POLITECNICO DI MILANO
Corso di Laurea Specialistica in Ingegneria Informatica
Dipartimento di Elettronica e Informazione

A COGNITIVE ARCHITECTURE BASED ON AN AMYGDALA-THALAMO-CORTICAL MODEL FOR DEVELOPING NEW GOALS AND BEHAVIORS: APPLICATION IN HUMANOID ROBOTICS

AI & R Lab
Laboratorio di Intelligenza Artificiale
e Robotica del Politecnico di Milano

Relatore: Prof.ssa Giuseppina Gini
Correlatore: Prof. Riccardo Manzotti
Correlatore: Ing. Flavio Mutti

Tesi di Laurea di:
Marco Burrafato, matricola 739401
Luca Florio, matricola 750029

Anno Accademico 2011-2012

A noi, perché ce lo siamo meritato.

Contents

List of Figures

Abstract

Cognitive development concerns the evolution of human mental capabilities, through experience earned during life; however, the way human beings develop new goals and behaviors during their lifetime is not completely understood. Many researches have been done in order to realize agents that could develop autonomously through experience, interacting with the environment and adapting to it. Different approaches in robotics aim to realize this kind of agents, from behavior-based robotics to developmental robotics. An important feature needed to accomplish this objective is the self-generation of motivations and goals, as well as the development of complex behaviors consistent with them.
The objective of this thesis is to realize a bio-inspired cognitive architecture, based on an amygdala-thalamo-cortical model, capable of autonomously develop new goals and behaviors. This cognitive architecture has been implemented and tested using a humanoid robot. Experimental results show the main features of the architecture: the development of new goals starting from hard-coded ones, and the generation of complex movements consistent with those goals. Despite the architecture is still under development, good experimental results demonstrate that it is a good starting point in order to develop a more complex and effective system.

Estratto in lingua italiana

Nel corso della sua vita, dall'infanzia fino all'età adulta, l'essere umano sviluppa le proprie capacità mentali attraverso un processo detto sviluppo cognitivo, che riguarda il modo in cui una persona percepisce il mondo attorno a se, come pensa, e come comprende il mondo attraverso l'interazione di fattori genetici ed acquisiti. Un aspetto fondamentale nello sviluppo cognitivo di una persona è la generazione autonoma di nuovi comportamenti ed obiettivi, il che permette all'individuo di adattarsi alle diverse situazioni che deve affrontare ogni giorno. Come gli esseri umani siano in grado di sviluppare autonomamente questi nuovi obiettivi nel corso della loro esistenza non è completamente noto. La robotica, per realizzare agenti in grado di interagire in modo efficace con gli esseri umani e integrarli nella loro vita, si deve interessare dei processi interni al cervello umano che permettono lo sviluppo cognitivo dell'individuo, così come delle modalità alla base della generazione di nuovi obiettivi e comportamenti.
Scopo della nostra tesi è creare un modello robotico bio-ispirato basato sui processi interni al cervello umano, che permetta all'agente di sviluppare autonomamente nuovi obiettivi, nonché nuovi comportamenti che siano coerenti con questi obiettivi.

Esistono già diversi approcci nel campo della robotica per risolvere il problema dell'adattabilità e sviluppo autonomo delle motivazioni in un robot. La robotica "behavior-based" permette all'agente di adattare il proprio comportamento ai cambiamenti nell'ambiente, al fine di raggiungere i propri scopi. In questo approccio, gli obiettivi sono preimpostati nel robot, e non è possibile svilupparne di nuovi. La robotica epigenetica studia lo sviluppo cognitivo nei sistemi naturali e artificiali allo scopo di adattarsi all'ambiente e sviluppare autonomamente nuove motivazioni ed obiettivi che non erano previsti in fase di progettazione. A differenza di questi approcci, il sistema qui presentato si propone di investigare quel livello cognitivo intermedio che permette agli esseri umani di essere consapevoli dell'ambiente circostante, e

quindi di interagire con esso per la maggior parte delle operazioni di base dell'organismo.
Questa capacità è una condizione essenziale per consentire al robot di adattarsi nella vita quotidiana dell'essere umano: un robot deve essere in grado non solo di agire in modo coerente ai cambiamenti nell'ambiente circostante, ma anche di sviluppare obiettivi che possono emergere dalla situazione in cui si trova, in modo da interagire efficacemente con le persone con cui essa verrebbe in contatto. Tale robot arriverà anche a sviluppare una personalità unica, a seconda delle esperienze che hanno contribuito alla creazione dei suoi nuovi obiettivi e comportamenti. Queste caratteristiche renderebbo il robot ideale per applicazioni complesse come quelle dell'intrattenimento, sia nel privato che nel pubblico, permettendo alla robotica di entrare in campi di applicazione di attività comuni.

Il sistema che presentiamo consente ad un agente di sviluppare nuovi obiettivi, oltre a quelli già presenti, e di adeguare il proprio comportamento a questi obiettivi. Questo sistema si ispira alla meccanica del cervello umano, in particolare alla comunicazione di tre diverse aree del cervello: la corteccia cerebrale, il talamo e l'amigdala. L'interazione di queste tre aree è un elemento chiave per lo sviluppo cognitivo umano (ovvero lo sviluppo delle capacità intellettuali), ed è quindi considerato come un buon punto di riferimento per l'attuazione del nostro sistema. Su questa base è stata creata l'Architettura Intenzionale: si tratta di una rete di unità elementari, chiamate Moduli Intenzionali, che consente lo sviluppo di nuovi obiettivi. In aggiunta ai Moduli Intenzionali nella rete è presente anche un Modulo Filogenetico, il quale contiene gli obiettivi preimpostati, cioè gli "istinti innati" dell'agente. Questo modulo esegue le stesse funzioni dell'amigdala nei sistemi biologici. Attraverso l'azione del Modulo Filogenetico, più lo stato corrente del robot incontra gli obiettivi già fissati, maggiore è l'intensità del segnale in uscita dal modulo.
Ogni Modulo Intenzionale è composto da due moduli: il Modulo di Categorizzazione e il Modulo Ontogenetico. Il Modulo di Categorizzazione rappresenta la corteccia cerebrale del nostro sistema, e il suo ruolo è quello di restituire un vettore che rappresenti l'attivazione neurale della corteccia in risposta allo stato d'input.
Il Modulo Ontogenetico è la base dello sviluppo di nuovi obiettivi; riceve il vettore di attivazioni neurali dal Modulo di Categorizzazione, e impiega una funzione di apprendimento Hebbiano per sviluppare nuovi obiettivi. Una volta che la sua elaborazione è conclusa, restituisce un segnale che indica se lo stato corrente è conforme ai nuovi obiettivi.

Il Modulo Intenzionale analizza i segnali provenienti dai moduli Filogenetico e Ontogenetico, restituendo il più rilevante dei due, e restituendo anche il vettore di attivazione neurale calcolato dal Modulo di Categorizzazione.
Pertanto, il flusso di esecuzione dell'Architettura Intenzionale inizia quando l'input sensoriale viene filtrato e quindi inviato alla rete di Moduli Intenzionali, la quale elabora le informazioni ricevute; ciascun modulo restituisce un vettore contenente informazioni sullo stato dell'ambiente, e un segnale che indica quanto lo stato effettivo soddisfi gli obiettivi attuali. La rete può essere composta da più strati, ognuno contenente diversi Moduli Intenzionali, che possono essere collegati tra loro in modo diretto o retroazionato. L'uso di diversi livelli di Moduli Intenzionali permette all'utente di astrarre lo stato del sistema dall'ingresso sensoriale originale.
Il vettore di attivazioni neurali e il segnale calcolato dall'Architectura Intenzionale vengono poi utilizzati dal Sistema Motorio per generare movimenti che possano essere coerenti con gli obiettivi dell'agente. Ogni movimento è formato da una serie di componenti elementari, chiamati primitive motorie, che sono alla base del movimento sia umano sia animale: esse rappresentano le attivazioni muscolari nel tempo, e la loro composizione (cioè la sinergia muscolare) porta alla realizzazione di movimenti complessi. Per ottenere questo risultato anche nel sistema qui proposto, le informazioni provenienti dall'Architectura Intenzionale sono utilizzate per adattare il comportamento dell'agente ai nuovi obiettivi e alle nuove situazioni: ogni Comportamento Dinamico analizza i dati di ingresso per suggerire all'agente il miglior movimento da applicare a quella specifica situazione, ovvero, quale azione porterà alla realizzazione dei suoi obiettivi, siano essi innati o acquisiti. Il Comportamento Dinamico restituisce un insieme di attivazioni muscolari (cioè una composizione di primitive motorie), ognuna riferita a ciascun giunto dell'agente collegato al comportamento specifico. Queste attivazioni muscolari vengono poi utilizzate dall'agente per eseguire il movimento corretto.

L'Architettura Intenzionale si occupa quindi dello sviluppo cognitivo del robot, con l'analisi dell'input e la generazione di nuovi obiettivi, mentre il Sistema Motorio consente al robot di muoversi e intraprendere azioni al fine di raggiungere tali obiettivi.

Il sistema presentato è stato testato con due esperimenti, che ci permettono di verificare l'abilità dell'architettura di sviluppare nuovi goal così come la capacità di adattare il proprio comportamento ai suoi obiettivi.
In un primo esperimento l'Architettura Intenzionale è composta da una semplice rete con un singolo Modulo Intenzionale; l'agente (NAO, un robot

umanoide prodotto dalla Aldebaran Robotics) impara un nuovo obiettivo a partire da uno preimpostato: a partire da un istinto innato relativo a figure con colori saturi, l'agente è in grado di sviluppare autonomamente un interesse per la forma delle figure.
In un secondo esperimento, invece, vengono testate le capacità motorie. Questa volta la rete ha due livelli e tre Moduli Intenzionali, e il robot NAO si muove per massimizzare il valore del segnale rilevante proveniente dall'Architettura Intenzionale. Inoltre i movimenti sono generati dalla combinazione lineare delle primitive motorie.
Questi risultati sperimentali dimostrano l'efficacia del sistema, e hanno fornito alcuni suggerimenti su possibili miglioramenti e sviluppi futuri del progetto. L'architettura ha dimostrato di imparare nuovi obiettivi, nonché di agire in modo intelligente per soddisfare questi obiettivi. Tuttavia, il sistema è ancora in fase di sviluppo e molti miglioramenti potrebbero essere apportati, dalla capacità di sviluppare obiettivi e comportamenti più complessi, al miglioramento dell'implementazione per ottenere prestazioni più elevate. Oltre ai miglioramenti del sistema, sarà interessante l'implementazione dell'architettura in diversi agenti, per studiare l'interazione e il possibile sviluppo di obiettivi di gruppo.

I contributi principali di questo lavoro sono:

- la progettazione e realizzazione di un'architettura cognitiva basata su un modello amigdala-talamo-corticale;
- il test dell'architettura stessa, sperimentando la funzionalità di generazione di obiettivi dell'architettura replicando un esperimento eseguito con una precedente implementazione di un singolo Modulo Intenzionale;
- l'estensione dell'architettura cognitiva con l'implementazione del Sistema Motorio, che consente la creazione di movimenti tramite la composizione lineare delle primitive motorie.

Questa tesi è strutturata come segue.
Nel capitolo 2 presentiamo lo stato dell'arte, con vari riferimenti riguardanti l'aspetto biologico relativo al nostro sistema, e quindi diverse applicazioni tecniche in cui questi concetti sono stati utilizzati.
Nel capitolo 3 si discute l'Architettura Cognitiva, composta dall'Architettura Intenzionale e dal Sistema Motorio. In primo luogo l'Architettura Intenzionale è descritta con tutte le sue componenti: il Modulo Intenzionale e il Modulo Filogenetico. Per ognuna di queste parti forniamo una descrizione

dettagliata delle sue funzionalità. Dopodichè mostriamo il funzionamento del Sistema Motorio con i Comportamenti Dinamici, e mostriamo come l'output dell'Architettura Intenzionale sia utilizzato per la generazione di un movimento che possa essere coerente con gli obiettivi dell'agente, mediante composizione di primitive motorie.
Nel capitolo 4 viene descritto il progetto IDRA - Intentional Distributed Robotic Architecture, ovvero l'implementazione dell'architettura cognitiva descritta nel capitolo 3. Viene fornita una descrizione tecnica del programma, così come le linee guida per l'utilizzo del software.
Nel capitolo 5 sono descritti i test sperimentali e i risultati ottenuti dal sistema presentato. Vengono presentati due esperimenti, ciascuno a prova delle caratteristiche principali dell'architettura: sviluppo di nuovi obiettivi e generazione di movimenti.
Il capitolo 6 contiene le conclusioni e gli sviluppi futuri: riassumiamo i nostri obiettivi, le nostre valutazioni dei risultati ottenuti e i possibili sviluppi futuri del progetto.
In appendice A vi presentiamo alcuni estratti del codice di IDRA, con una breve descrizione.
In appendice B mostriamo la libreria che abbiamo implementato per l'esecuzione dell'algoritmo Independent Component Analysis (ICA), nonché una breve descrizione teorica dell'algoritmo.
In appendice C mostriamo la libreria che abbiamo implementato per l'esecuzione dell'algoritmo K-Means per la clusterizzazione, nonché una breve descrizione teorica dell'algoritmo.
In appendice D presentiamo brevemente il progetto Behavior Simulation, un software che abbiamo sviluppato per testare le funzionalità dei comportamenti dinamici prima della loro implementazione nel Sistema Motorio.

Acknowledgements

Luca
Il ringraziamento più grande va sicuramente ai miei familiari, che mi hanno sostenuto (e mantenuto) in questi duri anni di studio. Mi spiace di essere stato un po' intrattabile in alcune occasioni, ma si sa che solo con le persone alle quali vogliamo più bene riusciamo ad essere completamente noi stessi, nel bene e nel male. Grazie mamma e papà, per avermi sempre lasciato libero di seguire la mia strada ed avermi sempre sostenuto nelle mie scelte. Grazie con tutto il cuore.
Grazie a Giorgia, la mia splendida ragazza, che da poco più di un anno a questa parte è entrata nella mia vita, standomi vicino e riuscendo nella non facile impresa di farmi sentire amato e capito ogni giorno che abbiamo passato insieme. Grazie, per avermi fatto capire la differenza tra essere sereno ed essere felice.
Grazie agli amici della mia compagnia, che non sto a nominare tutti perché (per mia fortuna) sono veramente tanti: mi avete regalato quei momenti di allegria e spensieratezza senza i quali sarebbe stata sicuramente più dura raggiungere questo obiettivo. Grazie anche agli amici che ho conosciuto al Politecnico e che hanno condiviso con me gioie e dolori di questa vita universitaria. In particolare grazie al Coach e Teo, due meravigliose persone che mi hanno accompagnato in questa avventura ogni giorno dal primo all'ultimo anno, e a Frank, senza il quale questa tesi avrebbe richiesto sicuramente più tempo per essere realizzata. Grazie a tutti voi amici miei, per me siete più importanti di quello che pensate.
Grazie alla professoressa Gini per averci fatto da relatrice. Grazie a Flavio e Riccardo, che non solo ci hanno aiutato e supervisionato durante la realizzazione di questo progetto, ma ci hanno anche insegnato tanto.
Grazie a Marco, perché è stato un ottimo compagno di tesi (che grazie a lui ha delle immagini veramente belle!)
Grazie al Politecnico, perché dopo tutto mi ha reso una persona migliore.

Marco
Alla nostra relatrice, la prof.ssa Giuseppina Gini, senza la quale non potremmo scrivere questa tesi. A Flavio, che ci ha seguiti lungo tutta la preparazione della tesi. A Riccardo, che ci ha fornito le basi concettuali, e soprattutto i mezzi tecnici, sui quali lavorare. Ai miei compagni di tesi, Luca e Alessio, senza il vostro aiuto non avrei mai potuto finire questo lavoro. A Ilhwan, che ci ha portato il Nao da un altro continente.
A mia madre e a mio padre, che mi hanno supportato, e sopportato, in questi (venti)sei anni. A mia sorella Valeria, che si litiga sempre ma alla fine ci si vuole un gran bene. Ai miei zii, cugini, nonni, parenti tutti, che mi sono stati vicini in questi anni.
A tutti i miei colleghi ed amici del Politecnico che mi hanno accompagnato in questi anni. Al Matteo, "IlCoach", il mio clone. A Roberta, la mia BFF. A Christian, e alla nostra amicizia ventennale. A Davide, "dadokkio", grazie al quale ho una tavoletta grafica, e grazie alla quale ho realizzato tutte le immagini di questa tesi. A Silvia, che ci si aiuta nei momenti difficili. A Smaranda, perchè è un'architetta ma è nerd lo stesso. A Fabio, e a tutti i caffè insieme. Al Dolph, e a tutti i suoi soprannomi. A Francesco, che mi ha accompagnato fino a Parigi. Al mio videgame developing team: Riccardo, Matteo, Christian. A Matteo, Erica, Gillo, Elisabetta, Giovanni, Ylenia, Francesco, Daniele, e tutti gli amici che mi hanno tenuto compagnia in segreteria e in Airlab.
A tutta la mia compagnia di Melegnano: Andrea, Elena, Simo, Pietro, Cristina, Fo, Michela, Vera, Daniele, Anna, Teo, David, Valentina... che nonostante tutto mi invitano ancora a uscire con loro.
A tutti i miei amici e compagni del karate: Fulvia, Liliana, Alfredo, Desi, Mauro... e in particolare a Massimo, che finalmente la smetterà di chiedermi quando finisco gli studi e vado a lavorare.
A tutti gli altri amici che mi hanno accompagnato al di fuori di queste quattro mura. A Maria "Eco" Elisa, e ai nostri drammi sentimentali. A Valentina, e ai nostri viaggi a Wacken. A Diego e Alice, grazie ai quali sono ancora più nerd. A Mara e Saverio, e alle nostre uscite. Alla mia sceneggiatrice Paola, che mi coinvolge in progetti mastodontici.
Ad Antonella, che ha fatto avanti e indietro innumerevoli volte per aprirci la porta del laboratorio. Al JJ e alla Latteria, per la loro fornitura quotidiana di piadine e focaccine. Alle ferrovie italiane, che non mancano mai di movimentarti la mattinata. A chi stamperà questa tesi, se mi farà uno sconto.
Al Politecnico, che mi ha tenuto qui e mi ha istruito per tutti questi anni.

A tutti voi, un sincero grazie. Se sono arrivato fin qui, lo devo a voi.

Chapter 1

Introduction

"A good question is, of course, the key by which infinite answers can be educed."

