Best Case.* The correct executed tasks are T_1, T_2, T_4, T_5 and T_8.



(b) *Worst Case.* The correct executed tasks are T_1, T_4, T_5 and T_8.

Figure 10. NOA sample outputs for Dataset $\mathcal{D}_2$ and $\lambda = 2$.



(a) *Best Case.* The correct executed tasks are T_1, T_2, T_4, T_5 and T_8.

(b) *Worst Case.* The correct executed tasks are T_1, T_4, T_5 and T_8.

Figure 11. NOA sample outputs for Dataset $\mathcal{D}_2$ and $\lambda = 3$.

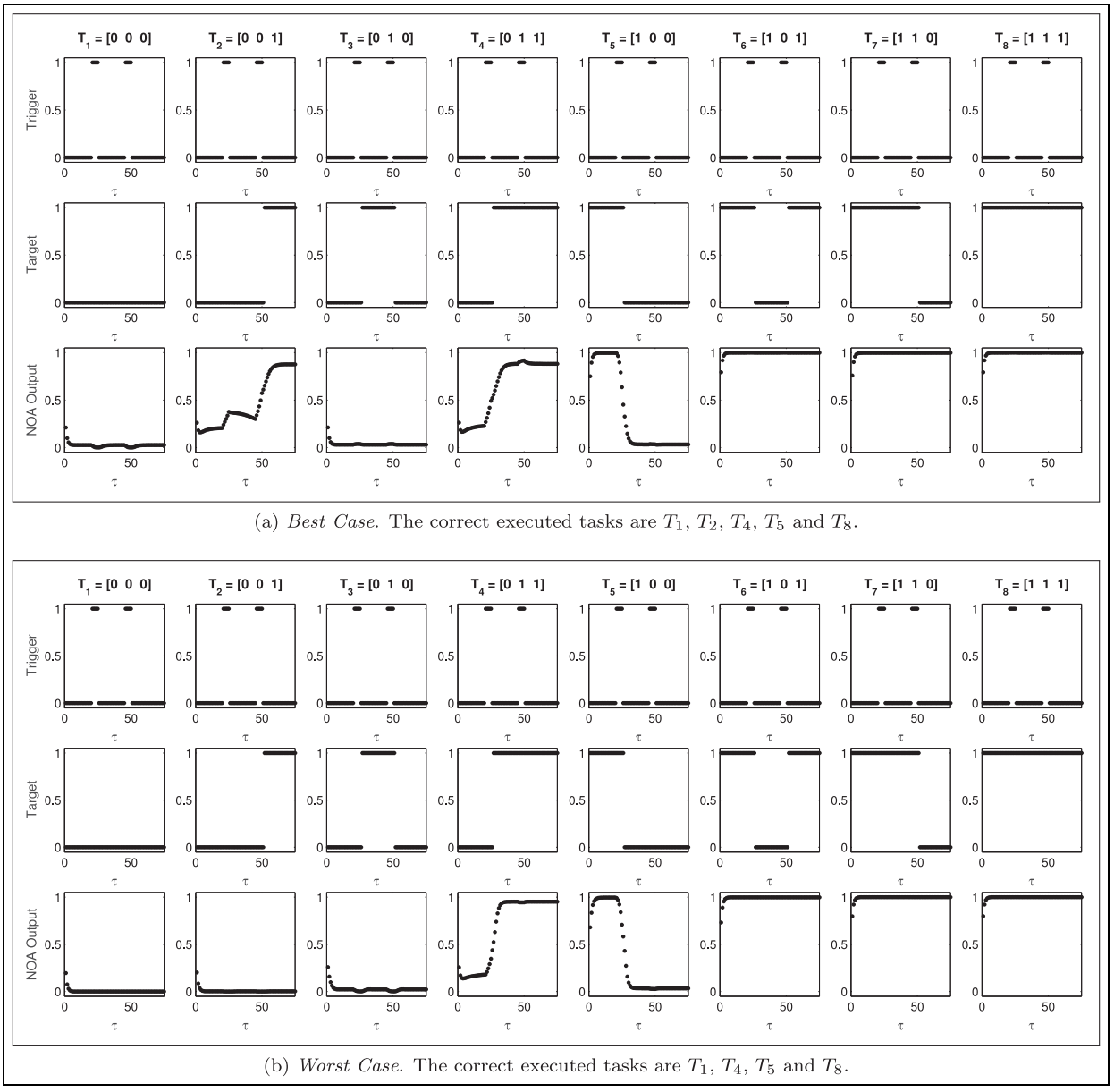
Thus, two neuron outputs of the L_1 module control the robot. In particular the activation of one neuron is devoted to the control of the linear speed of the robot, while another neuron controls the robot's angular velocity.

The neurons of the module share the same value of the characteristic time τ, that is of an order of magnitude bigger than the characteristic time of the multiplicative networks θ.

Then, by means of the w-substitution we construct the fixed structure interpreter L_1 made of 39 neurons with two kinds of input lines:

- a data input line that consists of the three inputs from the sonars;
- a programming input line that consists of inputs that codify the different structures simulated by the interpreter.

Consequently, we evolved a vector of 12 parameters in order to find the suitable *programs* able to let the network control the robot and perform the correct motor primitives. In our approach, given the fixed structure interpreter L_1, we used Algorithm 1 to learn the suitable motor-primitive codes. This is done by building suitable fitness functions, one for Right-Wall Follower P_r primitive, and a second one for the Left-Wall Follower P_l. Note that, in contrast with other approaches, it is possible to do this because the network structure is fixed and we do not evolve weights. Thus, we can divide the