Isaac Asimov, Foundation's Edge

During their life, from childhood through adolescence to adulthood, humans develop their mental capabilities: this process is called cognitive development, and concerns how a person perceives, thinks, and gains understanding of his or her world through the interaction of genetic and learned factors [1]. A fundamental aspect in the cognitive development of a person is the autonomous generation of new goals and behaviors, which allows the individual to adapt to the various situations he faces every day. How humans can develop autonomously new goals during their existence is not completely understood. In order to realize agents capable to interact in an effective way with humans and integrate in their life, robotics should study the processes of human brain which allow the cognitive development of the individual, as well as the modalities underlying the generation of new goals and behaviors.
Our thesis gives a contribution to the achievement of this objective: its purpose is to create a bio-inspired robotic model based on human brain processes, that must be able to allow the agent to autonomously develop new goals as well as new behaviors that could be consistent with these goals.

There are different approaches in robotics to the problem of adaptation and self-development of motivations. Behavior-based robotics allow the agent to adapt its behavior to changes in the environment, in order to accomplish its goals [2]. In this approach goals are hard-coded into the robot, which cannot develop new ones. Developmental robotics (also called

"epigenetic robotics") aims at modeling the development of cognition in natural and artificial systems [3]. Developmental robotics leads to the cognitive development of the agent, making it able to adapt to the environment and autonomously develop new motivations and goals, that were not present at design time. Unlike these approaches, the system we propose addresses an intermediate level of cognition that allows mammals and humans to be aware of the surrounding environment and then interact with it for most basic tasks.
This capability is an essential precondition to enable the robot to fit into the human's everyday life. A robot must be able not just to act in a consistent manner to the changes in the surrounding environment, but also to develop goals that can emerge from the situation in which it is located, in order to interact effectively with people with whom it would come into contact. Such robot would also develop a unique personality, depending on the experiences that contributed to the creation of its new goals and behaviors. These features would make it the perfect robot for advanced applications in the world of entertainment, either at home and not, pushing robotics beyond common application fields.

We present a system that allows the robot to develop new objectives, in addition to the hard-coded ones, and to adapt its own behavior to these objectives. This system is inspired by the mechanics of the human brain, in particular the communication of three different areas of the brain: the cerebral cortex, thalamus and amygdala. The interaction of these three areas is a key element in human cognitive development (i.e. the development of intellectual abilities), and it is therefore regarded as a good reference point for the implementation of our system [4, 5]. On this basis the Intentional Architecture has been created: it is a network of elementary units, called Intentional Modules, that enables the development of new goals. In addition to Intentional Modules in the network is also present a Phylogenetic Module, containing the hard-coded objectives, i.e. the "innate instincts" of the agent; this module performs the functions of the amygdala in biological systems. Through the action of the Phylogenetic Module, the more the current state of the robot meets the objectives already set, the higher is the signal coming out from the module.
Each Intentional Module consists of two internal modules: the Categorization Module, and the Ontogenetic Module. The Categorization Module is the cerebral cortex of our system, and its role is to return a vector that represents the neural activation of the cortex in response to the input state provided.

The Ontogenetic Module is the basis of the development of new objectives; it receives the vector of neural activations from the Categorization Module, and employs a Hebbian learning function to develop new goals. After its processing is done, it returns a signal indicating whether the current state meets the new goals defined.
The Intentional Module analyzes signals from the Phylogenetic and Ontogenetic Modules, returning the most relevant of the two, and returning also the vector of neural activation computed by the Categorization Module.
Therefore, the Intentional Architecture execution flow starts when the sensory input is filtered and then sent to the Intentional Modules network, which processes the received information; each module returns an vector containing information about the state of the environment, and a signal indicating how much the actual state is satisfying the actual goals. The network can be composed by several layers, each one containing several Intentional Modules, that can be connected to each other in straightforward or feedback mode. The use of different levels of Intentional Modules allows the user to abstract from the raw sensory input.
The vector of neural activations and the signal computed by the Intentional Architecture are then used by a Motor System to generate movements that can be consistent with the goals of the agent. Each movement is meant to be formed by a series of elementary components called motor primitives, which are on the basis of animal and human motion alike: they represent the muscles activations over time, and their composition (i.e. muscular synergy) leads to the execution of complex movements [6, 7, 8, 9, 10]. In order to obtain this result also in the system we propose, the informations coming from the Intentional Architecture are used to adapt the behavior of the agent to the new goals and situations: each Dynamic Behavior analyzes the input data to suggest to the agent the best movement to apply to that specific situation, that is, which action will lead to the fulfillment of its goals, either learned or innate. The Dynamic Behavior returns a set of muscle activations (namely a composition of motor primitives), each one referring to each joint of the agent connected to that specific Behavior. These muscle activations are then used by the agent to perform the correct movement.
The Intentional Architecture therefore is concerning with the cognitive development of the robot, by input analysis and the generation of new goals, while the Motor System allow the robot to move and take actions in order to accomplish these goals.

The presented system has been tested with two main experiments, which let us verify the goals development skills as well as the ability to adapt its

behavior to its goals.
In a first experiment the Intentional Architecture is put under test. With a simple network of a single Intentional Module, the agent (a NAO robot, a humanoid robot produced by Aldebaran Robotics) learns a new goal from a hard-coded one: starting from an innate instinct related to figures with high-saturated colors, the agent autonomously develops an interest to a particular shape of the figures.
In a second experiment instead, the motor capabilities are tested. This time the network has two layers and three Intentional Modules, and NAO robot moves to maximize the relevant signal coming form the Intentional Architecture; furthermore, movements are generated by linear combination of motor primitives.
These experimental results show the efficacy of the system, and they provided several clues about possible improvements and future developments of the project. The architecture has proven to learn new goals, as well to act in a smart way to satisfy these goals. However the system is still under development and a lot of improvements could be made, starting from the capability to develop more complex goals and behaviors, to technical improvements in order to obtain better performances. In addition to improvements of the system, it will be of interest the implementation of the architecture on several different agents, in order to study their interaction and the possible development of team goals.

The main contributions of this work are:

- the design and implementation of a Cognitive Architecture based on an amygdala-thalamo-cortical model;
- the validation of the architecture itself, by testing the goals generation feature of the architecture with the replication of an experiment performed with an early implementation of the Intentional Module structure;
- the extension of the Cognitive Architecture with the implementation of a Motor System, which allow the creation of movements by linear composition of motor primitives.

This thesis is structured as follows.
In chapter 2 we present the state of the art, with various references concerning the biological aspect related to our system, and then different technical applications where these concepts have been used.

In chapter 3 we discuss the Cognitive Architecture, composed by the Intentional Architecture and the Motor System. First the Intentional Architecture is described with all of its components: the Intentional Module and the Phylogenetic Module. For each one of these parts we provide a detailed description of its functionalities. Then we show the functioning of the Motor System with dynamic behaviors, illustrating how the output of the Intentional Architecture is converted for the generation of a movement that could be consistent with the goals of the agent, by means of composition of motor primitives.
In chapter 4 we describe the Intentional Distributed Robotic Architecture (IDRA) project, the implementation of the Cognitive Architecture described in chapter 3. A technical description of the program is provided, as well as the guidelines to use the software.
In chapter 5 we describe the experimental activities and results obtained by the presented system. Two experiments are described, each testing the main features of the architecture: goals development and complex movements generation.
Chapter 6 contains the conclusions and future development: we summarize our goals, our evaluations of these results and possible future developments of the project.
In appendix A we present some code excerpts of IDRA, with a brief description.
In appendix B we show the library that we implemented for the execution of the Independent Component Analysis (ICA) algorithm, as well as a brief theoretical description of the algorithm.
In appendix C we show the library that we implemented for the execution of the K-Means clustering algorithm, as well as a brief theoretical description of the algorithm.
In appendix D we briefly show the Behavior Simulation project, a software we developed to test the functionalities of Dynamic Behaviors before their implementation in the Motor System.

Chapter 2

State of the art

"Artificial Intelligence is whatever hasn't been done yet."

Larry Tesler

2.1 The Amygdala-Thalamo-Cortical model

Several studies have shown the importance of the amygdala-talamo-cortical model in the cognitive development [11, 12].

The cortex is the external part of the brain; it is divided in several sectors (primary visual cortex, posterior parietal cortex, primary motor cortex, etc.), and it receives signals from the sensory organs. Although most of the sectors of the cortex receive input from a specific source, different studies have proven that different areas can properly react to different stimuli sources [5]. Therefore the whole cortex is composed from the same kind of cells, and it is able to respond, store and adapt to different kind of stimuli. Moreover, the generic mechanism used by the cortex is what allows the human beings to adapt to problems that did not exist in nature, and to learn how to deal with them.
The main difference between an usual computer system and the brain, and the reason why the latter is more efficient at solving some kind of problems - e.g. ball catching, face recognition, etc. - lies in the fact that the brain does not compute everytime for this kind of problems. Instead, the cortex has been proven to act as a memory bank, and the brain use to compare the incoming data with memories from past situation, in order to find analogies, patterns, and invariant representations, and act accordingly [4]. Moreover,

it has been proven how the cortex stores these patterns in sequences, and how sequences can be recalled in an auto-associative fashion, thus creating the so-called stream of thoughts.

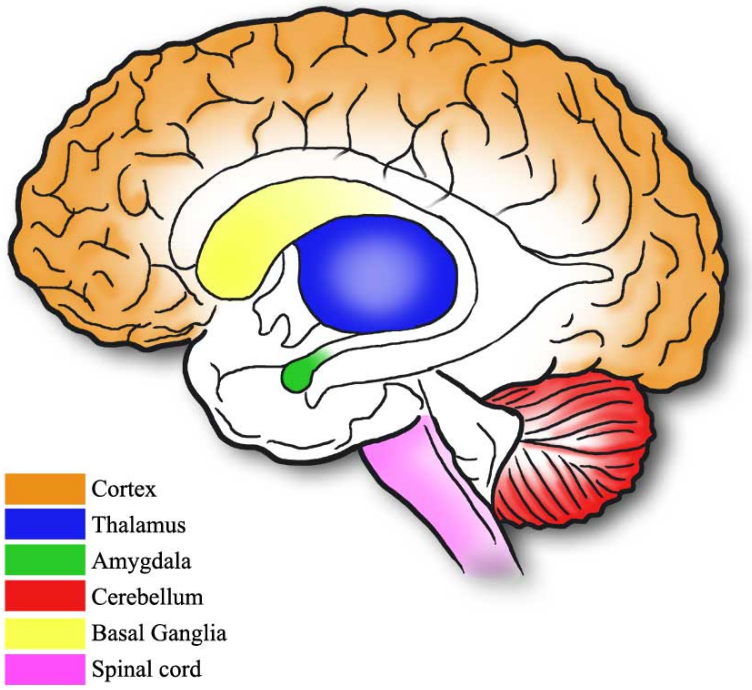


Figure 2.1: The brain areas of interest.

The thalamus is the largest component of the diencephalon, one of the most internal regions of the brain. It is the primary site of relay for all of the sensory pathways, except olfaction, on their way to the cortex. The thalamus plays a central role for the mammals in the development of new motivations and goals, as well in the choice of what goal to pursue. The thalamus is "a central, convergent, compact miniature map of the cortex" [13]. The thalamus is partitioned into about fifty segments, which do not communicate directly with each other. Instead, each one is in synchronized projection to a specific segment of the cortex, and it is receiving a projection from the same segment. Therefore, while the cortex is concerned with data processing, storing and distribution, the thalamus determines which goals have to be pursued [14]. Furthermore, each pair of cortex and thalamus sections seem to be intensively communicating. The cortex is believed to be

the main memory storage allowing consciousness and cognition, while the thalamus would determine how incoming stimuli are relevant with respect to actual goals and to the developing of new goals. Thanks to the thalamo-cortical model the brain is able both to learn how to achieve goals and which goals have to be pursued.

The amygdala is an almond-shaped group of nuclei located deep within the medial temporal lobes of the brain of complex vertebrates. It seems to be heavily connected to the cortex area; it is involved in the generation of somatosensory response on the basis both of innate and previously developed goals, and of sensory informations [15]. In particular, the amygdala seems to be an essential part in social and environment cognition in order to guide social behaviors.
The amygdala has a key role in the recognition of the emotions and in the generation of an adequate response. Evidence shows that this response is independent from the test subject [16]. Thus one of the principal tasks of the amygdala is to generate new goals taking advantage of hardwired criteria.

2.2 The movement generation via motor primitives

The cerebellum is supposed to have functionalities and structures similar to the classical perceptron pattern of a classification device. The cerebellum has an extended network of various types of neurons, giving different abilities including motor learning and motor coordination, thanks in particular to the Purkinje response cells, which can learn fast and that are able to easily distinguish patterns [17].
Other studies have shown the importance of the basal ganglia in the motor generation of the movement [18]. Basal ganglia and cerebellum seem to create two different sets of loop circuits with the cortex, both dedicated to different features of motor learning; both are independent and in different coordinates. Furthermore, there has been recently proposed a functional dissociation between the basal ganglia and the cerebellum: the former is implicated in optimal control of movements, that is optimization of costs and rewards obtained by the execution of a specific movement, and the latter seems to be able to predict the sensory consequences of a specific movement through the use of internal models [19].
The spinal cord is the lower caudal part of the nervous system; it receives and processes sensory information from the various parts of the body, and

controls the movement of the muscles and the joints of the main parts of the body [11]. It acts as a bridge between the body and the mind: it receives electrical signals input from the peripheral nervous system, and sends them to the central nervous system. In the same way, receives signals from the neurons in the cerebellum and the cortex and processes them before sending instructions to the specific muscles.
Additional studies show how the cortex is involved in motor generation, in particular the cortico-striatal circuit [20].

Several studies have shown how complex movements are generated from the combination of a limited number of waveform modules, which are independent from the considered muscle, its speed, and its gravitational load; these slighting varying modules control the active muscles in order to produce kinematic changes [21]. These studies have found how just five components can account for 90% of the muscles activations.
It was suggested that the nervous system do not need to generate all the muscles activity patterns, but to generate only a few basic patterns and combine them accordingly, to generate a specific muscle activation, through modulation of phasic component and magnitude of these primitives [22]. Different muscles performing different tasks showed different changes in patterns activities. This model can be represented by an oscillator that produces the output frequency basing on the input signal, a set of nonlinear functions that shape the oscillator output into the set of patterns, and a weighting functions for the generation of the muscle activity pattern [23].

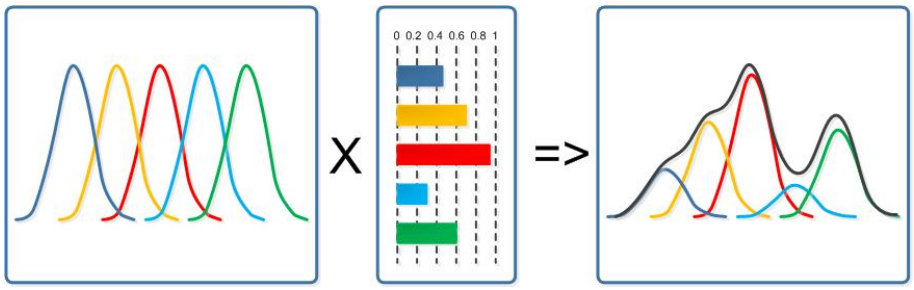


Figure 2.2: Motor primitives combination

A motor primitive is a specific neuronal network, found in the spinal cord, that generates a specific motor output by sending a definite signal to muscles for activations [24]; each primitive has a basic activation pattern (in function of time), with different weights and distributions for each muscle;

thus, the muscular activity during movement is determined by few basic patterns, independent from locomotion mode, direction or speed [25].

The Dynamic Movement Primitives - that is, a formulation of movement primitives via autonomous nonlinear differential equations - have been successfully used in a series of robotic applications, and they can be employed with supervised learning and reinforcement learning techniques in order to optimize trajectory and energy criteria [26]. Various application involved skill learning from supervised learning on a robot, or biped walking from imitation of an human, or the generation of rhythmic movement patterns using a nonlinear oscillator control [27].
However, we separate ourselves from this kind of experiments, which use learning methods commonly applicable to task-based robots; we will in fact use motor primitives in motivation-based robots, as we will see in the next chapters.

2.3 An intermediate level of cognition

The amygdala-thalamo-cortical model we addressed so far is the basis of the biological inspiration of the first part of our architecture. In this part we do not focus on high-level motor skills, nor on high level of reasoning and planning. We will instead focus on an intermediate level of cognition that allows mammals and humans to be aware of the surrounding environment and then interact with it for most basic behaviors.
Consciousness has been already suggested to be a product of an intermediate level of cognition [28]. The awareness of the surrounding environment is supposed to be not a direct product of sensations and other sense-data, the "phenomenological mind", nor a product of a high level conceptual thoughts, the "computational mind", but to be a product from several intermediate levels of representation [29].
This middle level has some interesting features related to consciousness: it underlines how we can interpret the surrounding environment and react to this awareness without the need for high-level conceptualizations nor complex motor controls, therefore solving the grounding problem of a semantic interpretation of a formal symbol system that is intrinsic to the system itself [30]. In particular, as we will see in the next chapters, we will deal primarily with the categorical representation, that are described as "learned and innate feature detectors that pick out the invariant features of object and event categories from their sensory projections".

Most robots - and their related intelligence - are designed with a specific task or a set of tasks. Some robots are equipped with some sort of skills that allow them to interact with the environment and in particular with humans. Most of these robots have a selection of preset behaviors that allow them to tune their actions according to past experience; they are reactive to their surroundings in order to reach their goals [2].
Some of these robots are not just able to tune their behaviors to the situation: they are also able to act on their own. While the behavior-based robots explore the environment in order to optimize their actions for reaching a predefined goal, these kind of robots must be able to explore the environment in order to find new goals on their own. They must be curious to explore their environment, and once explored, they must be able to do something new according to their abilities as well as to their experience. We call this kind motivational-based robotics [31], also known as epigenetic robotics, or developmental robotics [3].
Behaviors-based robots make use of motivations as well, but these are fixed, hardwired at design time; on the contrary, developmental robots must be able to perform actions following goals that were not present at design time. To make a comparison with human beings, the need for food is a hardwired criteria, and even a child will try to obtain it in the ways his known behaviors allow him to. At first he will scream until his mother will feed him; eventually he will learn to reach cutlery with his arm in order to bring food to his mouth.

Chapter 3

Cognitive Architecture

"They learn to speak, write, and do arithmetic. They have a phenomenal memory. If one read them the Encyclopedia Britannica they could repeat everything back in order, but they never think up anything original. They'd make fine university professors."

Karel Čapek, R.U.R.

3.1 Intentional Architecture: cognitive development

Cognitive development is the construction of thought processes, including remembering, problem solving, and decision making, from childhood through adolescence to adulthood. It refers to how a person perceives, thinks, and gains understanding of his or her world through the interaction of genetic and learned factors. Among the areas of cognitive development we can find information processing, intelligence, reasoning, language development, and memory [1].

The Intentional Architecture deals with the cognitive development of the agent, analyzing inputs and allowing it to develop new goals. The architecture is basically a net of linked modules, simulating connections and interactions between the three cerebral areas which allows the cognitive development in humans: the cerebral cortex, the thalamus and the amygdala. The modules composing the net are the Phylogenetic Module (amygdala) and the Intentional Module, which in turn is composed by the Categorization Module (cerebral cortex) and the Ontogenetic Module (thalamus). The

Intentional Architecture is composed by one Phylogenetic Module and several layers of Intentional Modules. Intentional Modules are linked in various ways, also with feedback connections, while the Phylogenetic Module broadcasts its signal to all the Intentional Modules, without receiving data back by them. This kind of structure can simulate the interaction of the three areas in human brain: different areas of thalamus can communicate with some respective areas of the cerebral cortex, which collects all the information coming from the thalamus, but different areas of the thalamus cannot communicate between them. The amygdala sends its information both to the thalamus and the cerebral cortex (Figure 3.1).
The input of the net is composed by various sensor types (video, audio, tactile, etc.), elaborated by some filters and sent to the Intentional Modules of the first layer. The output of the net is composed by a vector, representing the neural activation generated by sensory input, and by a signal, representing how much the actual input satisfies both hard-coded goals and new developed goals.
Thanks to the net of Intentional Modules, the Intentional Architecture can autonomously develop new goals (through the Ontogenetic Module) starting from hard-coded ones (provided by the Phylogenetic Module), which represent the "innate instincts" of the agent.

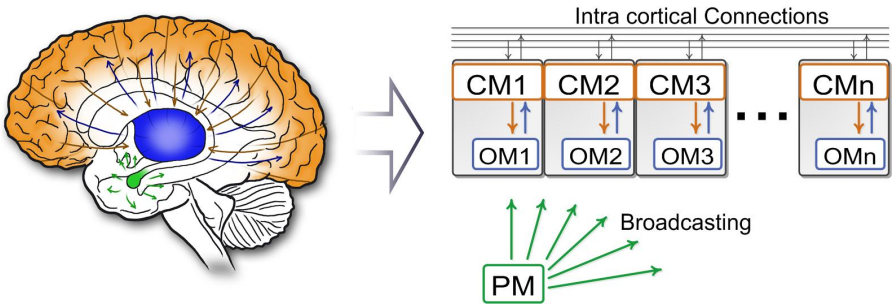


Figure 3.1: From the brain to our architecture

3.1.1 Input processing: sensory integration

Sensory integration is the neurological process that organizes sensation from a person's own body and the environment, thus making it possible to use the body effectively within the environment. Specifically, it deals with how the brain processes multiple sensory modality inputs into usable functional outputs. For some time it has been believed that inputs from different sensory

organs are processed in different areas of the brain [32]. The communication within and among these specialized areas of the brain is known as functional integration [33, 34, 35]. Newer researches had shown that these different regions of the brain may not be solely responsible for just one sensory modality, but they could use multiple inputs to perceive what the body senses about its environment. Sensory integration is necessary for almost every activity that we perform since the combination of multiple sensory inputs is essential for us to comprehend our surroundings.

The input of the Intentional Architecture comes from several sensors of the agent. Just like the information that are sent to our brain, the inputs of the agent are processed to give the architecture the same type of data and to perform sensory integration. In this way, the architecture does not depend on the type of input, while every kind of sensory information (video, audio, tactile, etc.) can be processed by the architecture in the same way. Furthermore, the Intentional Architecture can perform sensory integration by sending multiple sensory data to the same Intentional Module.
The data coming from every sensor is converted into arrays of different size and enclosed in a standard structure; every block of information is then sent to the first layer of Intentional Module to start the computation (Figure 3.2).

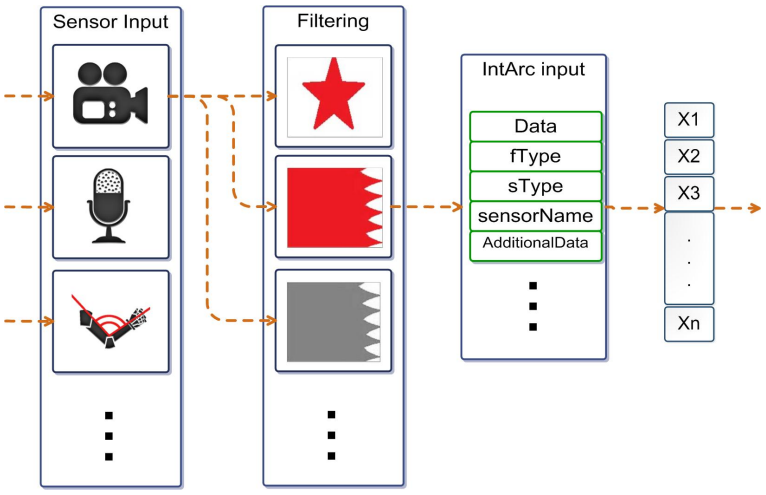


Figure 3.2: Sensory integration in our architecture

3.1.2 Phylogenetic Module: innate instincts

In biology, phylogenetics is the study of evolutionary relation among groups of organisms, which is discovered through molecular sequencing data and morphological data matrices. The result of phylogenetic studies is the evolutionary history of taxonomic groups: their phylogeny.
Phylogenetic processes contribute to the adaptation of an organism behaviors to the environment through the production of instincts [36]. An Instinct is the inherent inclination of a living organism toward a particular behavior, i.e. an impulse or powerful motivation from a subconscious source. The part of the brain that is associated with these types of reactions is the amygdala [37]. Different parts of the brain receive different signals and make them consciously known. The auditory cortex, for example, is responsible for hearing. The amygdala has its own set of "receivers" for sensory intake, and can retrieve information from the environment and take a decision about what to do, before a person could consciously think about it [38].

We implemented instincts in the system as hard-coded goals, in the Phylogenetic Module. this module contains embedded information and criteria in order to permit the bootstrap of the system. Phylogenetic Module tells the agent what is relevant according to its hard-coded instinctive functions. The input of this module comes from sensors; each sensory information is then processed by instinctive functions; each function processes only a certain input type, according to the instincts associated to that specific type (e.g. input from video sensors is processed only by instinctive functions related to video input). The output of this module is the phylogenetic signal, which tells how much the incoming stimulus is important according to the a priori stored criteria. The more considerable the stimulus, the higher the generated phylogenetic signal (for our purposes, as we will see in the next chapters, this value will always be normalized between zero and one).

3.1.3 Intentional Module: neuroplasticity

Plasticity, or neuroplasticity, is the lifelong ability of the brain to reorganize neural pathways based on new experiences.
Neuroplasticity can occur in different levels, ranging from cellular changes (e.g. during learning) to large-scale changes involved in cortical remapping (e.g. as consequence of response to injury). As we learn, we acquire new knowledge and skills through instruction or experience. Scientific research has now demonstrated how substantial changes can occur in the lowest neocortical processing areas, and how these changes can profoundly alter

the pattern of neuronal activation in response to experience [39]. Furthermore, experience can actually change both the brain's physical structure (anatomy) and functional organization (physiology) [40]. Neuroplasticity has replaced the formerly-held position that the brain is a physiologically static organ, and explores how and in which ways the brain changes throughout life into adulthood [39]. In order to learn or memorize a fact or skill, there must be persistent functional changes in the brain, and these changes represent the new knowledge. The ability of the brain to change with learning is what is known as neuroplasticity.

Neuroplasticity is a fundamental feature to give an agent the ability to learn new goals, so we designed the Intentional Architecture as a net of basic modules which can adapt to changes in sensory inputs and connections.
The Intentional Module is the basic building block of the Intentional Architecture. This module has to be able to adapt to various sensor modalities, as well as to combine with other instances of the same kind in some simple way. The Intentional Module can easily adapt to changes in sensory input. If we send to an Intentional Module some input from a video sensor, the Intentional Module will specialize to that type of input; however, if we decide to change some connections and send to this specialized Intentional Module a different type of input, e.g. an audio input, it will lose its specialization in video input and gradually adapt to the new sensory input.
An Intentional Module has other important features: since it is the basic unit of the architecture, it must be able to self-develop new goals and motivations. This feature of the Intentional Module has already been put to test in a very simplified robotic setup, aiming at developing new motivations and controlling the gaze of a camera towards unexpected classes of visual stimuli [31, 41, 42].
According to what we said, we can summarize the main objectives of the module:

- it must be able to adapt to any kind of input;
- it must be able to learn to categorize incoming stimuli;
- it must be able to use acquired categories to develop new criteria to categorize;
- it must be able to interface smoothly with similar modules and give rise to a hierarchical structure.

In order to satisfy all this objectives, the Intentional Module is composed

by two simple structures: the Categorization Module, which performs categorization, and the Ontogenetic Module, which can develop new goals.
The Intentional Module works as follows. At the beginning, both the Ontogenetic Module and the Categorization Module are empty. Incoming data, of any type and size they are, are sent to the Categorization Module, which has the important function of receive raw data and then to develop higher order categories. Once the categories have been created, they are sent to the Ontogenetic Module. This module perform Hebbian learning on incoming data to develop new goals, and returns a signal based on how much this new goals are satisfied. This signal is called ontogenetic signal; a high value of ontogenetic signal corresponds to high satisfaction of the developed goals. The Intentional Module receives as input also the signal from the Phylogenetic Module, and returns as output the maximum between the signal of the Phylogenetic Module and the Ontogenetic Module. This outgoing signal is called relevant signal; the Intentional Modules also sends as output the categories created by the Categorization Module (Figure 3.3).

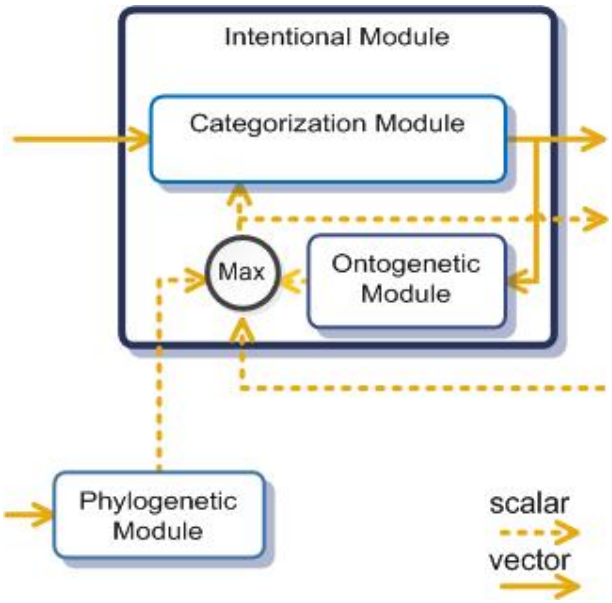


Figure 3.3: Intentional Module

Categorization Module: cognitive autonomy

Most of the actual information processing in the brain takes place in the cerebral cortex. The cerebral cortex is divided into lobes, and each of them has a specific function [11]. For example, there are specific areas involved

in vision, hearing, touch, movement, and smell. Other areas are critical for thinking and reasoning. Although many functions, such as touch, are found in both the right and left cerebral hemispheres, some functions are found only in one of the two cerebral hemisphere. For example, in most people, language abilities are found in the left hemisphere. The sensory areas are the regions that receive and process information from the senses [11]. Parts of the cortex that receive sensory inputs from the thalamus are called primary sensory areas. One of the main feature of the cerebral cortex is the ability to adapt to stimuli, whatever is their nature [5, 4]. Cerebral cortex presents a high degree of cognitive autonomy, so each cortical area is capable to compute any type of incoming data [43, 44, 45].

The Categorization Module represents the cerebral cortex of the Intentional Architecture. It receives the input from sensors or from other Intentional Modules and perform categorization. in order to do this operation, the input is elaborated twice: first with Independent Component Analysis (ICA, see Appendix B), then with a clustering algorithm such as K-Means (see appendix C).

Independent Component Analysis allows the module to generalize the input representation regardless to the type of incoming stimuli. In an early-development stage, independent components are extracted from a series of input through the ICA algorithm. After this training stage, the input is projected in the bases space (i.e. the previously extracted independent components), in order to reduce the dimension of the data and to get a general representation:

$$\overline{W} = IC \times \overline{I} \tag{3.1}$$

Where $\overline{W}$ is the resulting vector of weights, IC is the matrix of independent components and $\overline{I}$ is the input vector.
This results in a vector of weights where clustering is performed, using K-Means algorithm. Clustering is a good and simple way to get the neural code of the information. Neural coding consists in the translation of a stimulus into a neural activation; according to this, a net of neurons can represent every kind of incoming information. Basing on the theory that sensory and other information is represented within the brain by networks of neurons, it is supposed that neurons can encode any type of informations [46]. During clustering, each vector is assigned to an existing cluster, if the distance from existing clusters is below a previously-set threshold, otherwise a new cluster is created, using the newly acquired vector.

The output of the Categorization Module is a vector containing the activations of clusters, which depend on the distances of the input data from the center of each cluster (namely a category). This vector corresponds to the activation of a neuron centered in each cluster:

$$y_i = \rho(x, C_i) \tag{3.2}$$

Where y_i is the distance of the actual input from the center of the cluster i, x is the input and C_i is the center of the cluster i. For our purposes, the values are normalized between zero and one.
A new category is created by the categorization module depending on the value of the relevant signal computed by the Intentional Module. This way, only relevant inputs are categorized, so that the module saves only meaningful information (Figure 3.4).

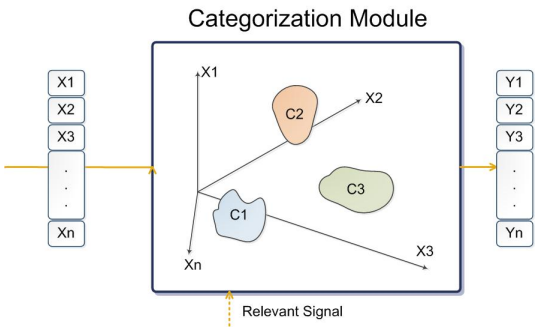


Figure 3.4: Categorization Module

Ontogenetic Module: goals generation

The thalamus is a structure of the brain composed by four parts, or nuclei, situated between cerebral cortex and midbrain. Functionalities of the thalamus include elaboration and relay of input and motor signals to cerebral cortex, as well as regulation of consciousness, sleep, and alertness [47]. Furthermore, thalamus is a "miniature map" of the cerebral cortex [13]. All thalamic nuclei, with the exception of the reticular thalamic nucleus, project primarily to the cerebral cortex. Additionally, each portion of the thalamus receives a reciprocal connection from the same portions of the cerebral cortex whereby the cortex can modify thalamic functions. These connections are more data intensive from cerebral cortex to thalamaus, while backward connection going from thalamus to cortex are weaker [48]. This close connection between thalamus and cortex and their interplay, as well as the implication

of thalamus in consciousness, led to the idea that goals generation is indeed spread everywhere in cerebral cortex and it is obtained by the interaction between thalamus and cortex.

The Ontogenetic Module represents the thalamus of our system and it is the core part of the goals development of the Intentional Module. the name ontogenetic derives from ontogeny, namely the origin and the development of an organism; it covers, in essence, the study of an organism lifespan, unlike phylogeny which covers instead the evolution of the entire species of an organism. Just like in the brain thalamus and cerebral cortex present strong connections, Ontogenetic Module is closely connected to the Categorization Module. It uses the categories computed by the Categorization Module and a Hebbian learning function to develop new goals.
The values of neural activations, provided by the Categorization Module, are evaluated using a vector of weights, and the resulting ontogenetic signal is the maximum value between the evaluated neural activations:

$$o_s = \max_i(y_i w_i) \tag{3.3}$$

Where o_s is the resulting ontogenetic signal, y_i is the activation of neuron i and w_i is the vector of weights associated to neuron i. Each weight is normalized between zero and one (Figure 3.5).

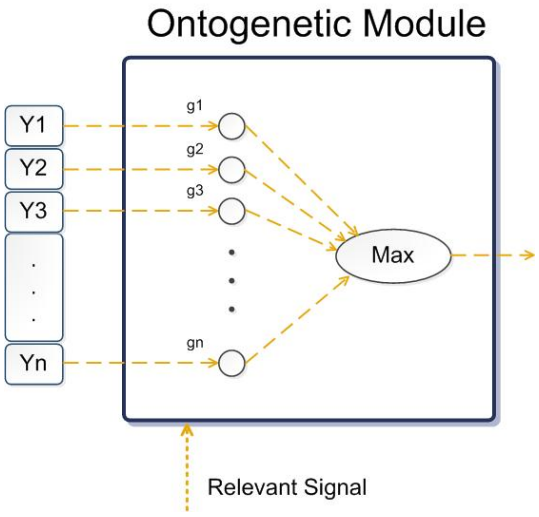


Figure 3.5: Ontogenetic Module

The ontogenetic signal strongly depends from the vector of weights used

to evaluate the input. These weights are updated at each iteration, using a Hebbian learning function:

$$w_i = w_i + \eta(h_s y_i - (w_i y_i^2)) \tag{3.4}$$

In this equation η is the learning rate, while h_s stands for the hebbian signal, which is a control signal coming from the Intentional Module (it could be the relevant signal itself). A threshold is fixed, so that if a weight is beyond or equal the threshold value, its value is always set to one. This is done to strenghten signals that have already proven to be good enough.
The output of the Ontogenetic Module is the ontogenetic signal, whose value represents how much the actual input state satisfies the new goals developed through the hebbian learning process.

3.2 Motor System: movement generations

In order to accomplish its goals, the agent should be able to move and interact with the environment. The problem of moving and acting in a smart way, according not only to hard-coded goals, but also to new developed goals, is not trivial. We suggest a solution to this problem based on Dynamic Behaviors (for movement evaluation) and on the concept of motor primitives (for movement generation).
The input of this motor part comes from the Intentional Architecture, and it is composed by a vector of neural activations and a relevant signal. The vector, representing the state of the environment in a high level of abstraction, is clustered using K-Means algorithm. The output of the clustering is the cluster corresponding to the current state; this information is used with a State-Action table to choose the best movement to do, according to the relevant signal. The movement is composed by a linear combination of primitives, and it is sent as output to the agent. In order to execute the movement, the agent uses the output of this motor part to compute joints values of its actuators.

3.2.1 Motor primitives: elementary movements

In recent years different lines of evidence have led to the idea that motor actions and movements, in both vertebrates and invertebrates, are composed of elementary building blocks, called motor primitives. Motor primitives might be equivalent to “motor schemas”, “prototypes” or “control modules” [49, 50, 51].