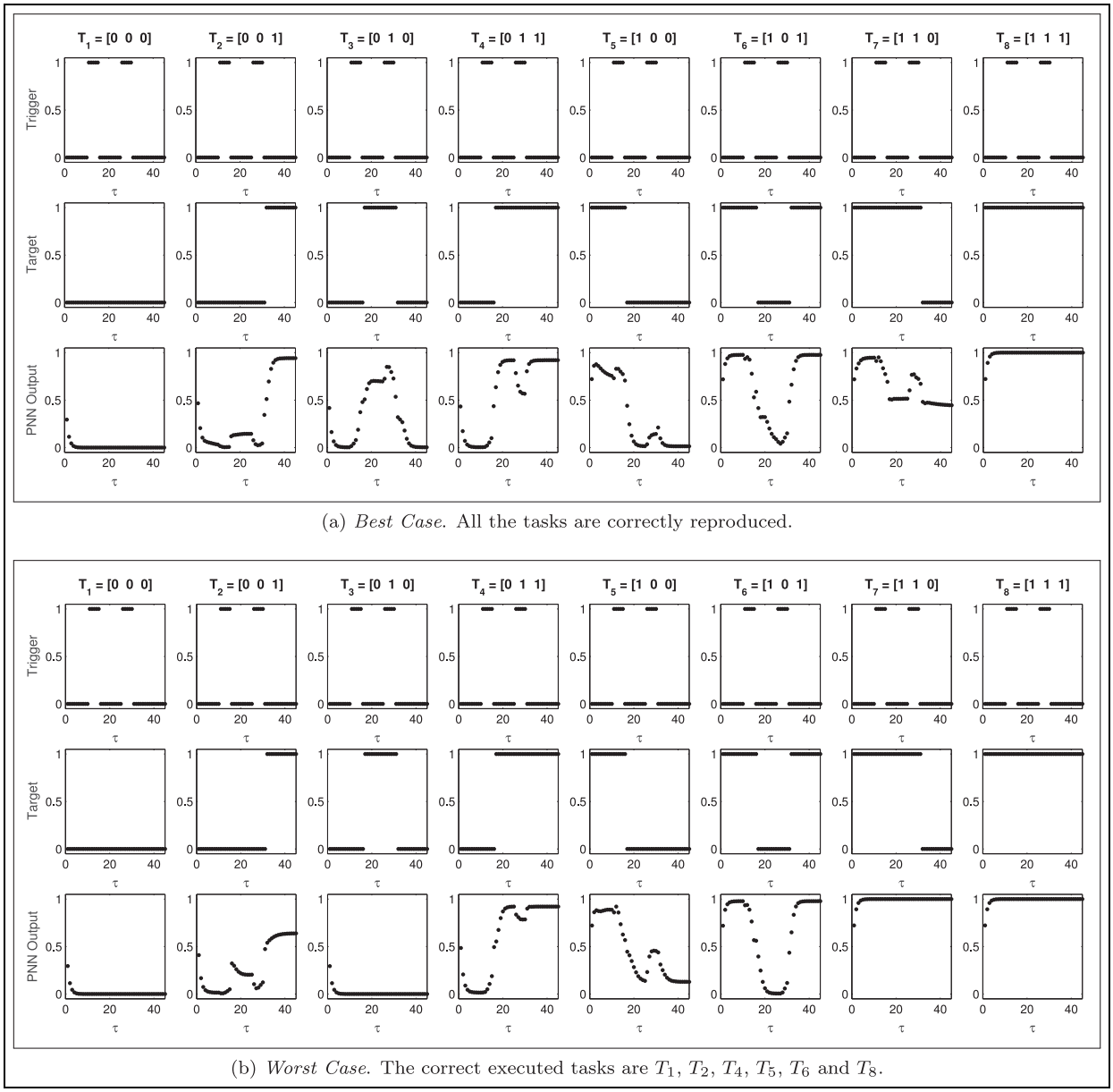
(a) *Best Case.* The correct executed tasks are T_1, T_2, T_4, T_5 and T_8.

(b) *Worst Case.* The correct executed tasks are T_1, T_4, T_5 and T_8.

Figure 12. NOA sample outputs for Dataset $\mathcal{D}_2$ and $\lambda = 4$.

learning into two epochs without erasing previously learned capabilities.

For each epoch we initialized a population of 20 elements controlled by networks with codes randomly chosen in the range $[-5, 5]$. Each controller obtained is evaluated with a fitness function specific for each program, i.e. F_R and F_L, while performing the task of behaving as a right or as a left follower, respectively. A new population is obtained using the best element of the previous population. In our training we used a crossover coefficient ($CR \in [0, 1]$) of 0.8 and a step-size coefficient ($F \in [0, 1]$) of 0.85, this means that our algorithm builds the next generation preserving the architecture of the best element of the previous generation (the value of the crossover coefficient is low), but even preserving the variance of the previous generation (the value of the step-size coefficient is high). The task used to evaluate the robot is structured as follows. We used a T-maze as the learning environment (see Figure 17b). Each robot is placed at the beginning of each crossroads and it is free to run for about 30 seconds. The final evaluation of the "life" of a robot is the product of the evaluations obtained in each of the distinct simulations. The fitness function that evaluates the robot in every crossroad is made of two components. The first component F_M is derived from the one proposed by Floreano and Mondada (1994) and consists of a reward for straight, fast movements and obstacle avoidance. This component is the same in the left and the right-follower task. The second component changes between the two epochs; in the right-follower training it rewards the robot that turns right at a crossroads (F_R), in the



(a) *Best Case.* All the tasks are correctly reproduced.

(b) *Worst Case.* The correct executed tasks are T_1, T_2, T_4, T_5, T_6 and T_8.

Figure 13. HPNNA sample outputs for Dataset $\mathcal{D}_2$ and $\lambda = 2$.

left-follower training it rewards the robot that turns left (F_L). In equation (11)$\bar{V}$ is the average speed of the robot, V_A is the average angular speed. S_{min} is the value of the shortest distance measured from an obstacle during the task period

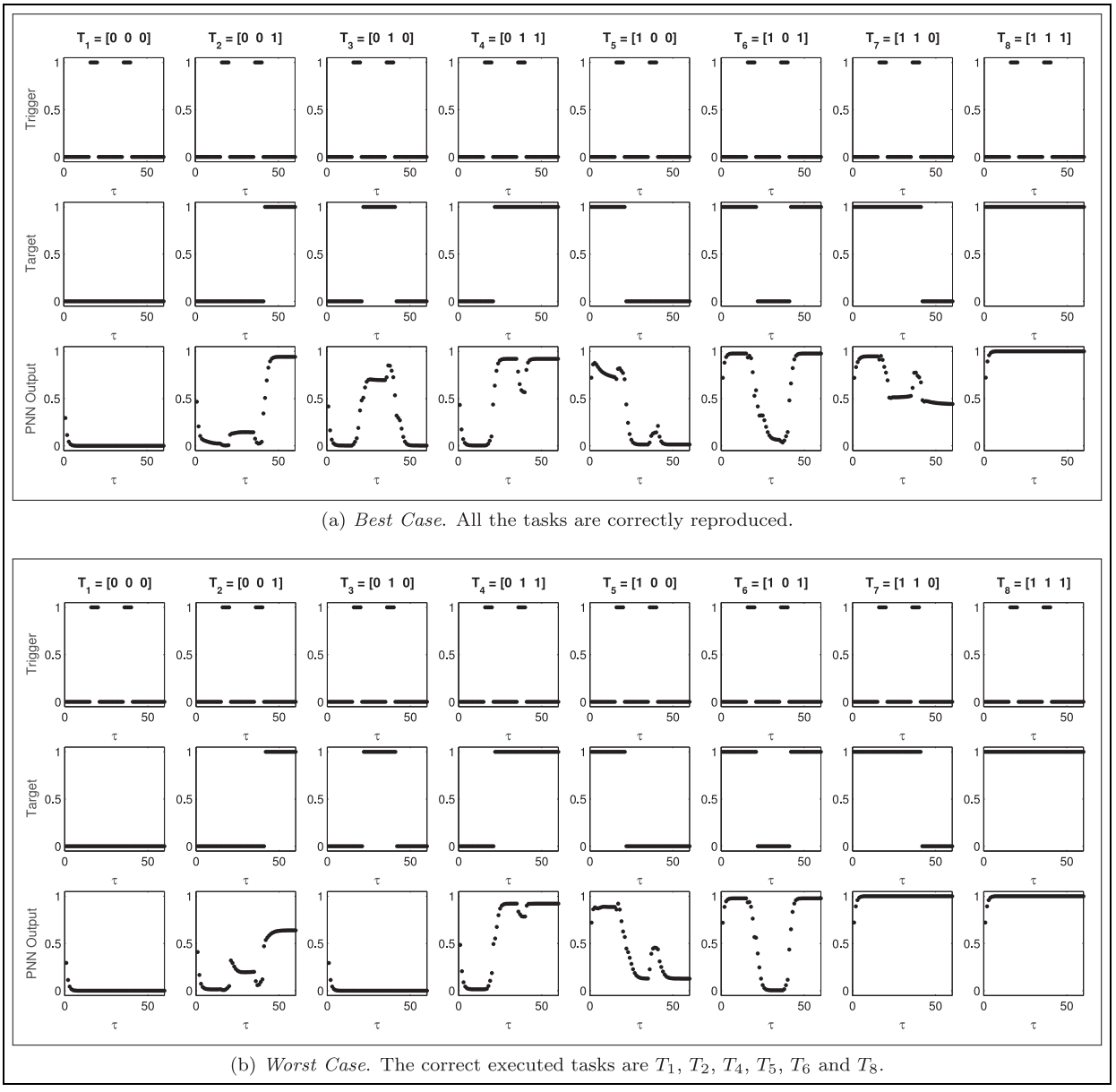
$$F_M = \bar{V} \cdot (1 - \sqrt{V_A}) \cdot S_{min} \qquad V_A \in [0,1], S_{min} \in [0,1]; \tag{11}$$

$$F_R = \bar{S}_1 + \bar{S}_2 - \bar{S}_9 - \bar{S}_{10} \qquad F_L = \bar{S}_9 + \bar{S}_{10} - \bar{S}_1 - \bar{S}_2. \tag{12}$$

In F_R the average measure of the left sonars over the task period is subtracted from the average measure of the right ones, the opposite happens in F_L.

In Figure 18 we show the mean fitness evolution per step of the motor primitives. The interpreter L_1 fed with the best evolved code programs was tested placing the robot in ten different positions in the maze and observing the robot behavior while driving through the crossroads three times. The positions were chosen in such a way that the robot starts its test in the middle of a corridor, oriented with its lateral part parallel to the wall. We tested one code at a time for each execution without dynamically changing the values. In these conditions the interpreter L_1 fed with P_r and P_l was successful in all the trials, showing the appropriate behavior in each of the corridors: L_1 was able to control the robot without crashing and preserving the right motor primitive.

Finally, we show the results of the whole HPNNA control framework in mazes of the kind of Figure 6, for



(a) *Best Case.* All the tasks are correctly reproduced.

(b) *Worst Case.* The correct executed tasks are T_1, T_2, T_4, T_5, T_6 and T_8.

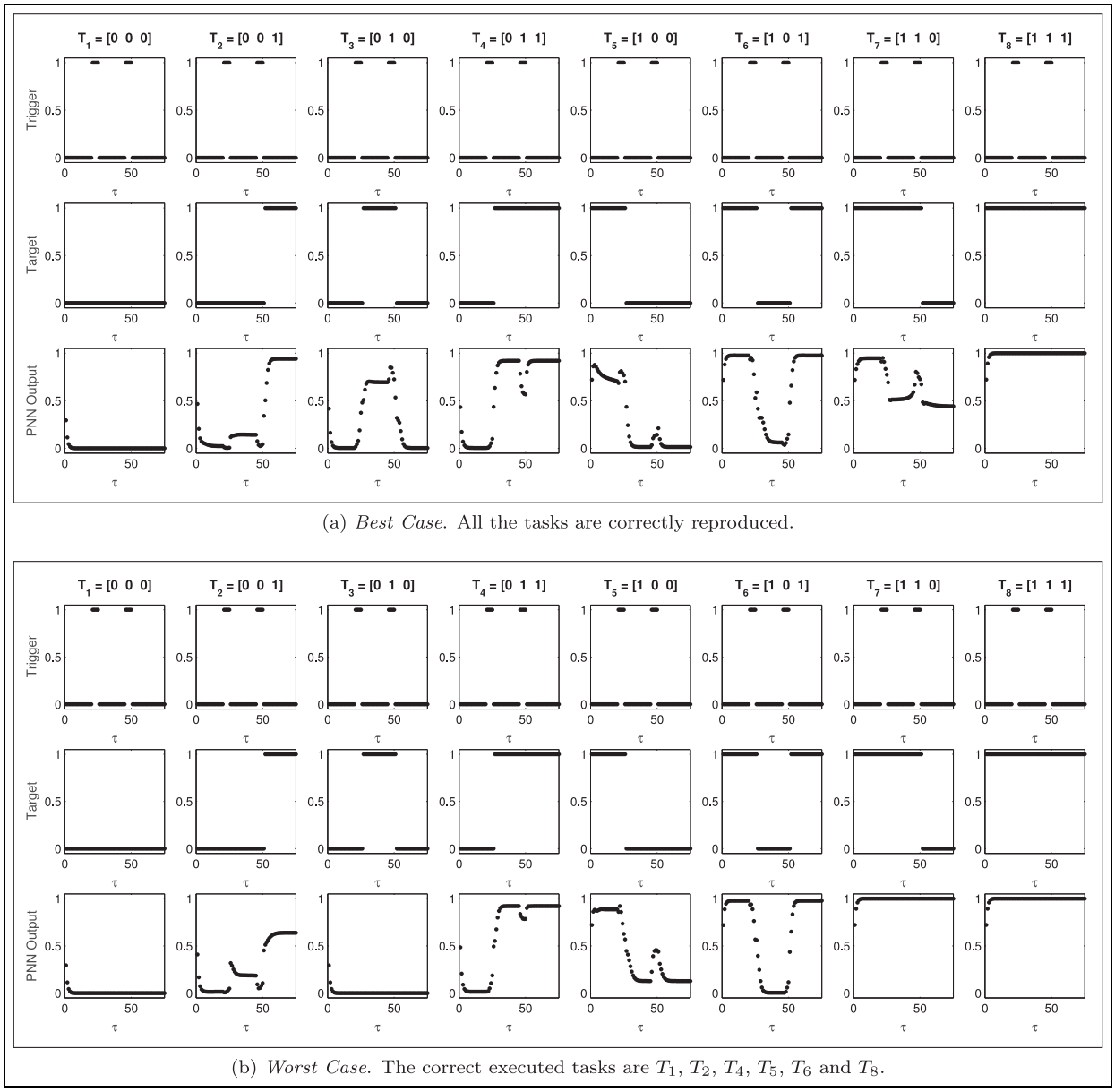
Figure 14. HPNNA sample outputs for Dataset $\mathcal{D}_2$ and $\lambda = 3$.