Motor primitives could be transformed with a set of operation and combined in different ways, according to well defined syntactic rules, in order to obtain the entire motor repertoire of motor actions of an organism. At neuronal level, a primitive corresponds to a neuron assembly, for example, of spinal or cortical neurons [9, 52].
Studies on the motor system suggests that voluntary actions are composed by movement primitives, that are bonded to each other either simultaneously or serially in time [6, 7, 8, 9, 10].

Following this idea, we use this concept of motor primitives to create muscular activations that allow the agent to perform a complex movement. A motor primitive could be seen as the activation of a muscle during time. The higher the value of the primitive, the stronger the muscle activation, which will bring to a faster execution of the movement. Activating different muscles in time, a complex movement can be performed. We implement primitives as Gaussian functions delayed in time:

$$p = e^{\frac{-(x-c)^2}{2\sigma^2}} \tag{3.5}$$

Where c is the center of the muscular activation of the primitive p. We chose the bell-shaped profiles of Gaussian function for primitives according to biological evidences: when humans move their limbs from one position to another they generally change joint angles in a smooth manner, such that angular velocity follows a symmetrical, bell-shaped profile [53]. To generate a complex movement, primitives are linearly combined, producing a muscular sinergy:

$$m = \sum_i w_i p_i \tag{3.6}$$

Each weight w is initially randomly generated during movement generation, thus creating a a random complex movement to be performed (See Figure 2.2).

3.2.2 Dynamic Behaviors: complex movements

In human brain motor cortex is involved in planning, control and execution of movements. Motor cortex is composed by several parts, each one contributing in generation of movements; however, the main contributor to generating neural impulses to pass down to the spinal cord and control the execution of movement is the primary motor cortex. Scientific evidence suggests that each neuron in the primary motor cortex contributes to the force

in a muscle: the more activity in the motor cortex neuron, the more muscle force [54]. Furthermore, the primary motor cortex is somatotopically organized, which means that stimulation of a specific part of the primary motor cortex elicits a response from a specific body region (Figure 3.6).

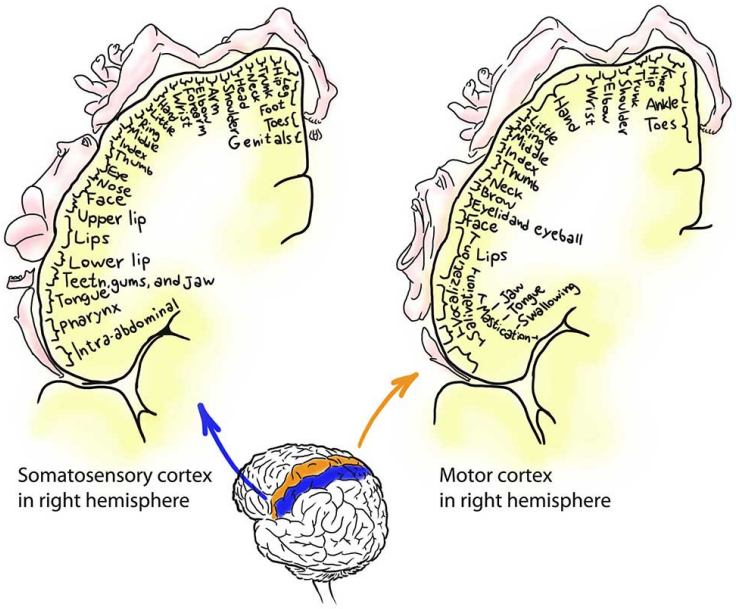


Figure 3.6: The Somatosensory Cortex and the Motor Cortex

According to the fundamental role of the motor cortex in movement generation, the approach to this task should be similar to the one used for categorization; more precisely, we need a neural code of information like the one computed by the categorization module. Following this idea, the first step in order to perform a movement is clustering (Figure 3.7), like in Categorization Module. Clustering is performed by K-Means algorithm (see Appendix C), the same used in the Intentional Module. However, in this case we don't need to create new categories, so clusters are defined in a previous training phase. We need the cluster representing the current state, namely the part of the primary motor cortex that is stimulated by the current state. This approach respects the idea that the same neural activation produces the same muscular response.

Movement generation is not the only goal to be accomplished; we need something able to select the best movement according to the current state of the environment, depending on the goals we want to satisfy. Dynamic

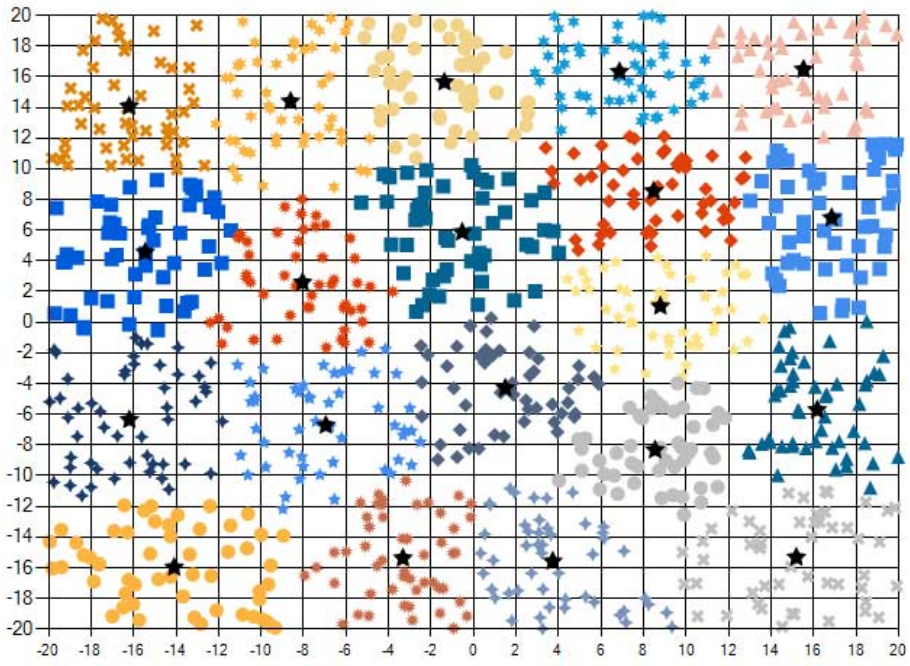


Figure 3.7: An example of clustering

Behaviors allow the agent to perform the best movement, given the state of the environment and the relevant signal coming from the Intentional Architecture, as well as the ability to learn the best movement to execute in an unknown situation. The relevant signal depends on the ontogenetic signal, coming from the thalamus of the system, and the phylogenetic signal, computed by the amygdala of the system; its use in the motor development of the agent is based on scientific evidences, confirming the fundamental role of the basal ganglia, a group of nuclei in the brain containing the amygdala, and of the thalamus in the process of motor learning [20]. Moreover, studies on monkeys prove that lesions on the thalamus seriously impair their motor learning capabilities [55].
Each Dynamic Behavior is composed by a list of actuators of the agent that have to be moved in order to satisfy a goal. If we want the agent to look to a ball, for example, we can create a Behavior linked to the actuators which control head yaw and pitch angles. The Dynamic Behavior selects the set of movements to be executed from the actuators in order to get the best relevant signal as a response from the Intentional Architecture. The computation of the best movement to be performed is based on a State-Action table (Figure 3.8); the table associates a state and a movement to a relevant

signal: when the system is in a certain state and performs a movement, the relevant signal generated by this state-performed movement combination is stored in the table. The policy for movement selection for each input state follow these rules:

- if there is a movement associated with a relevant signal above a defined threshold, that movement is selected;
- otherwise, if there is a movement not performed yet, that movement is selected;
- otherwise, if all movements have already been performed at least once, and no one is associated with a relevant signal above a defined threshold, a new random movement is added to the list.

Each movement is a set of weights to apply to primitives in order to obtain a muscular activation. Once a movement is selected, it is used to linearly combine primitives, and the resulting vector is sent to the agent. The agent can use this output to compute the correct joint values and move according to the information it has received.

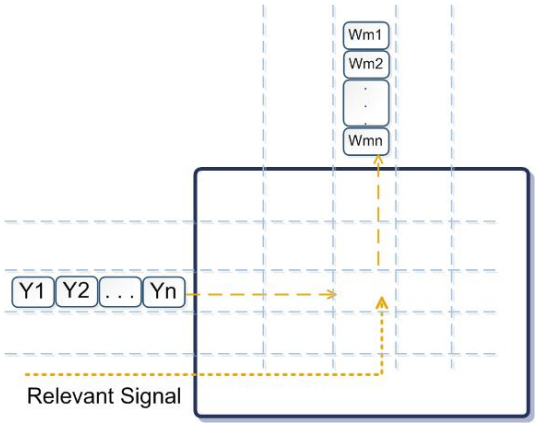


Figure 3.8: The State-Action table

Chapter 4

IDRA software architecture

"It - could - work!"

Doctor Frankenstein, Young Frankenstein.

The project IDRA - Intentional Distributed Robotic Architecture - is the main program reflecting the implementation of the Cognitive Architecture model here proposed.
It was specifically designed to be as modular as possible, in order to allow further addition of innate abilities in the Phylogenetic Module, new kinds of sensor's inputs, and new kinds of actuator's behaviors. Moreover, it is designed to be virtually able to adapt to every robot the user would like to use it with.

The project was developed in C# using Microsoft Visual Studio 2010, making extensive use of libraries and data standards that we specifically developed for this project. In the current state of the work the project is provided with C# implementation and XML config files for an Aldebaran NAO robot, and with a dummy robot useful for preliminary testing.

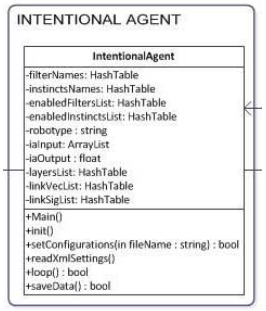
4.1 Design concepts

During the project design and development, we introduced some important distinctions and classifications between the various kinds of data flowing in the program, in order to represent at best the whole kind of data flowing in a bio-inspired model.
First of all, we have the *Sensor* data. This is the actual data coming directly from the sensors of the robot. Over the course of the program, the sensor data identifies the raw untreated signals coming directly from the physical sensors of the device: a RGB logpolar image sized with the maximum width and height of the camera, is sensor data. They represent the electrical signals coming from the biological senses.
Then, Sensor data is passed to various filters, obtaining the *Filter* data. these filters are several functions that process sensor data and extract the desired features: a center-cropped Canny edge image, a saturation signal, a soundwave with low frequency erased, are filtered data. They represent the data as it arrives from the senses to the cortex.
The filtered data is then sent to the *Instincts*. These are the functions implemented in the Phylogenetic Module, which respond to specific values of specific types of filtered data; a high phylogenetic signal may be caused by a high saturation signal, if the robot has an instinct for liking high saturated images. They represent the sensory data after the preprocessing performed by the amygdala.
Once the Intentional Architecture has produced its output, it is passed to the movements part of the architecture; here we have the *Behaviors*, that is the functions responding to specific output values of the Intentional Architecture, and ordering the robot to accomplish some kind of action; a function responding to a certain output from the IA making the robot walk toward a direction, is a Behavior. They represent the combination of motor patterns that can be found in the cerebellum, the spinal cord and the motor cortex.
Finally, when we talk about *Actuators*, we are talking about the actual motors of the robot. Over the course of the program, the actuators receive instructions from the Behaviors and initiate the physicals motors, the voice reproducers, the LEDs activation, etc. They represent the actual muscles of the body.

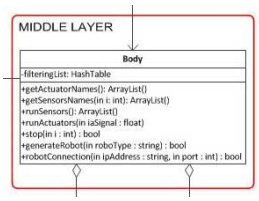
4.2 Software design

The architecture has been designed and implemented keeping in mind the principles of software engineering. Its modularity allows a wide scalability,

and easy maintenance of the code. In addition, the parts of which it is composed have been designed to resemble the biological architecture, although its structure will allow, in future, to be converted without too much difficulty to a version where each module works in parallel and asynchronous way.
In Figure 4.1 we can see the complete UML diagram of the architecture. Squares with white background represent different projects inside the same Visual Studio solution, while squares with dark background represent folders containing dll and XML files.
The architecture works in a straightforward way: in a normal single loop of the working process, first of all the architecture retrieves the sensors data from the robot; then it passes the data to the filters, and after their processing, the treated data is sent to the Intentional Architecture, in particular to the Phylogenetic Module and to the first layer of the Intentional Architecture net. After the net has performed its computation, the output is sent to the Behaviours, which determine the movement that have to be sent back to the robot.



The *IntentionalAgent* class is the starting point of our architecture; it is called by the graphic interface at the bootstrap of the application, and it instantiates both the Body and the Intentional Architecture classes. During the running loop of the application, the IntentionalAgent class serves as a bridge between the other two classes, in order to exchange their data.



The *Body* class, as the name suggests, receives inputs from the 'senses' (the robot class), transforms them (through various filters), and sends the 'electrical signal' to the 'nervous system', which is the Intentional Architecture. In a dual way, Body receives orders from the 'brain' and sends 'electrical signals' to the robot 'muscles'. Therefore, at the start of the working loop, the Body retrieves input signals from the specific robot class, through the RobotInterface.

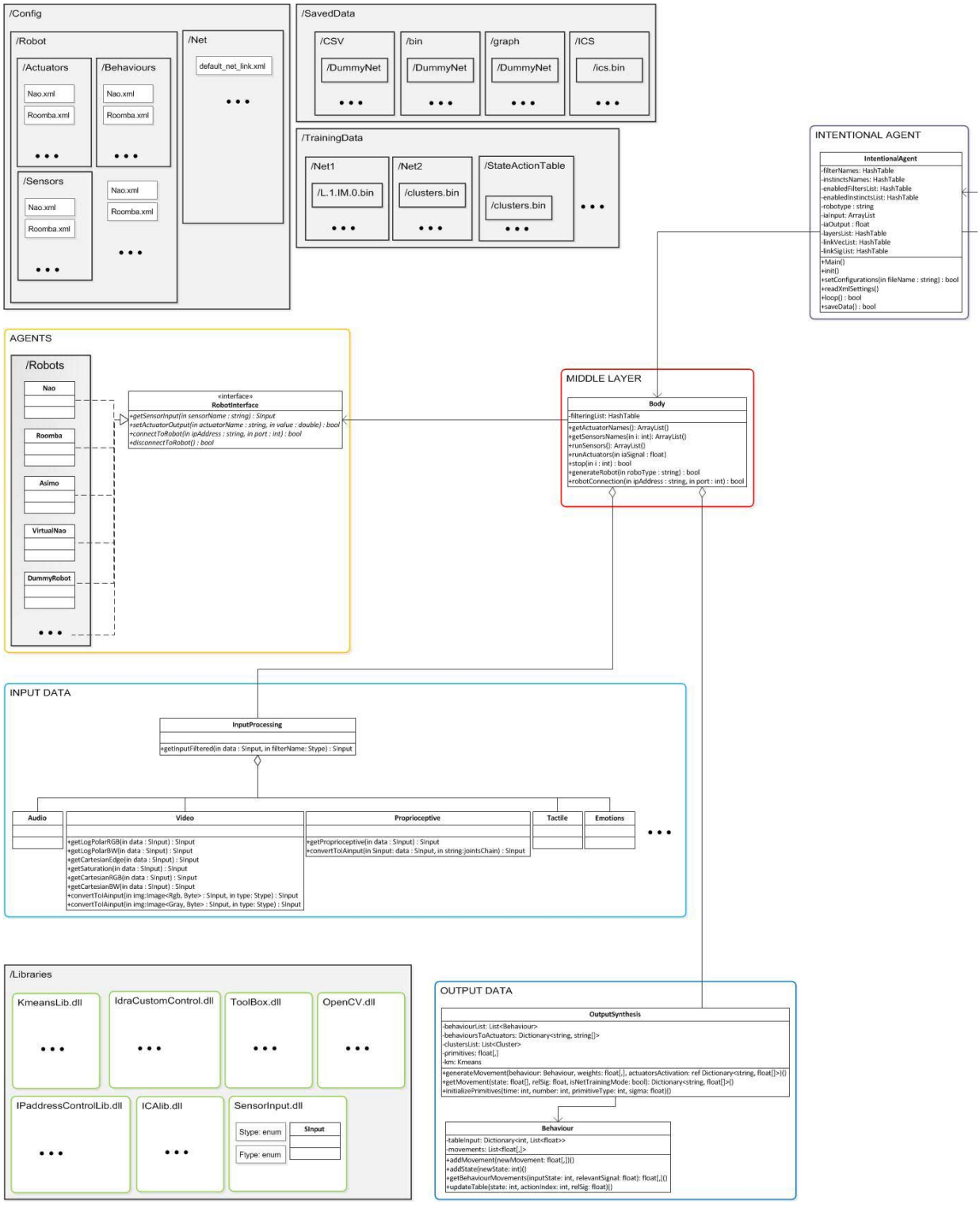
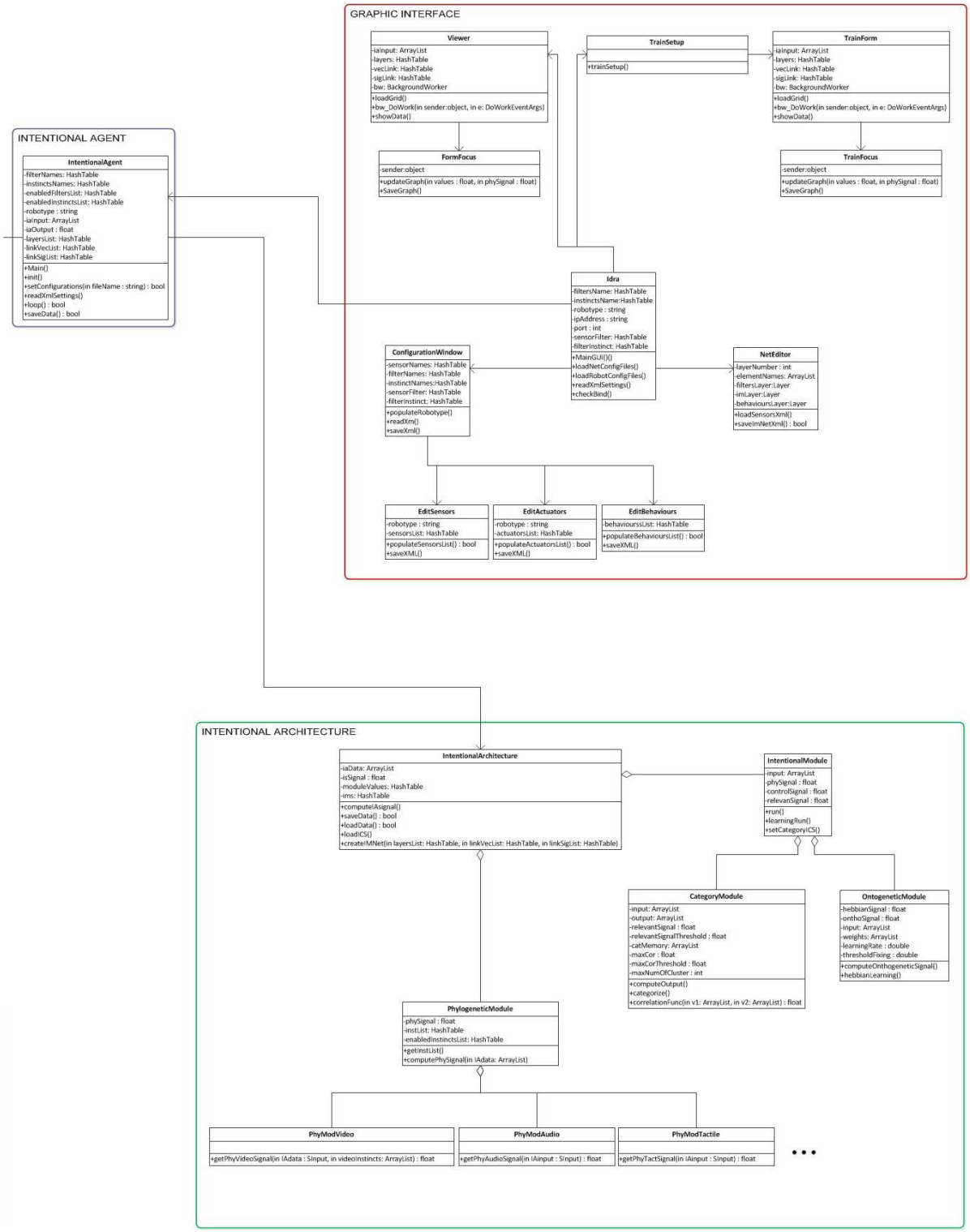
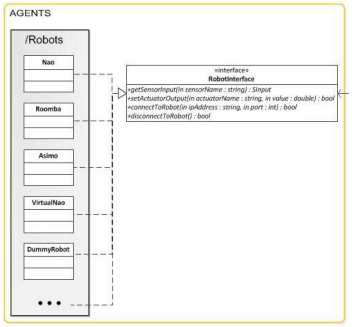