all possible programs learned in the exploring behavior experiments. The trigger signal for the higher level interpreter is for simplicity derived from the output of the angular velocity neuron of the first interpreter (however clever triggering signals from the interpreter could be imagined). We tested it on mazes with different sizes ($\lambda \in 2, 3, 4$). This is to stress that what we learned is not a particular trajectory in an environment but a high level goal encoded by the program and not influenced by moderate changing in the environment. A test is considered successful if the distance from the goal is under a certain threshold and the robot does not crash. Thus, if the robot reaches a place different from the one "programmed" the test fails. Moreover, we stressed the robustness of the programs learned by applying a relative error ε on each learned parameter p during the execution in the maze environment. These noisy parameter values were drawn from a Gaussian distribution centred in the parameter value p and with a standard deviation of $\varepsilon = k{\cdot}w/100$ where $k \in \{10, 15, 20\}$.

Table 13 shows the results. The small decrease in performance for shorter corridor lengths is mainly due to the shorter duration of the trigger on the higher level network. High values of relative error make the probability of failure increase in the maze exploring. However even a relative error of 15% does not erase the behavior of the HPNNA preserving a high success rate.

4 Conclusions

We have proposed a hierarchical programmable neural network architecture, HPNNA, composed of a



(a) *Best Case.* All the tasks are correctly reproduced.

(b) *Worst Case.* The correct executed tasks are T_1, T_2, T_4, T_5, T_6 and T_8.

Figure 15. HPNNA sample outputs for Dataset $\mathcal{D}_2$ and $\lambda = 4$.

hierarchy of modules where each module can be viewed both as an interpreter network capable of running different programs without modifying its synaptic connections and as programmer network capable of controlling the behavior of the lower modules. This implies that the same neuronal substrate can encode multiple motor primitives. Furthermore, the motor primitives can be learned incrementally by increasingly adding more programs to the interpreter network. The learning of primitives in a lower level is transferred to the higher level; new primitives can be added in a fixed lower level by searching for the corresponding programming inputs that the higher level should send. We explored the parameter space resulting from this modelization by means of an evolutionary-based learning approach. The programming inputs of a higher level are fixed with respect to the dynamics of its corresponding lower level, thus learning multiple behavior codes (*programs*) resulted in being computationally simpler with respect to learning dynamics of multiple behaviors. We successfully tested the performance of the HPNNA architecture in tasks of increasing complexity. Our proposal has implications from both neuroscientific and computational perspectives as we discuss below.

4.1 Neuroscientific perspectives

From a neuroscientific perspective, we present a novel proposal on (hierarchical) action organization and control by the brain, which can be summarized as an

Table 10. Generalization capabilities of HPNNA and NOA when trained with dataset $\mathcal{D}_1$ (λ = 2). The architectures were tested for two unknown new maze sizes corresponding to λ = 2.4 and λ = 3.6.

λ	Architecture	B/W case	Fitness value (number of correct turns) per task								Task success
			T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	
2.4	NOA	*Best*	0.003 (3/3)	16.165 (2/3)	3.055 (3/3)	1.034 (3/3)	1.033 (3/3)	17.177 (2/3)	16.163 (2/3)	0.003 (3/3)	5/8
		Worst	0.003 (3/3)	16.164 (2/3)	17.173 (2/3)	0.048 (3/3)	0.024 (3/3)	17.174 (2/3)	16.164 (2/3)	0.003 (3/3)	4/8
	HPNNA	*Best*	0.005 (3/3)	7.137 (3/3)	0.065 (3/3)	0.045 (3/3)	0.028 (3/3)	1.064 (3/3)	2.088 (3/3)	0.005 (3/3)	8/8
		Worst	0.005 (3/3)	3.109 (3/3)	17.175 (2/3)	0.085 (3/3)	0.099 (3/3)	0.054 (3/3)	0.098 (3/3)	0.005 (3/3)	7/8
3.6	NOA	*Best*	0.003 (3/3)	22.226 (2/3)	3.062 (3/3)	1.037 (3/3)	1.036 (3/3)	23.237 (2/3)	22.223 (2/3)	0.003 (3/3)	5/8
		Worst	0.003 (3/3)	22.224 (2/3)	23.233 (2/3)	1.060 (3/3)	0.027 (3/3)	23.234 (2/3)	22.224 (2/3)	0.004 (3/3)	4/8
	HPNNA	*Best*	0.005 (3/3)	16.215 (2/3)	0.071 (3/3)	0.048 (3/3)	0.031 (3/3)	23.254 (2/3)	7.142 (3/3)	0.005 (3/3)	6/8
		Worst	0.005 (3/3)	20.258 (2/3)	23.235 (2/3)	0.103 (3/3)	0.122 (3/3)	0.065 (3/3)	6.149 (3/3)	0.005 (3/3)	6/8