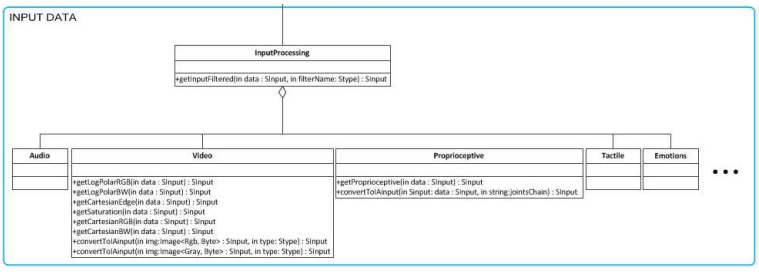
Figure 4.1: UML diagram of the architecture


GRAPHIC INTERFACE
Viewer
-iaInput: ArrayList
-layers: HashTable
-vecLink: HashTable
-sigLink: HashTable
-bw: BackgroundWorker
+loadGrid()
+bw_DoWork(in sender:object, in e: DoWorkEventArgs)
+showData()
TrainSetup
+trainSetup()
TrainForm
-iaInput: ArrayList
-layers: HashTable
-vecLink: HashTable
-sigLink: HashTable
-bw: BackgroundWorker
+loadGrid()
+bw_DoWork(in sender:object, in e: DoWorkEventArgs)
+showData()
FormFocus
-sender:object
+updateGraph(in values : float, in phySignal : float)
+SaveGraph()
TrainFocus
-sender:object
+updateGraph(in values : float, in phySignal : float)
+SaveGraph()
INTENTIONAL AGENT
IntentionalAgent
-filterNames: HashTable
-instinctsNames: HashTable
-enabledFiltersList: HashTable
-enabledInstinctsList: HashTable
-robotype : string
-iaInput: ArrayList
-iaOutput : float
-layersList: HashTable
-linkVecList: HashTable
-linkSigList: HashTable
+Main()
+init()
+setConfigurations(in fileName : string) : bool
+readXmlSettings()
+loop() : bool
+saveData() : bool
Idra
-filtersName: HashTable
-instinctsName:HashTable
-robotype : string
-ipAddress : string
-port : int
-sensorFilter: HashTable
-filterInstinct: HashTable
+MainGUI()()
+loadNetConfigFiles()
+loadRobotConfigFiles()
+readXmlSettings()
+checkBind()
ConfigurationWindow
-sensorNames: HashTable
-filterNames: HashTable
-instinctNames:HashTable
-sensorFilter: HashTable
-filterInstinct: HashTable
+populateRobotype()
+readXml()
+saveXml()
NetEditor
-layerNumber : int
-elementNames: ArrayList
-filtersLayer:Layer
-imLayer:Layer
-behavioursLayer:Layer
+loadSensorsXml()
+saveImNetXml() : bool
EditSensors
-robotype : string
-sensorsList: HashTable
+populateSensorsList() : bool
+saveXML()
EditActuators
-robotype : string
-actuatorsList: HashTable
+populateActuatorsList() : bool
+saveXML()
EditBehaviours
-behaviourssList: HashTable
+populateBehaviourssList() : bool
+saveXML()
INTENTIONAL ARCHITECTURE
IntentionalArchitecture
-iaData: ArrayList
-isSignal : float
-moduleValues: HashTable
-ims: HashTable
+computeIASignal()
+saveData() : bool
+loadData() : bool
+loadICS()
+createIMNet(in layersList: HashTable, in linkVecList: HashTable, in linkSigList: HashTable)
IntentionalModule
-input: ArrayList
-phySignal : float
-controlSignal : float
-relevanSignal : float
+run()
+learningRun()
+setCategoryICS()
CategoryModule
-input: ArrayList
-output: ArrayList
-relevantSignal : float
-relevantSignalThreshold : float
-catMemory: ArrayList
-maxCor : float
-maxCorThreshold : float
-maxNumOfCluster : int
+computeOutput()
+categorize()
+correlationFunc(in v1: ArrayList, in v2: ArrayList) : float
OntogeneticModule
-hebbianSignal : float
-onthoSignal : float
-input: ArrayList
-weights: ArrayList
-learningRate : double
-thresholdFixing : double
+computeOnthogeneticSignal()
+hebbianLearning()
PhylogeneticModule
-phySignal : float
-instList: HashTable
-enabledInstinctsList: HashTable
+getInstList()
+computePhySignal(in iAdata: ArrayList)
PhyModVideo
+getPhyVideoSignal(in IAdata : SInput, in videoInstincts: ArrayList) : float
PhyModAudio
+getPhyAudioSignal(in IAinput : SInput) : float
PhyModTactile
+getPhyTactSignal(in IAinput : SInput) : float

The *RobotInterface* interface is called by Body when it has to instantiate the robot; Body refers to RobotInterface whenever it has to interact with the user-defined robot. Thus, any new robot class implemented by the user has to implement the interface methods in order to work with the rest of the architecture. In particular, RobotInterface defines four functions for robot connection and disconnection, for receiving its sensor input, and for sending commands to the actuators.

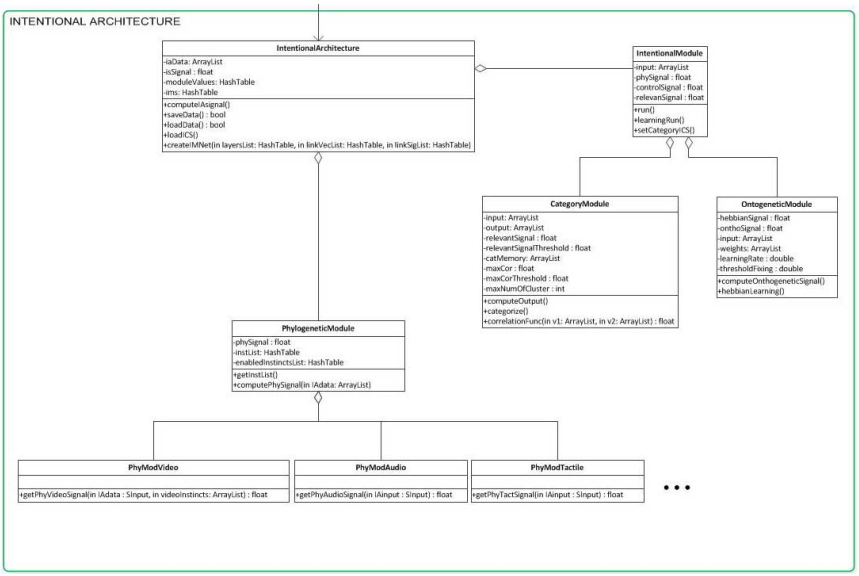


The *Robots* folder contains all the user-defined robots available, each one implemented through a CS file. Each file describes how the application can connect to and disconnect from the robot, how to retrieve data from each one of its sensors, and how to send instructions to each one of its actuators. This is beacuse most robots - commercially and not - available nowadays use their specific functions and API in order to activate their peripherals; this methods allow the architecture to abstract from the functioning of the specific robot.
For example, during the testing phase, we used the Aldebaran NAO robot. This machine uses the functions provided by the Aldebaran itself. These functions are contained in the *naoqidotnetvc90.dll* library, located in the *Libraries* folder; but any other robot would need to have its proper libraries added in this folder.
Once the Body has called the robot's functions, it passes the untreated data to the *InputProcessing* class, which contains all the filters used to process the input from the sensors, before sending it to the Intentional Architecture.



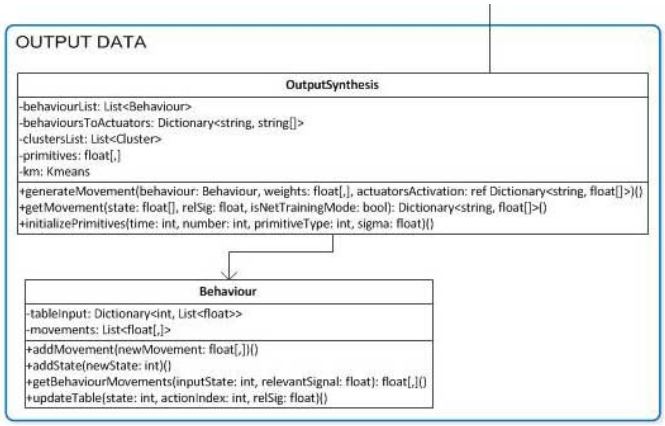
It is divided in many subclasses, one for each type of sensor (audio, video, proprioceptive, etc.); in particular, each subclass contains functions that allow to encapsulate each type of sensor into a specific data format, IntArc-

Type, which can be easily processed from the Intentional Architecture.
The *IntentionalArchitecture* class contains the Phylogenetic Module and the Intentional Modules net. This class receives the filtered input and sends it to the first layer of the net.
The *PhylogeneticModule* class is the Phylogenetic Module, which receives in input the filtered sensors data, and produces as output the phylogenetic signal. Notice that this module can receive input from different sources, and treat each one of them in a different way, and then broadcast the relevant signal to the whole network.

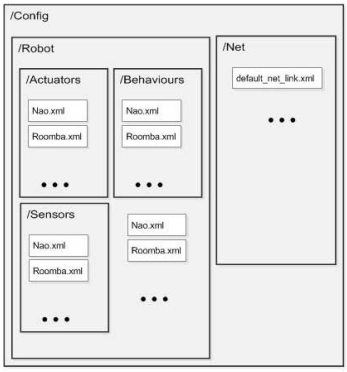


The *IntentionalModule* class contains a Categorization module, which stores the various categories, and an Ontogenetic Module, which receives the categories and performs the Hebbian learning.
Notice that the IntentionalArchitecture class, as well as all its subclasses, has a twin class in the *NetTrainingLib* Visual Studio project. This is a modified version of the Intentional Architecture, and it is used for the preliminary training of the IA net and of the motor part, as we will describe in the next section. The functioning of the classes contained in this project is completely analogous to the ones contained in the IntentionalArchitecture project.
Once the Intentional Architecture has finished its computations, it sends the Global Output Data and the Global Control Relevant Signal to the *OutputSynthesis* class, which contains all the Behaviors classes used to generate movements in the robot's actuators. This class contains a list of all the

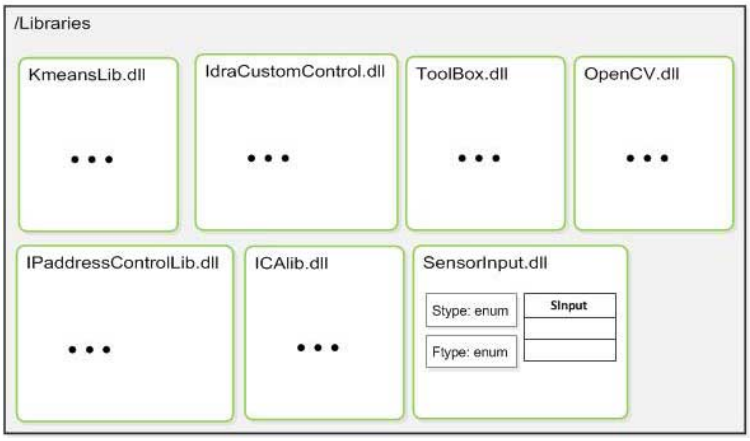
Behaviors instantiated, each one related to a list of Actuators.



Each *Behavior* class contains a state-action table, that allows the Intentional Architecture to choose the best movement that specific joints have to perform. Each row represents a specific state of the world, as indicated by the Global Output Data; each column contains the weights records that have to be applied to the motor primitives, and the class combines the primitives in order to generate the final muscle activation signal.
The OutputSynthesis class sends the output of the Intentional Architecture to each Behavior, which in return sends back the movements to apply to a specific Actuator; the class sends the total list of movements to the Body class, which in turn sends them to the Robot class, thus closing the loop.
In order to allow the user an easy configuration and combination of all the various modules present in the software, several XML config files have been designed and implemented; most of them are editable through the graphical interface, as described in Chapter 4.3. The *Robot* folder, for example, contains several XML files describing the name of the robot, its IP address and port used for the connection, the sensor-filter associations (which ones of the various filters has to be applied to each sensor's input), the filter-instinct associations (which ones of the various instincts has to be applied to each filtered input), and the behavior-actuators associations (which behaviors to use and which actuators to be used with a specific behavior).

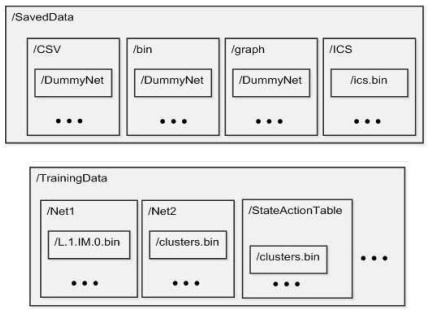


The *Sensors*, *Actuators* and *Behaviors* folders contain several XML files, one for each robot, and they describe a list of all the Sensors, a list of all the Actuators, and a list of all the available Behaviors for a specific robot, respectively. Notice that the lists in the Sensors and Behaviors are different from the analogous lists that can be found in the robot XML file: the robot could have many sensors and behaviors available, but the user can choose which of these to use. The Robot XML file is then supposed to have two subsets of the items in the lists contained in the Sensors and Behaviors XML files, which instead contain the complete lists of all the possible sensors and behaviors available.
Similarly, the *Net* folder contains the XML files describing all the modules to be considered in the intentional module net (filtered sensor data, Intentional Modules), and how all of this blocks are linked together.
The architecture makes use of various libraries, contained in the *Libraries* folder. Some of this libraries are external or provided by third parties: for example, we use the already cited *naoqidotnetvc90.dll* library, and also *Emgu* and *OpenCV* libraries, which are used for the image elaboration, in the InputProcessing class.



Furthermore, we make use of several custom libraries that we developed, in order to provide different functionalities to the architecture. Among these are the *IdraCustomControls.dll* library, which contains several custom controls used to represent a filtered input and an Intentional Module in the NetEditor window, in the TrainForm window, and in the Viewer window (see Chapter 4.3 for a description of these windows); the *ICAlib.dll* library, which contains various functions for the Independent Component Analysis computation (see Appendix B for more informations about ICA and its usage within our project); the *KmeansLib.dll* library, which contains various functions for the K-Means clustering algorithm computation (see Appendix C

for more informations about K-Means and its usage within our project); and finally the *SensorInput.dll* library, which contains various enum variables, i.e. Stype, which describes the sensor input type (video, audio, proprioceptive, etc.), and Ftype, which describes the filters type (LogPolarRGB, CartesianBW, etc.)
Moreover, this library contains the Sinput class, which is a generic object containing all various types of sensors inputs, formatted in a standard way, in order to be usable by Body regardless of the nature of the original input. For example, if the Sinput's type is the VideoInput subclass, the data structure will be composed of three integers (width, height, number of layers), a colorspace enum (RGB, BW), and a three-dimensional byte array (representing the image itself); if Sinput's type is intArcInput - the input accepted by the intentional architecture - it will be composed by an array of floats for the data, two enums indicating the original sensor and filter type, the sensor's name, and a generic object for additional data. All input from all kind of video sensor must be formatted according to this container structure, before being passed to Body.
Finally, we have two folders where we store all the output files of the architecture. The *saveData* folder contains all the data used and generated from the IDRA architecture, in various formats: in particular it contains a bin subfolder, a CSV (Comma Separated Values) folder, a graph folder, and an ICS (Independent ComponentS) folder.



The *trainData* folder, instead, contains all the data used and generated from the IDRA architecture during the training phase: in particular it contains several subfolders, one for each trained net and one for each trained robot, and each folder contains several BIN files containing data about the clustering and the categories for each module of the net. See Chapter 4.3 for a description of the training process.

4.3 User interface

Starting the program, the first window appearing is the IDRA window (Figure 4.1): From here the user can see the available robot configurations (XML

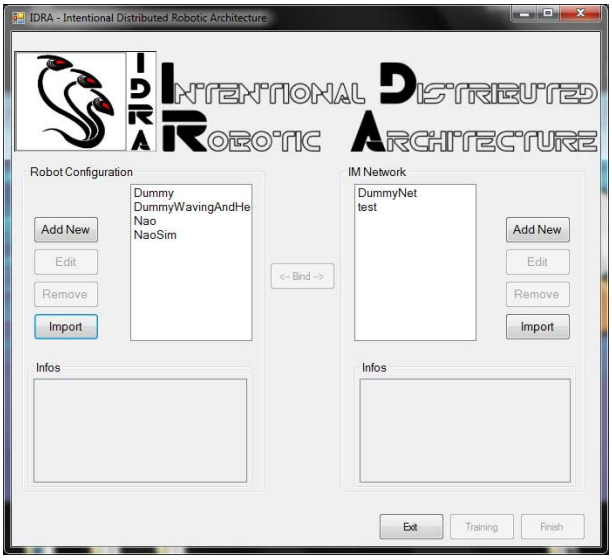


Figure 4.1: The IDRA window

files located in the /IDRA/Config/Robot folder) and the available Intentional Modules Network configurations (XML files located in the /IDRA/-Config/Net folder).
For both categories, the user can select one of the existing configurations, add a new one, edit an existing one, import, and remove.
The Bind button checks that the robot configuration and the net configuration selected are compatible. They are compatible only if every input filtered signal indicated in the net config file is provided by the robot, as indicated in the robot config file. Note that the opposite is not true: the robot can send to the Intentional Architecture data that is not used from the Intentional Module net (for example, data which is used only by the Phylogenetic Module).
The info Panels show the main informations about the selected configurations.
The Train button starts the TrainingSetup form. This button is disabled until the Bind button gives a positive response.
The Finish button starts the Intentional Agent operation's loop. This button is disabled until the training for the specific net and robot is complete.
Clicking the Add new or Edit button in the left column of the Idra window,

the RobotEditor window appears (Figure 4.2): From here the user can select

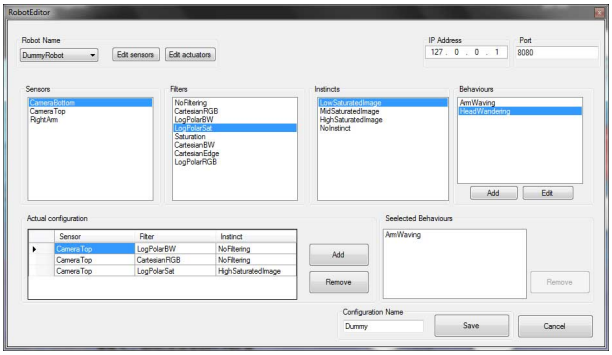


Figure 4.2: The RobotEditor window

the robot name, choosing from the available robots (CS files located in the /IDRA/AgentsLib/Robots folder).
Each robot has a list of physical sensors in the first column; for each sensor, the applicable filters are listed in the second column; for each couple, the applicable instincts are listed in the third column.
Once the user has selected the triplet of values, clicking the Add button will add it to the actual configuration, in the bottom.
The fourth column shows the available behaviors for the selected robot; the user can select a behavior and click the Add button to add it to the actual configuration. Clicking the Edit button will show up the BehaviorsEditor window (Figure 4.3): From here the user can add new behaviors or edit

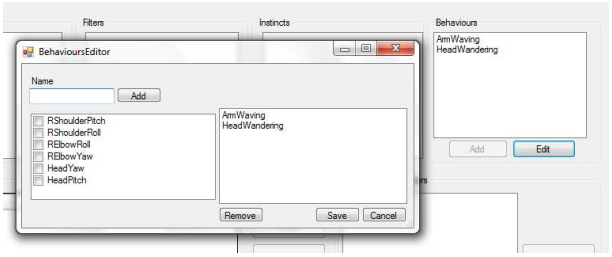


Figure 4.3: The BehaviorsEditor window

existing ones, by checking the related robot's actuators.
If the selected robot has not a sensors list already, or if the user wants to edit the sensors list of an existing robot, clicking the Edit sensors button will show up the EditSensors form (Figure 4.4): From here the user can see the list of available sensors for a specific robot, and he can add new sensors, specifying its type from the drop-down list.
If the selected robot has not an actuator list already, or if the user wants

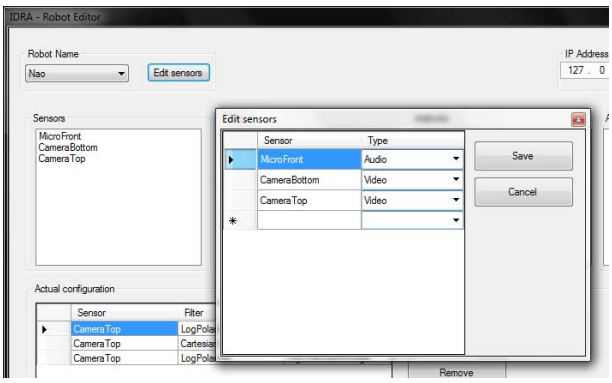


Figure 4.4: The EditSensors window

to edit the actuators list of an existing robot, clicking the Edit actuators button will show up the EditActuators form (Figure 4.5): From here the

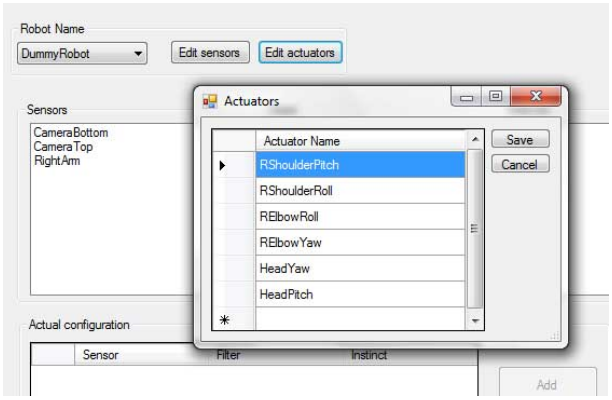


Figure 4.5: The EditActuators window

user can see the list of available actuators for a specific robot, and he can add new actuators.
Once the user has set the IP address, port, and configuration name, he can click Save to write the XML file and return to the IDRA main window.

Clicking the Add new or Edit button in the right column of the Idra window, the NetEditor appears (Figure 4.6): In the left column, the user can see a drop-down menu with all the possible filters applicable, the Intentional Architecture modules (Phylogenetic Module, Intentional Module, Layer), and all the possible Behaviors applicable.
The user can drag and drop them from left to right in the desired position. The minimum configuration provides the Filters Layer, a single empty IM Layer, and the Behaviors Layer.

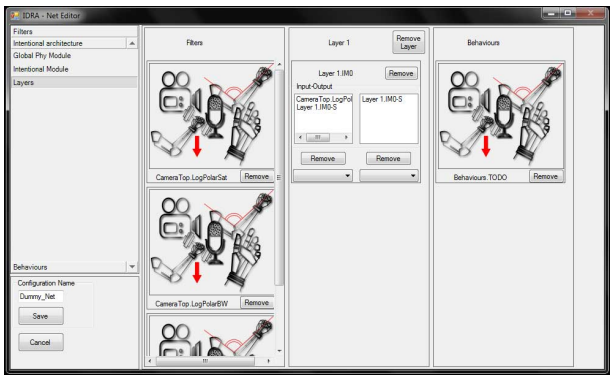


Figure 4.6: The NetEditor window

Each IM has two drop-down menu, one for the input and one for the output; using these menus the user can choose how to connect the various modules in the net (Figure 4.7). The drop-down menus list an entry for

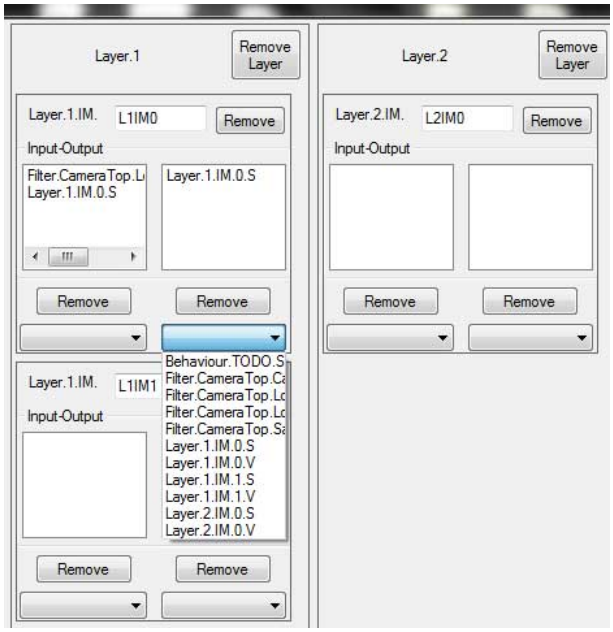


Figure 4.7: Intentional Modules in the NetEditor window

each sensor input (which is treated as a vector), an entry for each behavior (which receives input as a scalar), and two entries for each IM (since they have two inputs and two outputs: the category vector, and the relevant signal). These vectors and scalars can be distinguished due to the final letter in the list's name (V and S respectively).

Once the user has finished configuring the net, he can save it as a XML file choosing an appropriate name and clicking the Save button.

Clicking the Train button will show up the TrainingSetup form (Figure 4.8). This window allows the user to set a number of parameters necessary for the testing phase. In particular, it is used to tune several data relative to the ICA algorithm (see Appendix B), several data relative to the K-Means algorithm (see Appendix C), and several data relative to the Motor Primitives combination (see Chapters 3.2). It also allows to choose if the user wants to perform the IM network training, the robot's motor training, or both. Clicking the OK button will show up the TrainViewer window

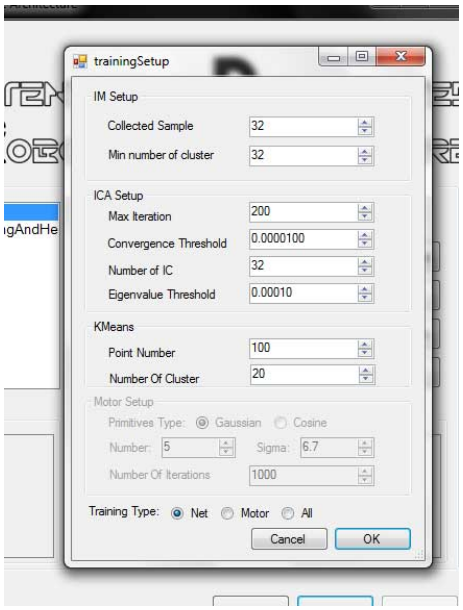


Figure 4.8: The trainingSetup window

(Figure 4.9). This window presents five buttons at the bottom:

- **Connect to Robot:** creates the connection with the robot; the other four buttons are disabled, before the connection is estabilished.
- **Play:** runs the logic for an indefinite number of turns.
- **One Step**: runs the logic one single turn.
- **Pause:** stops the logic keeping memory of what it's done, and allowing to start again with Play or One Step.

- **Stop:** stops the logic without keeping memory of what is done; clicking Play or One Step would make the computing start all over again.

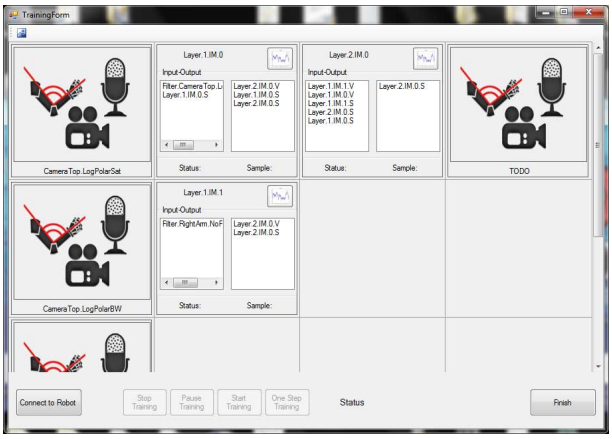


Figure 4.9: The TrainingViewer window

If you run the logic with the console instead of the GUI, the training start automatically in Play mode.
In the main panel, the TrainingViewer presents a grid, where each column represent an Intentional Modules layer (except the first column, where there are listed all the filtered inputs, and the last column, where there are listed all the behaviors).
For every filtered input, a graphical representation is given (e.g., an image is shown as an image; an audio signal is shown as an audio wave, etc.); for every Intentional Module, it is shown its ID, the list of all the origins of the input signal, the list of all the destination of the output signal, and the values of the Ontogenetic signal and Relevant signal.
Clicking on the Focus button, on the top right corner of an Intentional Module, would show up its related Focus window, showing detailed informations about the module. Here are displayed the number of collected samples and their dimension, the number of extracted independent components and their dimensions, and the number of categories (see Figure 4.10).
Once the training is complete (i.e. when all the IMs in the net will have collected a satisfactory number of categories), the results of the training will automatically be saved as BIN files.
Clicking the Disconnect from Robot and the Finish button will bring the user back to the IDRA window.

The Viewer window (Figure 4.11) presents five buttons at the bottom:

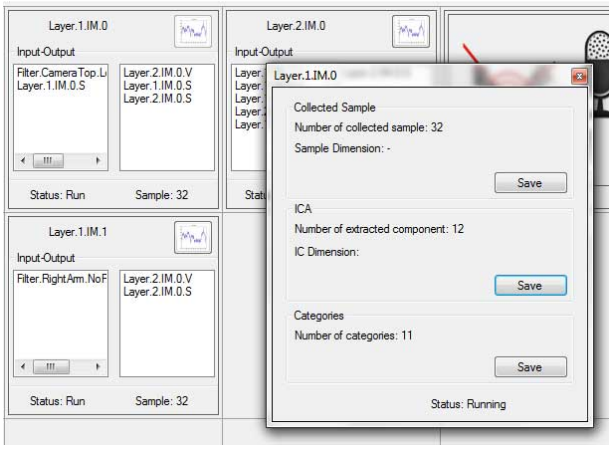


Figure 4.10: The TrainingFocus window in training mode.

- **Connect to Robot:** creates the connection with the robot; the other four buttons are disabled, before the connection is established.
- **Play:** runs the logic for an indefinite number of turns.
- **One Step**: runs the logic one single turn.
- **Pause:** stops the logic keeping memory of what it's done, and allowing to start again with Play or One Step.
- **Stop:** stops the logic without keeping memory of what is done; clicking Play or One Step would make the computing start all over again.

Figure 4.11: The Viewer window

If you run the logic with the console instead of the GUI, the programs start automatically in Play mode.

In the main panel, the Viewer presents a grid, where each column represent an Intentional Modules layer (except the first column, where are listed all the filtered inputs, and the last column, where are listed all the Behaviors). For every filtered input, a graphical representation is given (e.g., an image is shown as an image; an audio signal is shown as an audio wave, etc.); for every Intentional Module, are shown its ID, the list of all the origins of the input signal, the list of all the destination of the output signal, and the values of the Ontogenetic signal and Relevant signal.
Clicking on the Focus button, on the top right corner of an Intentional Module, would show up its related Focus window, showing detailed informations about the module. Here are displayed a line chart of Ontogenetic, Relevant and Phylogenetic signal, and a bar chart of the categories (see Figure 4.12). Once the simulation is complete (i.e. clicking the PAUSE button, NOT the

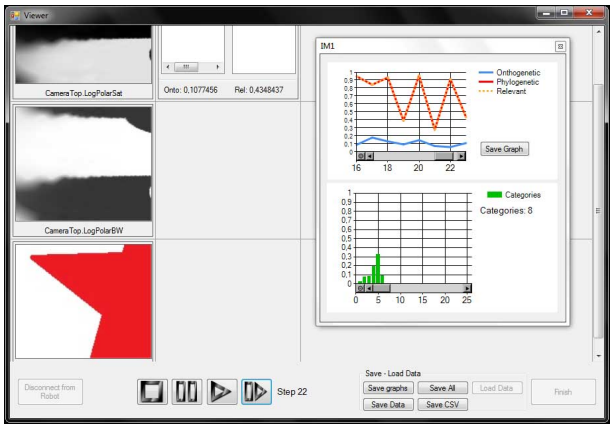


Figure 4.12: The formFocus window.

stop button), the user can click some of the Save buttons. The buttons are:

- **Save Data:** saves the categories in a BIN file, a lightweight and easy-to-read file format;
- **Save CSV:** saves the categories in a CSV - Comma Separated Values - file, which can be easily imported in Matlab or in a spreadsheet program for further analysis and computations.
- **Save graphs:** saves the graphs of the Ontogenetic and Phylogenetic history as a JPG file, for further analysis and visualization.
- **Save All:** does all of the above.

Clicking the Finish button will bring the user back to the IDRA window.

4.4 Use cases

According to what has been described in the previous Chapter, there are some possible use cases for the program. Figure 4.13 shows the activity diagram for a generic user that opens the program for editing several config files, that he could use later. We can see how he navigates through the RobotEditor and NetEditor windows, and how he can add, edit or remove items from the various lists of components, add or remove the Sensor-Filter-instincts triplets, or drag and drop Intentional Modules and then connect them.

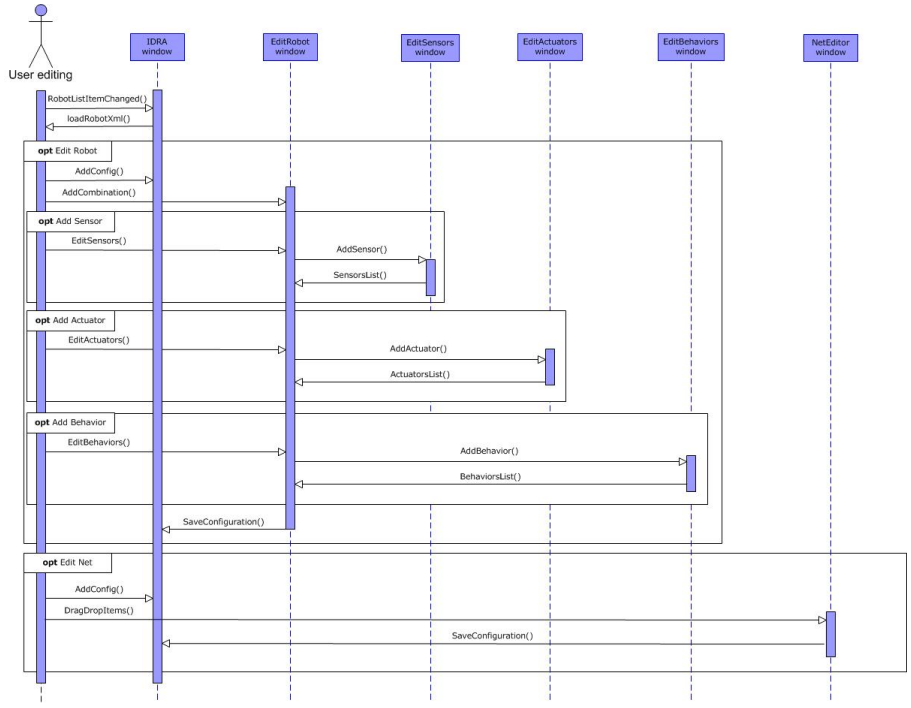


Figure 4.13: Activity Diagram 1

At the end of the editing session, the user can simply click the Finish button, and he can find his config files in the /Config folder.