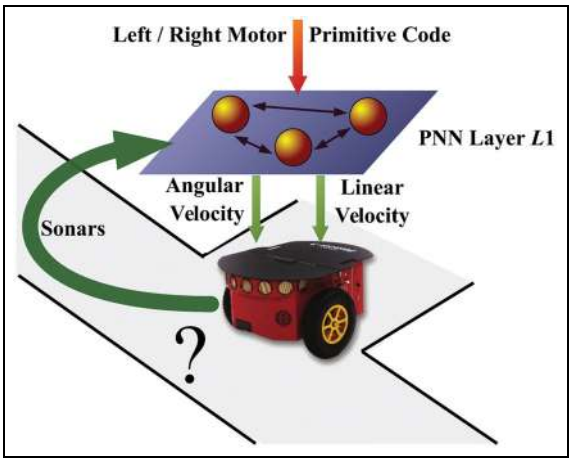


Figure 16. General presentation of the sought L_1 module. Its output controls the Pioneer 3DX angular and linear velocity. It has two inputs, a motor-primitive code and the sonars.

interpreter-programmer computational scheme. The *interpreter* network is able to store multiple action primitives within a common neural substrate. Not only is this encoding scheme parsimonious, avoiding the shortcomings of strong modularity, but it also affords flexible and plausible cognitive control by the *programmer* network. The programmer network can enforce rule-like behaviors by instantaneously instructing the interpreter network, without the necessity of re-learning. Such fast switches of behavior are the hallmark of cognitive control.

The system learns to represent goals (encoded in the programming input), not trajectories in the environment; this affords the flexible adaptation to changing environmental conditions (e.g. moderate changes of dimensions and sensory cues in a maze). Furthermore, the proposed computational scheme can be iterated to realize hierarchies having increasing levels of complexity (as a network playing the role of *programmer* relative to a lower-level *interpreter* can also play the role of an *interpreter* relative to a higher-level *programmer*). This provides a novel organizing principle for cortical hierarchies and their role in supporting goal-directed actions.

Overall, the proposed *interpreter-programmer* scheme is consistent, on the one hand, with the idea of multiple motor primitives in (pre)motor areas (Rizzolatti, Camarda, Fogassi, Gentilucci, Luppino, & Matelli, 1988), and on the other hand with control- and information-theoretic approaches to prefrontal cortex (Koechlin & Summerfield, 2007), and with its role in biasing (instantaneously) behavior (Miller & Cohen, 2001). At the same time, it goes beyond theoretical proposals on executive functions and suggests a plausible neural mechanism (programmability) for exerting cognitive control, which is based on the idea of "reusable" or "recycled" neuronal networks (Anderson, 2010;

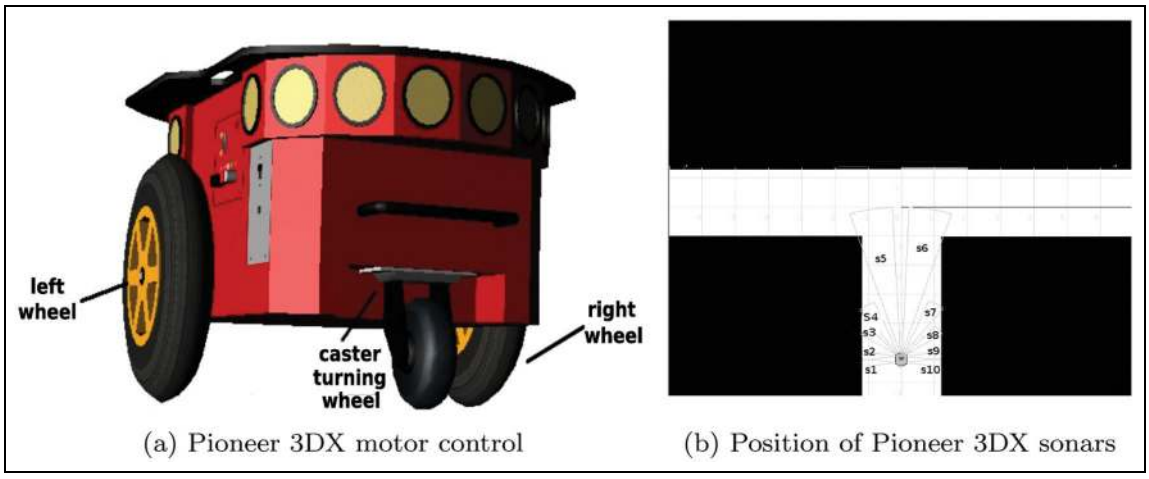


Figure 17. Pioneer 3DX simulation in Player-Stage environment.

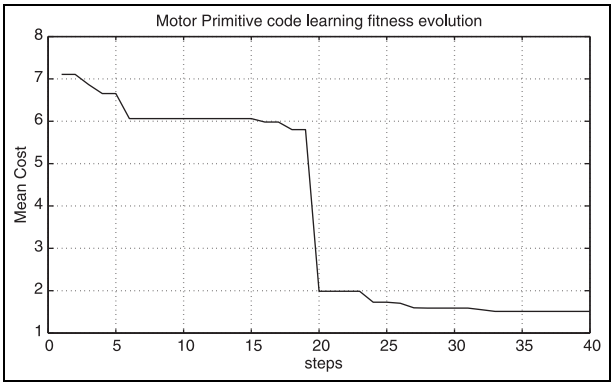


Figure 18. Mean cost evolution per step in Motor-Primitive code learning in the simulated robotic environment.

Dehaene, 2005) (and is therefore alternative to the idea of "gates" and of strongly modular networks). Further studies are of course necessary to evaluate the merits of this proposal, but it has to be noted that its computational parsimony, robustness and scalability (compared to alternative proposals) could offer advantages from an evolutionary viewpoint.

4.2 Computational perspectives

From a computational perspective, this architecture has numerous advantages. Concerning learning, the framework permits incremental learning and the separation of learning phases in different epochs. In fact, it is certainly possible to learn the same behavior in a "classical" way, by learning the weights of a network that at the same time receives the trigger and a program. In that case, one can apply two different strategies: (a) to train a single network able to exhibit all the behaviors; (b) to train one specific network for each behavior so as to obtain eight networks performing the desired behaviors. However, in both cases it is not trivial to accomplish this kind of training. In the first case, because one should be forced to learn all behaviors at the same time, which results in increasing difficulty as soon as the number of behaviors increases. In the second case, the drawback is the necessity of constructing a single network that combines different special purpose networks and switches among their output whenever it is needed. Moreover, in both cases it is difficult to add new behaviors to the learned system. Thus, our architecture suggests a promising neural network approach for these kinds of issues (Umedachi, Ito, & Ishiguro, 2015).

Furthermore, the possibility to steer goal sequences entails flexible behavioral control in the face of uncertain and (moderately) changing environments. Robustness of control is also advantageous to scale up the architecture hierarchically. As moderate errors in program values do not change the overall architecture behavior, programs can be used as outputs of other network modules, realizing hierarchies of control. When a hierarchical organization is built, higher-level modules are necessarily slower than lower-level ones, as they need to guide the realization of sequences of actions (Paine & Tani, 2004).

4.3 Open issues

Finally, the proposed architecture can further be improved in a number of directions. Firstly, in our approach the discovery of new input programs lets the level exhibit novel primitive patterns, without having to relearn already acquired behavior (i.e. incrementally). However, this incremental learning may eventually suffer limitations, for two main reasons;

1. Each hierarchical level has a fixed level complexity, i.e. can simulate networks of a finite size. If the primitive to be learned has a larger complexity, it

Table 11. Generalization capabilities of HPNNA and NOA when trained with dataset $\mathcal{D}_2$ ($\lambda \in \{2, 3, 4\}$). The architectures are tested for two new maze sizes corresponding to $\lambda = 2.4$ and $\lambda = 3.6$.

λ	Architecure	B/W case	Fitness value (number of correct turns) per task								Task success
			T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	
2.4	NOA	*Best*	0.013 (3/3)	2.134 (3/3)	17.178 (2/3)	0.102 (3/3)	1.036 (3/3)	17.174 (2/3)	16.164 (2/3)	0.004 (3/3)	5/8
		Worst	0.003 (3/3)	16.164 (2/3)	17.179 (2/3)	1.066 (3/3)	0.036 (3/3)	17.175 (2/3)	16.163 (2/3)	0.003 (3/3)	4/8
	HPNNA	*Best*	0.005 (3/3)	0.052 (3/3)	0.083 (3/3)	0.061 (3/3)	0.059 (3/3)	0.062 (3/3)	0.164 (3/3)	0.005 (3/3)	8/8
		Worst	0.005 (3/3)	0.107 (3/3)	17.175 (2/3)	0.059 (3/3)	0.106 (3/3)	0.045 (3/3)	16.165 (2/3)	0.005 (3/3)	6/8
3.6	NOA	*Best*	0.017 (3/3)	3.176 (3/3)	23.240 (2/3)	1.127 (3/3)	1.040 (3/3)	23.234 (2/3)	22.225 (2/3)	0.004 (3/3)	5/8
		Worst	0.003 (3/3)	22.224 (2/3)	23.240 (2/3)	0.082 (3/3)	0.041 (3/3)	23.235 (2/3)	22.223 (2/3)	0.003 (3/3)	4/8
	HPNNA	*Best*	0.005 (3/3)	0.066 (3/3)	0.102 (3/3)	0.071 (3/3)	0.078 (3/3)	0.068 (3/3)	0.221 (3/3)	0.005 (3/3)	8/8
		Worst	0.005 (3/3)	0.141 (3/3)	23.235 (2/3)	0.069 (3/3)	0.128 (3/3)	0.048 (3/3)	22.225 (2/3)	0.005 (3/3)	6/8

Table 12. Experimental parameters for the *Motor Primitives* learning task.

Network parameters $\boldsymbol{\beta}$	Time constants	2τ
	Minimum weight value w_{min}	-5
	Maximum weight value w_{max}	5
	Integration step Δt	0.2
	Fitness optimum value *opt*	2
	Maximum number of iterations I_{max}	1000
DE parameters $\boldsymbol{\theta}$	Population number	20
	Parameter space size	12
	Step size	0.85
	Crossover probability	0.8
	Strategy	DE/BEST/2/*BIN*

Table 13. Results of the HPNNA control in *T*-mazes in tests with three different sizes and with different relative errors on the programs. For each test the success rate is reported.

Maze Type	Success Rate
$\lambda = 3$	100%
$\lambda = 4$	99%
$\lambda = 2$	94%
$\lambda = 3$, 10% error	98%
$\lambda = 3$, 15% error	77%
$\lambda = 3$, 20% error	68%

cannot be added without increasing the size of "slow" neurons of the level.

2. Each hierarchical level is affected by an intrinsic "precision error" because of the presence of the *mul* approximation that crucially relies on the settings of the time constants and on a finite number of neurons M. In other words, an output noise on the *mul* networks is present that could prevent the learning from adding the wanted novel primitive behavior.

In both cases, the changing of a hierarchical level structure exposes the cost of potentially disrupting all previously learned primitives. A future modeling improvement would be to add a mechanism capable of augmenting the structure without disrupting the existing programs.

Moreover, in our hierarchical scheme implementation, each level receives a "standard" data input that can be an external sensory input (as in our tests with sonars in L_1). In principle, the L_2 data input could rely on some other sensor output, however in our tests we showed that, as a matter of fact, the trigger information on T -intersection detection is already contained in the outputs of L_1. As a general consideration, in our scheme, it is a good idea to include, on the data input line, a feedback input coming from the lower level that could bring information on the timing of the task.

However, possibly other (probably lower) levels could bring essential information for the "current level", so this choice could be somewhat limiting. On the other hand, to include all the levels as a possible input connection would increase computational cost especially for "deep" hierarchies. One line of research would be to add this input choice at a learning level, letting the system decide on the input (coming from the available levels) that maximizes the primitive learning and consequently adapt its connections.

Another open issue is how and where to store the program values in a neural system so that they will be available when needed. This might be met using some reverberant scheme, which in the end will probably require appealing again to synaptic plasticity in ancillary networks. Finally, two lines for future research are assessing the biological plausibility of the proposed model, and advancing more detailed proposals (at the neuronal level) of its mechanisms of learning and recall of the programs.

Acknowledgement

The authors would like to thank Giuseppe Trautteur for the inspiring discussions and comments that greatly contributed to improve the paper.

Funding

The present research is funded by the Human Frontier Science Program (HFSP), award number RGY0088/2014, by the EU's FP7 under grant agreement no FP7-ICT-270108 (Goal-Leaders). The GEFORCE Titan used for this research was donated by the NVIDIA Corporation.