Figure 4.14 shows the same user selecting two already existing configurations, one for the robot and one for the network, and then perform a training. The user can decide to train the network, train the motor part, or both. After the training phase, the related BIN files are automatically saved, and he can return to the main window. At this point, the user can start the main program functions, and see through the Viewer window hot

the Intentional Architecture performs all the various operations described so far.

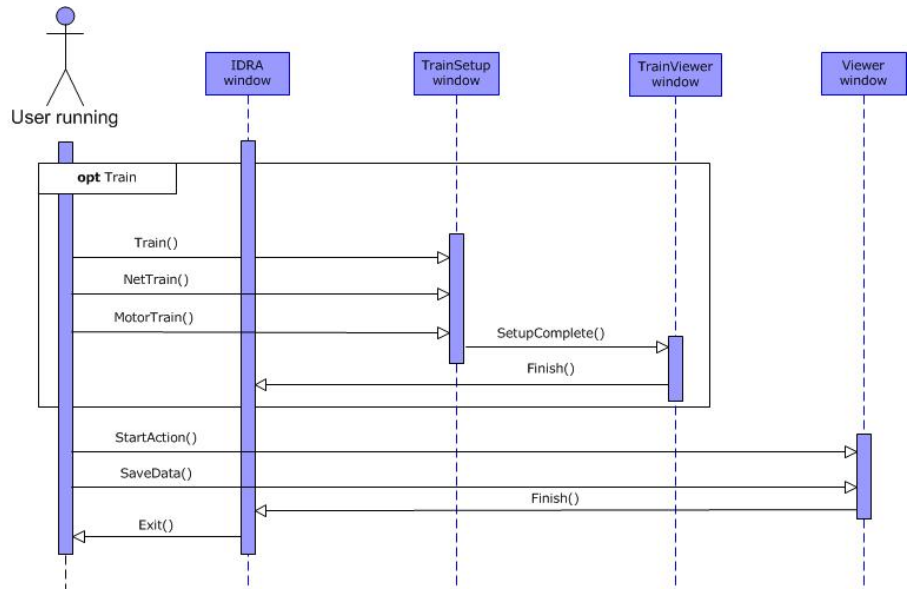


Figure 4.14: Activity Diagram 2

Once the user decides to stop the execution, he can click on the stop button, and, if he plans to restart the execution from where it stopped, he can save the necessary information in BIN and CSV files; otherwise, the execution of the program is finished.

Chapter 5

Experimental results

Marvin: "I am at a rough estimate thirty billion times more intelligent than you. Let me give you an example. Think of a number, any number."
Zem: "Er, five."
Marvin: "Wrong. You see?"

The Hitchhiker's Guide to the Galaxy

The Cognitive Architecture has been tested with various experiments, in order to prove its main features: goals generation and smart movements with Dynamic Behaviors. In the first experiment, we demonstrate how the agent is able to learn the shape of an object, starting from an instinct related to the color saturation of what it can see; in the second experiment, it moves according to the relevant signal coming from the architecture, learning to perform the movements that maximize this signal.

5.1 Experiment one: learning of new goals

One of the main features of the Intentional Architecture is its capability to develop new goals, starting from hard-coded ones. To test the efficacy of this feature, we have performed a simple experiment, with a basic network composed of a single Intentional Module, and with the input coming from a camera; the agent should be able to extract information about the color saturation of the image, according to a hard-wired instinct, and then learn the shape of the observed figure, thus developing a new interest for that particular shape. This simple experiment is similar to the one performed in [31], where the employed agent was a two degree of freedom robotic camera. Although the experiment is similar, it is a good starting point to test the goals generation feature of the Intentional Architecture in a simple

environment, as well as a way to validate it. Moreover, we used an agent far more complex compared to the previous one; in addition the previous experiment was performed with a single IM implementation, while this was the first time we performed the test with the full surrounding architecture.

5.1.1 Objectives

One of the main objectives of this architecture is to develop an agent who can completely change the way in which it relates with the environment through the experience that he receives from changes in the environment itself. The architecture presents a clear-cut distinction between the goals already present in the Phylogenetic Module and the ones that are generated in the Ontogenetic Module. The former are hard-coded, while the latter could be completely different according to the interactions of the agent with the world. This is strictly related to what happens in nature, where the former correspond to the animal's instincts, and the latter to its personality and how it evolves through the course of its life.
To prove this ability, a simple experiment has been set up, with a basic architecture endowed with a single innate ability, i.e. the "attraction" to colored objects; the test consists in showing how a new interest, i.e. the "attraction" to specific shapes, could show up without the need to hard-code it.

5.1.2 Settings

The experiment has been done using a NAO robot, a humanoid robot produced by Aldebaran Robotics. It is a complex robot, presenting 21 degrees of freedom and numerous sensors including two cameras, a sonar, bumpers, microphones, etc. During this experiment we have used only one sensor, the frontal camera, namely CameraTop, and two actuators controlling head movement, namely HeadPitch and HeadYaw (Figure 5.1).

The architecture runs on a Windows PC with the following technical specifications:

- Processor: Intel Core i7 920;
- Video Card: Nvidia GeForce GTS 250;
- Memory: 4 GB DDR3 Ram;
- Hard-drive: 1 TB 7200 rpm;

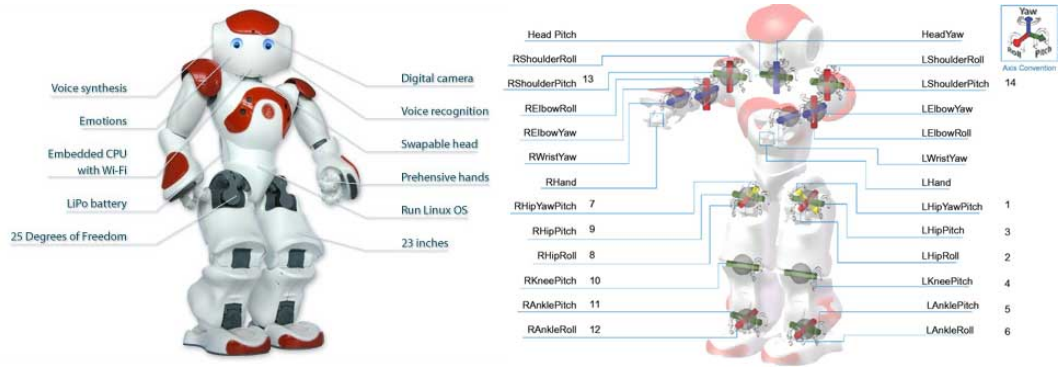


Figure 5.1: The Aldebaran Robotics NAO robot.

The test environment is limited to two board presenting different geometrical figures with different colors (Figure 5.2). The first board presents a series of black, colorless shapes, among which there is the shape of a black star. The second board presents some shapes of stars, painted in highly saturated colors. These boards are put on a wall in front of the NAO robot, at an adequate distance to allow the camera to see the entire board while moving.

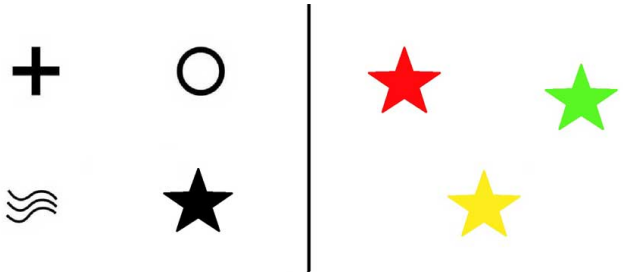


Figure 5.2: The two boards used in test one.

During this test we used a 0.6 threshold for the creation of new categories, a 0.6 threshold for determining the correlation of a signal to a category, a 128 limit for the number of clusters and number of categories, a 0.8 threshold and a 0.1 learning rate for Hebbian learning. These parameters allowed us to obtain a good rate in the memorization of categories and in learning from them.

5.1.3 Network of Intentional Modules

The network tested in this experiment is the simplest one. It is composed by only one layer including a single Intentional Module, and it receives as input just the video signal coming from the frontal camera.
This input is filtered in three different ways:

- logPolarBW filter: retrieves an RGB image from camera and returns the same image in log-polar coordinates, in a single channel color space;
- logPolarSat filter: retrieves an RGB image from camera and converts it in HSV (Hue Saturation Value) space, then returns the saturation channel extracted from the image in log-polar coordinates.
- cartesianRGB filter: retrieves an RGB image from camera and returns the same image in Cartesian coordinates in a three-channel color space. This input it is not actually used by the architecture: it is sent to the Viewer window (see Chapter 4.3), and it is useful in testing environments, for letting humans to easily understand what the robot is looking at, in every specific moment.

The Intentional Module receives as input the data from the logPolarBW filter in array form, while the data from the logPolarSat filter is received from the Phylogenetic Module. The phylogenetic signal here represents the percentage of high-saturated pixels in the image. The output of the Intentional Architecture is the output of the single Intentional Module composing the net (Figure 5.3).

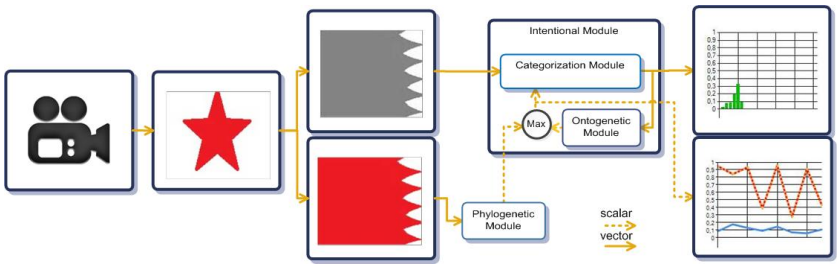


Figure 5.3: The architecture used in test one.

Since the experiment is related only to the development of new goals and the Intentional Architecture, we did not use the motor implementation to move the robot. Instead, in order to get better data from the test, movements are randomly generated, using a uniform distribution for the angle

and a bell-shaped Gaussian function for the amplitude of the movement. The probability function of the amplitude is:

$$p(r, \lambda) = \frac{e^{-\lambda r^2}}{\int_{-r_{max}}^{r_{max}} e^{-\lambda \rho^2} d\rho} \tag{5.1}$$

where r is a random variable and λ is the relevant signal output from the Intentional Module. We translated these mathematical functions in code as:

```
28          randomValue = new Random();
29
30          double cnst1 = 0.05;
31          double cnst2 = 15;
32          double cnst3 = 0.5;
33          delta = cnst3 / (iaSignal * cnst2 + 2);
34
35          double amp = Math.Sqrt(-2.0 * Math.Log(randomValue.NextDouble()*(1 -
                cnst1) + cnst1)) * Math.Sin(2.0*Math.PI*randomValue.NextDouble());
36          double angle = randomValue.NextDouble() * 2 * Math.PI;
37          float randomHeadYaw = (float)(delta * Math.Cos(angle) * amp);
38          float randomHeadPitch = (float)(delta * Math.Sin(angle) * amp);
39
40          float[] headTurning = new float[2];
41          headTurning[0] = oldHeadYaw + randomHeadYaw;
42          headTurning[1] = oldHeadPitch + randomHeadPitch;
```

As we can see, the variance of the Gaussian function depends on the relevant signal computed by the Intentional Architecture: an high value of the relevant signal produce a small variance, thus making a narrow bell-shape; as opposite, a low value of relevant signal produce a big variance making a large bell-shape. According to this, when the NAO robot is interested in what it sees, the movement of the head is small, while non attractive images lead to wide head movements.

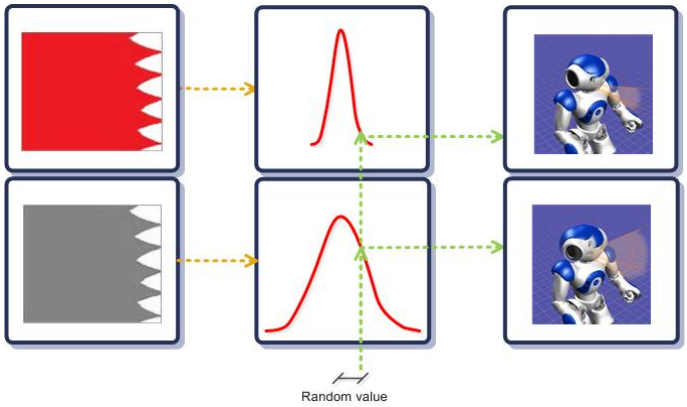


Figure 5.4: From Gaussian values to head rotation.

Training of the net

In order to project input in the basis space, independent components must be extracted through ICA algorithm. Since the net is composed by one single Intentional Module receiving a video input, ICA is executed on images coming from the camera. Parameters used for extract independent components are:

- number of sample: 2000;
- max number of iteration: 200;
- convergence threshold: 10^{-5};
- max number of independent components: 32;
- eigenvalue threshold: 10^{-4}.

Since the Motor System is not used in this test, the training is limited to the extraction of independent components.

5.1.4 Test execution and results

The experiment produced good results in the goal generation task.
In the first part of the experiment, the board is composed only by black colored geometrical figures (including a star-shaped figure). The interest of the NAO robot is more or less equal for every part of the board (Figure 5.5a). Figure 5.5b shows a chart of the points the agent is looking at. We can see how it aimlessly points at every part of its visual field, since he cannot find anything interesting. Figure 5.5c shows a line graph of the phylogenetic signal (red) and ontogenetic signal (blue); of course in this first phase they are always set to zero.

After some steps, the board is changed with a new one, showing only star-shaped figures with high-saturated colors. Now the interest of the NAO robot is focusing on the three star-shaped figures areas (Figure 5.6a).

Figure 5.6b demonstrates how the gazes of the agent are now focused around the objects of interests, that is the colored shapes (it is just bad luck if it scarcely looked at the red star, since it is always moving in a random direction; more steps in this phase would have shown how it was going to focus on the three stars equally). Figure 5.6c shows how the phylogenetic signal (red line) rapidly increases when it finds something interesting. We

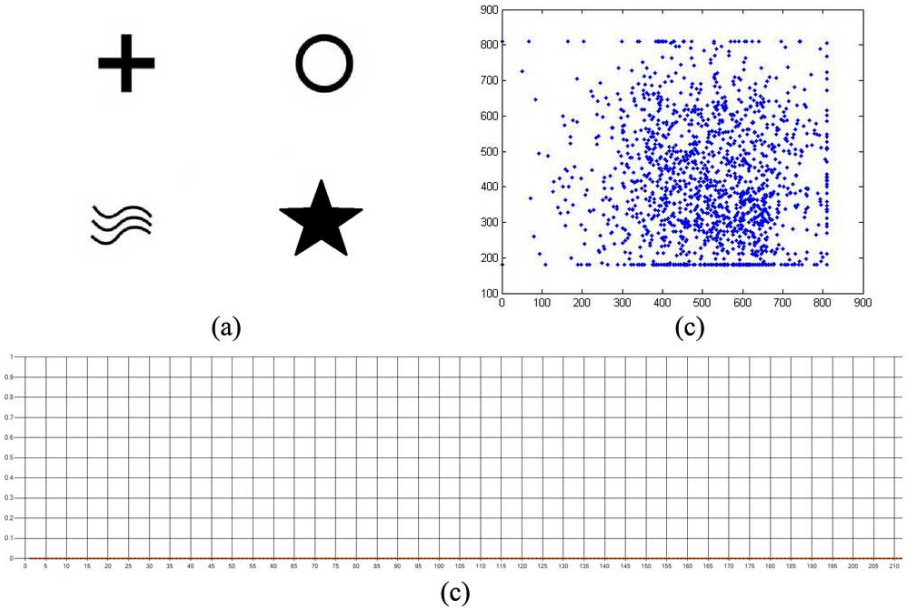


Figure 5.5: The first test, part one.

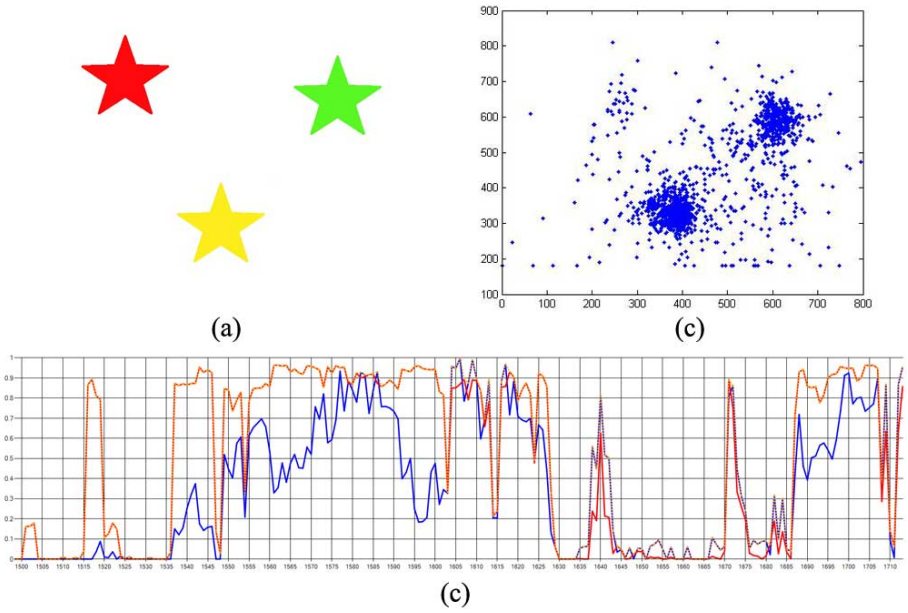


Figure 5.6: The first test, part two.

can also see how the red line is soon followed by an increase in the value of the blue line; this indicates how the Ontogenetic Module is starting to analyze the stored categories, and therefore it is starting to develop interest in the shapes of the objects that it can see.

Phylogenetic signal is high when the camera is focusing on colored figures, making the Intentional Module developing new categories. Through the Hebbian learning function, the Ontogenetic Module uses these developed categories in order to develop a new goals. The ontogenetic signal increases as the learning process is running.
After some categories have been developed, the board is switched again with the first one. Unlike before, the interest of the NAO robot is now focused on the star-shaped figure, which is black colored, meaning that the learning process of the Ontogentic Module has developed a new interest in the shape of the figure, which goes in addition with the previous interest for its color (Figure 5.7).