References

Agmon, E., & Beer, R. D. (2013). The evolution and analysis of action switching in embodied agents. *Adaptive Behavior*, *22*(1), 3–20.

Anderson, M. L. (2010). Neural re-use as a fundamental organizational principle of the brain. *Behavioral and Brain Sciences*, *33*(04), 245–266.

Araújo, D., Diniz, A., Passos, P., & Davids, K. (2014). Decision making in social neurobiological systems modeled as transitions in dynamic pattern formation. *Adaptive Behavior*, *22*(1), 21–30.

Bakker, B., & Schmidhuber, J. (2004). Hierarchical reinforcement learning based on subgoal discovery and subpolicy specialization. In: F. Groen, N. Amato, A. Bonarini, E. Yoshida, & B. Krose (Eds.), *Proceedings of the 8th Conference on Intelligent Autonomous Systems, IAS-8* (pp. 438–445). Amsterdam, The Netherlands.

Bargmann, C. I. (2012). Beyond the connectome: How neuromodulators shape neural circuits. *Bioessays*, *34*, 485–65.

Beer, R. D. (1995). On the dynamics of small continuous-time recurrent neural networks. *Adaptive Behavior*, *3*(4), 469–509.

Candidi, M., Curioni, A., Donnarumma, F., Sacheli, L. M., & Pezzulo, G. (2015). Interactional leader–follower sensorimotor communication strategies during repetitive joint actions. *Journal of The Royal Society Interface*, *12*(110), 453–467.

Chersi, F., Donnarumma, F., & Pezzulo, G. (2013). Mental imagery in the navigation domain: A computational model of sensory-motor simulation mechanisms. *Adaptive Behavior*, *21*(4), 251–262.

d'Avella, A., Portone, A., Fernandez, L., & Lacquaniti, F. (2006). Control of fast-reaching movements by muscle synergy combinations. *The Journal of Neuroscience*, *26*(30), 7791–7810.

De Falco, I., Della Cioppa, A., Donnarumma, F., Maisto, D., Prevete, R., & Tarantino, E. (2008). CTRNN parameter learning using differential evolution. In: M. Ghallab, C. D. Spyropoulos, N. Fakotakis, & N. Avouris (Eds.), *ECAI 2008, 18th European Conference on Artificial Intelligence* (pp. 783–784). Patras, Greece: IOS Press.

Dehaene, S. (2005). Evolution of human cortical circuits for reading an arithmetic: The "Neuronal Recycling" hypothesis. *From Monkey Brain to Human Brain: A Fyssen Foundation Symposium* (pp. 133–157). Bradford, USA: MIT Press.

Dindo, H., Donnarumma, F., Chersi, F., & Pezzulo, G. (2015). The intentional stance as structure learning: A computational perspective on mindreading. *Biological Cybernetics*, *109*(4), 453–467.

Donnarumma, F., Murano, A., & Prevete, R. (2015a). Dynamic network functional comparison via approximate-bisimulation. *Control & Cybernetics*, *44*(1), 99–127.

Donnarumma, F., Prevete, R., Chersi, F., & Pezzulo, G. (2015b). A programmer–interpreter neural network architecture for prefrontal cognitive control. *International Journal of Neural Systems*, *25*(6), 1550017 (16 pages).

Donnarumma, F., Prevete, R., & Trautteur, G. (2010). How and over what timescales does neural reuse actually occur? Commentary on "Neural re-use as a fundamental organizational principle of the brain", by Michael L Anderson. *Behavioral and Brain Sciences*, *33*(04), 272–273.

Donnarumma, F., Prevete, R., & Trautteur, G. (2012). Programming in the brain: A neural network theoretical framework. *Connection Science*, *24*(2–3), 71–90.

Eliasmith, C. (2005). A unified approach to building and controlling spiking attractor networks. *Neural Computation*, *17*(6), 1276–1314.

Eliasmith, C., & Anderson, C. H. (2004). *Neural engineering: Computation, representation, and dynamics in neurobiological systems*. Cambridge, MA: The MIT Press.

Flash, T., & Hochner, B. (2005). Motor primitives in vertebrates and invertebrates. *Current Opinion in Neurobiology*, *15*(6), 660–666.

Floreano, D., & Mondada, F. (1994). Automatic creation of an autonomous agent: Genetic evolution of a neural-network driven robot. In: *Proceedings of the Conference on Simulation of Adaptive Behavior* (pp.421–430). Cambridge, MA: MIT Press.

Fogassi, L., Ferrari, P., Chersi, F., Gesierich, B., Rozzi, S., & Rizzolatti, G. (2005). Parietal lobe: From action organization to intention understanding. *Science*, *308*, 662–667.

Friston, K. (2003). Learning and inference in the brain. *Neural Networks*, *16*(9), 1325–1352.

Gerkey, B., Vaughan, R., & Howard, A. (2003). The player/stage project: Tools for multi-robot and distributed sensor

systems. In: *International Conference on Advanced Robotics (ICAR)* (pp. 317–323). Coimbra, Portugal: IEEE Press.

Graziano, M. (2006). The organization of behavioral repertoire in motor cortex. *Annual Review of Neuroscience*, *29*, 105–134.

Hamilton, A. F. d. C., & Grafton, S. T.2007. The motor hierarchy: From kinematics to goals and intentions. In: P. Haggard, Y. Rossetti, & M. Kawato (Eds.), *Sensorimotor foundations of higher cognition* (pp. 381–408). Oxford: Oxford University Press.

Haruno, M., Wolpert, D., & Kawato, M. (2003). Hierarchical MOSAIC for movement generation. In: T. Ono, G. Matsumoto, R. Llinas, A. Berthoz, H. Norgren, & R. Tamura (Eds.), *Excepta Medica International Congress Series* (pp. 575–590). Amsterdam, The Netherlands: Elsevier Science.

Hioki, T., Miyazaki, Y., & Nishii, J. (2013). Hierarchical control by a higher center and the rhythm generator contributes to realize adaptive locomotion. *Adaptive Behavior*, *21*(2), 86–95.

Hopfield, J. J., & Tank, D. W. (1986). Computing with neural circuits: A model. *Science*, *233*, 625–633.

Igari, I., & Tani, J. (2009). Incremental learning of sequence patterns with a modular network model. *Neurocomputing*, *72*(7–9), 1910–1919.

Kelly, J. P. (1991). The neural basis of perception and movement. *Principles of Neural Science* (3rd ed., pp. 283–295). New York, NY: Elsevier.

Koechlin, E., & Summerfield, C. (2007). An information theoretical approach to prefrontal executive function. *Trends in Cognitive Science*, *11*(6), 229–235.

Maisto, D., Donnarumma, F., & Pezzulo, G. (2015). Divide et impera: Subgoaling reduces the complexity of probabilistic inference and problem solving. *Journal of The Royal Society Interface*, *12*(104), 20141335.

Mcgovern, A., & Barto, A. G. (2001, June 18–22). Accelerating reinforcement learning through the discovery of useful subgoals. In: *Proceedings of the 6th International Symposium on Artificial Intelligence, Robotics, and Automation in Space: i-SAIRAS, Canadian Space Agency* (pp. 13–18). Montreal, Canada: Electronically Published.

Miller, E. K., & Cohen, J. D. (2001). An integrative theory of prefrontal cortex function. *Annual Review on Neuroscience*, *24*, 167–202.

Montone, G., Donnarumma, F., & Prevete, R. (2011). A robotic scenario for programmable fixed-weight neural networks exhibiting multiple behaviors. In: *Adaptive and Natural Computing Algorithms* (pp. 250–259). Heidelberg, Germany: Springer Berlin.

Mussa-Ivaldi, F. A., & Bizzi, E. (2000). Motor learning through the combination of primitives. *Philosophical Transactions of the Royal Society of London. Series B: Biological Sciences*, *355*(1404), 1755–1769.

Paine, R. W., & Tani, J. (2004). Motor primitive and sequence self-organization in a hierarchical recurrent neural network. *Neural Networks*, *17*(8–9), 1291–1309.

Paine, R. W., & Tani, J. (2005). How hierarchical control self-organizes in artificial adaptive systems. *Adaptive Behavior*, *13*(3), 211–225.

Park, H.-J., & Friston, K. (2013). Structural and functional brain networks: From connections to cognition. *Science*, *342*(6158), 1238411.

Pezzulo, G., Donnarumma, F., & Dindo, H. (2013). Human sensorimotor communication: A theory of signaling in online social interactions. *PLoS ONE*, *8*(11), e79876.

Pezzulo, G., Donnarumma, F., Iodice, P., Prevete, R., & Dindo, H. (2015). The role of synergies within generative models of action execution and recognition: A computational perspective: Comment on grasping synergies: A motor-control approach to the mirror neuron mechanism by A D'Ausilio et al. *Physics of Life Reviews*, *12*, 114–117.

Price, K. V., Storn, R. M., & Lampinen, J. A. (2005). *Differential evolution: A practical approach to global optimization*. Natural Computing Series. Springer-Verlag.

Rizzolatti, G., Camarda, R., Fogassi, L., Gentilucci, M., Luppino, G., & Matelli, M. (1988). Functional organization of inferior area 6 in the macaque monkey. II. Area F5 and the control of distal movements. *Experimental brain research*, *71*(3), 491–507.

Tani, J. (2003). Learning to generate articulated behavior through the bottom-up and the top-down interaction processes. *Neural Networks*, *16*(1), 11–23.

Tani, J., Ito, M., & Sugita, Y. (2004). Self-organization of distributedly represented multiple behavior schemata in a mirror system: Reviews of robot experiments using RNNPB. *Neural Networks*, *18*(1), 103–104.

Tani, J., Nishimoto, R., & Paine, R. W. (2008). Achieving "organic compositionality" through self-organization: Reviews on brain-inspired robotics experiments. *Neural Networks*, *21*(4), 584–603.

Tani, J., & Nolfi, S. (1999). Learning to perceive the world as articulated: An approach for hierarchical learning in sensory-motor systems. *Neural Networks*, *12*(7), 1131–1141.

Thoroughman, K. A., & Shadmehr, R. (2000). Learning of action through adaptive combination of motor primitives. *Nature*, *407*(6805), 742–747.

Trautteur, G., & Tamburrini, G. (2007). A note on discreteness and virtuality in analog computing. *Theoretical Computer Science*, *371*, 106–114.

Umedachi, T., Ito, K., & Ishiguro, A. (2015). Soft-bodied amoeba-inspired robot that switches between qualitatively different behaviors with decentralized stiffness control. *Adaptive Behavior*, *3*(2), 97–108.

Woodman, M., Perdikis, D., Pillai, A. S., Dodel, S., Huys, R., Bressler, S., & Jirsa, V. (2011). Building neurocognitive networks with a distributed functional architecture. *Advances in Experimental Medicine and Biology, 718*, 101–109. doi:10.1007/978-1-4614-0164-3_9

Yamauchi, B. M., & Beer, R. D. (1994). Sequential behavior and learning in evolved dynamical neural networks. *Adaptive Behavior*, *2*(3), 219–246.

About the Authors

Francesco Donnarumma (MSc in physics, PhD in computer and information science) has been a research fellow at ISTC-CNR since 2011. His research focuses on computational modelling of cognitive brain functions by the developing of biologically inspired models investigating social interactions and studying multi-purpose interpreter architectures based on dynamical neural networks.

Roberto Prevete (MSc in physics, PhD in information science) is an Assistant Professor of Computer Science at the Dept.of Electrical Engineering and Information Technologies (DIETI), University of Naples Federico II, Italy. Director of the laboratory for Computational Vision and Neural Networks (ViNe) at DIETI. His current research interests include computational models of brain mechanisms, machine learning and artificial neural networks and their applications.

Andrea de Giorgio is currently working on his final thesis in machine learning as a master student at KTH, Royal Institute of Technology in Stockholm, Sweden. In 2013 he received his bachelor's degree in electronic engineering at the University of Naples Federico II, Italy. His research interests focus on deep learning and modeling of brain functions.

Guglielmo Montone (MSc in physics, PhD in computer and information science) is a postdoctoral researcher at LPP, UniversitÃl' Paris Descartes. His research focuses on artificial neural networks and their applications in artificial intelligence. His current research is about the development of the concept of space in simple and biologically plausible agents.

Giovanni Pezzulo (MSc and PhD in cognitive psychology) is a researcher at the Institute of Cognitive Sciences and Technologies, National Research Council, Rome, Italy. His main research interests are prediction, goal-directed behaviour, internally generated neuronal activity and joint action in living organisms and robots. His current research interests are focused on the realization of biologically realistic cognitive models for decision making and planning.