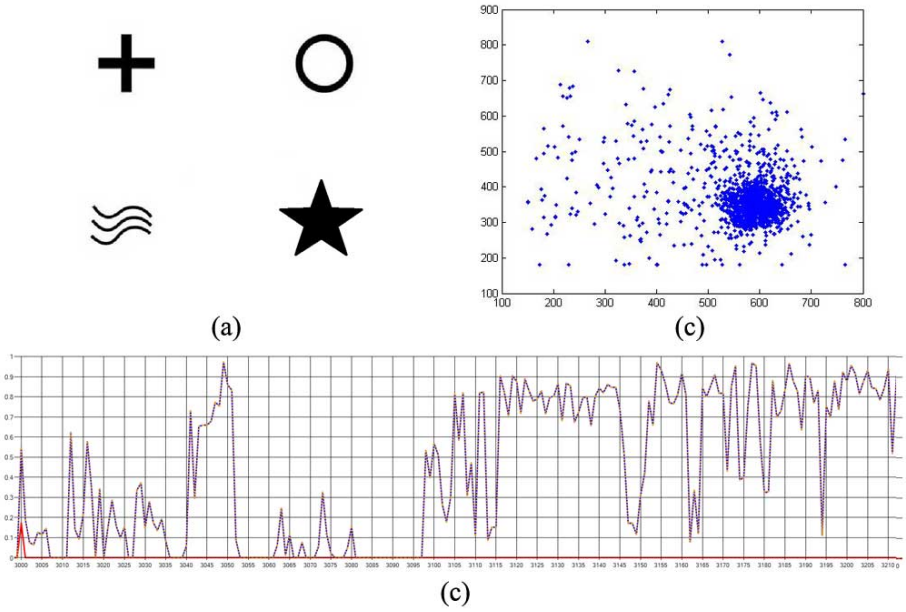


Figure 5.7: The first test, part three.

Figure 5.7b Demonstrates how the gazes of the agent are now focused around the objects of interests, that is the star shape, which was basically ignored in phase 1. Figure 5.7c shows how the phylogenetic signal (red line)

turns back to zero, since it cannot see anything interesting according to innate criteria. The blue line, instead, is showing high values, just like in phase 2. This is because the Intentional Module learned to be interested in the shapes of the objects that the robot can see.

This result confirms the capability of the intentional architecture of developing new goals that can be different from hard-coded ones.

5.2 Experiment two: learning of new movements

In the previous experiment we tested the ability of the architecture to develop new goals; in this second experiment, we test the ability of the architecture to learn new movements, according to its predefined behaviors.

5.2.1 Objectives

One of the main limitations of the first experiment is that it relegates the agent to a passive role, in its ability to learn from experience. The Gaussian movement of the head is a basilar solution in order to have a visual representation of the interest of the agent in what it can see: we could have obtained the same results holding the head still and manually showing some figure in front of the camera.
Once the agent has learned from the experience, we need it to be able to take some course of actions in order to interact with the environment, and so change its own perceptions towards a more satisfying condition.
The objective of the experiment is to test the Behavior and OutputSynthesis parts of the architecture, in particular its movement generation features. Starting from a simple hard-coded instinct, i.e. the "attraction" to colored object, and the ability to move only one of its arms, the robot learns which movements would allow him to see a colored object and then how to increase its reward signal.

5.2.2 Settings

The experiment is conduced using the same NAO robot we used in the previous experiment. During this experiment we have used only one sensor, the frontal camera, namely CameraTop, and four actuators controlling the movement of the right arm: RShoulderPitch, RShoulderRoll, RElbowRoll, RElbowYaw (Figure 5.8).

Figure 5.8: The NAO robot and its beloved heart.

The test has been performed on the same previous PC.
The test environment did not require anything more than the robot itself, a colored object, and a white background (to avoid the presence of other colored objects).

During this test we used a 0.5 threshold for the creation of new categories, a 0.28 threshold for the K-means algorithm, a 0.3 threshold for the minimum distance between centroids of clusters, a 1000 limit for number of points per cluster, a 10 limit for the number of clusters and number of categories, a 0.8 threshold and a 0.1 learning rate for Hebbian learning, a 0.8 threshold for the choice of the best possible movement in the Behaviors. For the motor primitives part, we have used 5 Gaussians as primitives, each one with a 6.7 standard deviation and a mean value calculated in order to equally distribute the functions on a scale from 0 to 100.
With respect to the previous experiment, we drastically reduced the number of categories and clusters; this because in the first experiment the head pointing was largely random, and therefore what the agent could see was continuously changing. In the second experiment, instead, the head was fixed, therefore the head kept looking at the same point, unless the arm itself did not pass in front of the camera, thus with scarce chances to create a decent number of categories. Instead, with regard to the arm, it has a very

limited input dimension (4, as much as the number of considered joints), especially with respect to the video input (we used a 160x120 image, thus the video input had a dimension of 19200 pixels). Limiting the number of categories allowed us to have a decent representation of the scarcely changing environment in which we put the robot.
What we said does not mean, of course, that the system could not work with a higher categories limit; nevertheless, the creation of a high number of meaningful categories and clusters would require considerably longer training times, which is beyond the scope of this thesis. As most of the offline training algorithms used nowadays, ICA and K-Means require a solid input signals base, in order to obtain good analysis data; and these basis require long times for the acquisition, and even longer times for the computation.
Anyway, even if in a limited scale, our test showed the desired results, as we will show in Section 5.2.4.

5.2.3 Network of Intentional Modules

The network in the second experiment is slightly more complex than the network we used in test one.
First of all, we have two input signals, filtered in four different ways:

- logPolarBW filter: retrieves an RGB image from camera and returns the same image in log-polar coordinates, in a single channel color space;
- logPolarSat filter: retrieves an RGB image from camera and converts it in HSV (Hue Saturation Value) space, then returns the saturation channel extracted from the image in log-polar coordinates.
- cartesianRGB filter: retrieves an RGB image from camera and returns the same image in Cartesian coordinates in a three-channel color space This input it is not actually used from the architecture: it is sent to the Viewer window (see Chapter 4.3), and it is useful in testing environments, for letting humans to easily understand what the robot is looking at, in every specific moment.
- rightArmPosition filter: retrieves the proprioceptive informations about the four joints of the right arms: that is, it returns a vector containing the joints angles values expressed in radians.

Then, we have an Intentional Architecture composed by two layers; the first layer contains two Intentional Modules. The first Intentional Module is similar to the one we have seen in the first experiment: it receives logpolar

gray images and records the shapes of the objects. The second intentional module, instead, receives the proprioceptive data about the position of the right arm.
Both these modules send their output vector and relevant signals to a third module, situated in layer 2. This module's output therefore represents the state of the known world: it combines the information about what it can see (how the world looks like), and where its arm is (what can it do to modify the world).
The Phylogenetic Module, like before, receives the logPolarSat filtered input, and then broadcast its signal to all the net. Therefore, if the robot is in a state where he can see colored objects, the third Intentional Module will have a high outgoing relevant signal (Figure 5.9).

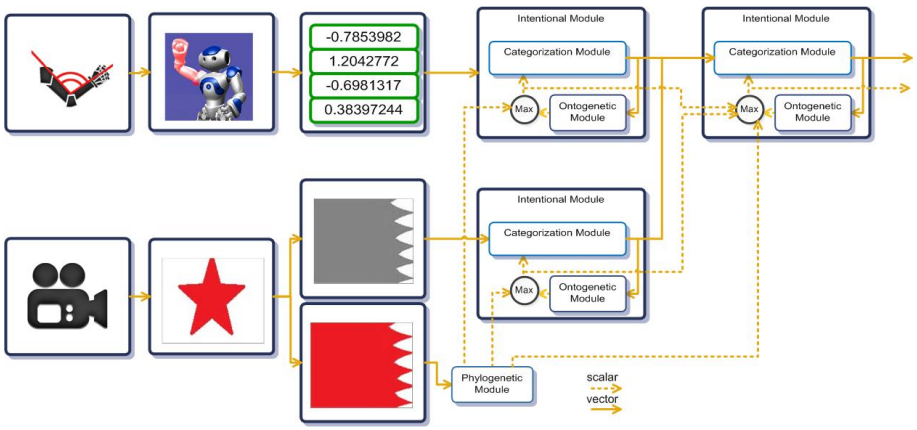


Figure 5.9: The architecture used in test two.

Training of the net

In order to project input in the basis space, independent components must be extracted through ICA algorithm. In this second test, the net is composed by two layers of Intentional Modules receiving different type of information: Intentional Modules of first layer receives video and proprioceptive information, while the Intentional Module of the second layer receive as input the output of first layer. According to this, the first Intentional Module computes ICA on video images, the second one on joints values of right arm (RShoulderPitch, RShoulderRoll, RElbowRoll, RElbowYaw) and the third Intentional Module computes ICA on the combined output of the others. Parameters used for extract independent components are:

- number of sample: 2000;
- max number of iteration: 200;
- convergence threshold: 10^{-5};
- max number of independent components: 32;
- eigenvalue threshold: 10^{-4}.

Once the net is trained, clusters for the Motor System should be computed. While the net is running, samples of the output are collected, then K-Means algorithm is executed to create clusters (see Appendix C). Parameters used for creation of the clusters are:

- number of sample: 2000;
- number of clusters: 100;.

5.2.4 Test execution and results

The experiment starts with the robot in a random position. When the program begins its execution, the NAO robot is free to move its right arm according to activation profiles received from the Motor System. At the beginning, the State-Action table is empty, and movements are chosen and executed randomly. After a number of steps, the table starts filling, as explained in chapter 3.2, and movements begin to be coherent with the maximization of the relevant signal received from the architecture. When the table contains a good number of entries, movements start to be frequently repeated, and we can observe that the same positions are reached, cyclically: several positions of the arms (rows in the State-Action table) know the reference to a movement (column in the State-Action table) that bring the hand to a position with a high relevant signal; several positions with a high relevant signal know many movements, but none of them brings the hand to a position with a high relevant signal. Therefore we observe the robot starting from a random position, going towards a good position (one with an high relevant signal), and then going towards a random position, and so on. The results of the experiment is shown in Figures 5.10 and 5.11:

Figure 5.10 shows the positions reached by right hand during a thousand iterations, in a three-dimensional space, from three points of view: front, side and top. The frame of reference is centered in the body of the robot, and we can notice the spherical displacements of the positions reachable from the

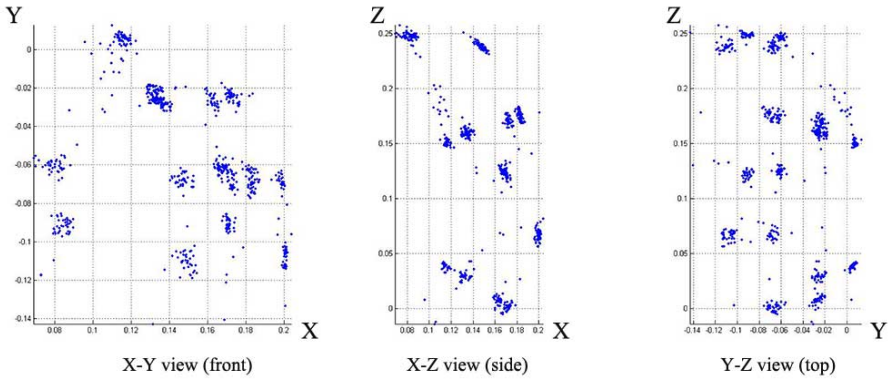


Figure 5.10: The hand positions in test two.

end effector of the NAO robot. More important, we can see the creation of some clusters, which represents the various positions cyclically reached by the robot.
In Figure 5.11, we can see the positions clusters manifesting the higher relevant signals, colored in red. For each cluster, we report a 3D representation of the NAO robot, showing its position, and what it can see through its top camera. We can notice how in the three pictures the ball is roughly at the center of its visual field.

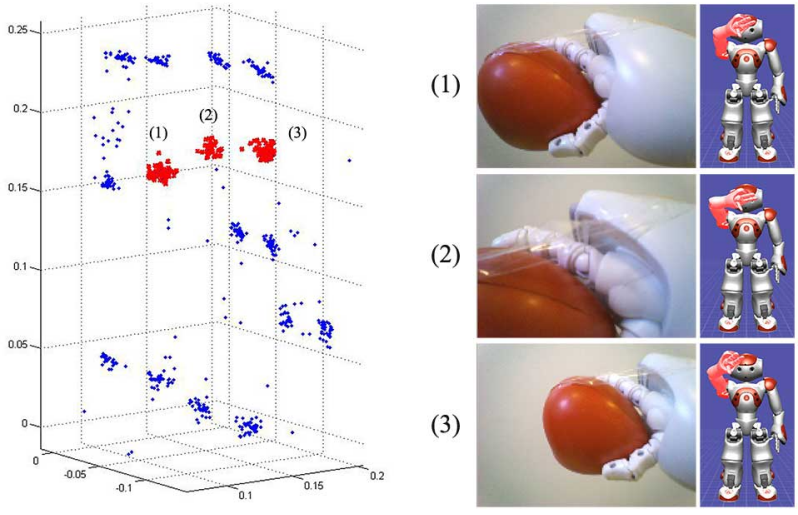


Figure 5.11: The most relevant hand positions in test two.

The implemented Motor System is extremely simple, with obvious limitations: the movements that maximize the relevant signal are performed only if the state is known, and if the State-Action table has the corresponding entry. According to this, the table requires a lot of entries to produce an effective movement selection system.
Furthermore, the motor training has been run for a relatively short period of time, compared to the long training times usually employed for training algorithms such as ICA or K-means clustering. As a consequence, the State-Action table presented a rather limited extension, in comparison with the high dimension of the input representing all the possible states of the environment. In particular, the limited number of columns and the static environment (nothing changes except for the position of the arm), causes the robot to execute always the same movements, whence the clustering: from any positions, a movement can bring the arm to a very limited set of positions in the three-dimensional space. This is worsened by the static environment: every movement always produces approximately the same movement, thus once the robot found a movement with a high relevant signal associated, he will never feel the need to explore new possibilities. Even more, although the brain has been proven to have an associative memory of sequences of patterns [4], the system here presented has no memory of previous actions: each action is "atomic" and totally unrelated to others. Therefore, the system is not able to "concatenate" two or more actions in order to reach a goal that is impossible to reach from a specific position in a single move.
Despite these limitations, our results are perfectly coherent with the objective of the experiment: the robot moves accordingly to the linear combination of primitives, and it is able to learn what movements has to be performed to go from a known state to a state with a high reward in the form of a high relevant signal. In addition to this, the experiment led to the creation of a sensorimotor map through the cognitive architecture. For all these reason, the implemented Motor System is an excellent starting point for the development of an effective and complex system which allows the agent to move in a smart way according to its goals.

Chapter 6

Future development and conclusions

"For now, we assume that self-evolving robots will learn to mimic human traits, including, eventually, humor. And so, I can't wait to hear the first joke that one robot tells to another robot."

Lance Morrow

6.1 Conclusions

The aim of this thesis is the creation of a bio-inspired software architecture based on the processes that take place in the human brain, based on an amygdala-thalamo-cortical model; this architecture must be able to learn new goals, as well as to learn new actions that could let it achieve such goals.
The architecture has been successfully designed and implemented as shown in Chapter 4. Its modularity will permit to add new functionalities at ease; the already present functionalities demonstrate how the architecture is able to receive input from the agent's sensors, elaborate them according to predefined goals, store the information about objects of relevance, and then develop new goals starting from its memory of the past experiences. Our first experiment, for example, has shown how the architecture is able to process what it can see according to a single criteria, and then develop an interest in some specific features of what it can see, interest that was not present at design time.
Moreover the architecture has been extended, and it is now able to employ different actuators of the robot's body, as well as to learn the best action

to perform in a given situation, in order to interact with the environment and then to obtain the highest reward possible from its actions. Our second experiment has shown how the agent is able to keep memory of the past situations where it had found itself at, and act accordingly to the achievement of its goals, whether innate or acquired. Once the position of its arm is registered, and once the agent learned the correct movement to perform in order to bring the object in front of its face, it is able to execute the right movement every time its arm is found in that particular position. Besides this, the intrinsic dynamicity of the architecture allows the agent to acknowledge the changings in the environment that are independent from its own actions, and then recalibrate the goodness of its actions for a particular situation.
The ability of developing new goals have been successfully tested, and it is working correctly as long as the agent receives sufficient stimuli for its basilar instincts; the simple addition of new instincts would let the agent to learn from many different aspects of the surrounding world.
The ability of developing new behaviors has been tested, and the results are satisfying. The motor part is correctly analyzing the situation of the world, and it is able to determine the best movement for any given situation. Yet, during the test the State-Action table has reached a limited dimension, due to the shortness of the training period itself, and this is a great limitation for the ability of the robot to cover the whole extent of its capabilities, since the robot was not able to test the whole extent of the environment states, and thus limiting its actions to a restricted pool.
Anyway, the architecture is well able to achieve the goals that we had set, and therefore this is an excellent starting point for future developments, that would improve the performances and precision of the implementation described here.

6.2 Future development

Despite the cognitive architecture we proposed in this thesis obtained good experimental results, it is still at a preliminary stage of development, since it is based on new ideas and concepts, and it was raised from scratch during the course of this project thesis. Therefore, a lot of improvements could be implemented to further develop this new architecture, and they are planned for the future.

6.2.1 Multiple-layer dynamic network

Ideally, the Intentional Architecture has the ability to develop new complex goals, and make mental associations in order to learn new concepts. To reach this objective, a lot of improvements have to be done on the intentional architecture, from input processing to the network of Intentional Modules itself.

At present, the Intentional Architecture receives inputs from a few types of sensors and filters (video and proprioceptive). The use of different types of input (e.g. audio, tactile, sonar, etc.) and their integration could led to a better representation of the environment, giving more information to the computation of new categories. These inputs are processed by a network composed by only two layers: the first one receives inputs from sensors and the second one, composed by one module, computes the data from the first layer and elaborates the output of the entire Intentional Architecture. although this network is enough for the generation of new goals, it is one of the simplest that could be done. A more complex network, composed by more than two layers, should be implemented to test the idea that an increasing number of Intentional Modules leads to better performances (Figure 6.1). Lots of research should be spent to study the correct way to connect Intentional Modules, making them communicate in a smart and efficient way.
Another important features to be made is the adaptability of the network of Intentional Modules. The network should be dynamic: new modules have to be created at run time if needed, as well as existing modules have to be deleted once they are obsolete. According to this, we need to study when to create new modules and when to destroy existing ones, basing our research on the dynamics of the brain.
An Intentional Architecture with all these feature could be a good candidates to reach our complex goal.

6.2.2 Improvements in data abstraction

Abstraction from raw input data must be done in order to get generality, as well as adaptability to unknown situations. To obtain this feature, we implemented the Independent Component Analysis algorithm (ICA, see appendix B). This is a statistical-based algorithm, and it needs a lot of samples to extract the independent components. According to this, a time-expensive training phase has to be done to allow the system to extract its features: thousands of samples must be collected, before the algorithm can compute its output. As a future development, the system need to be able to get ab-

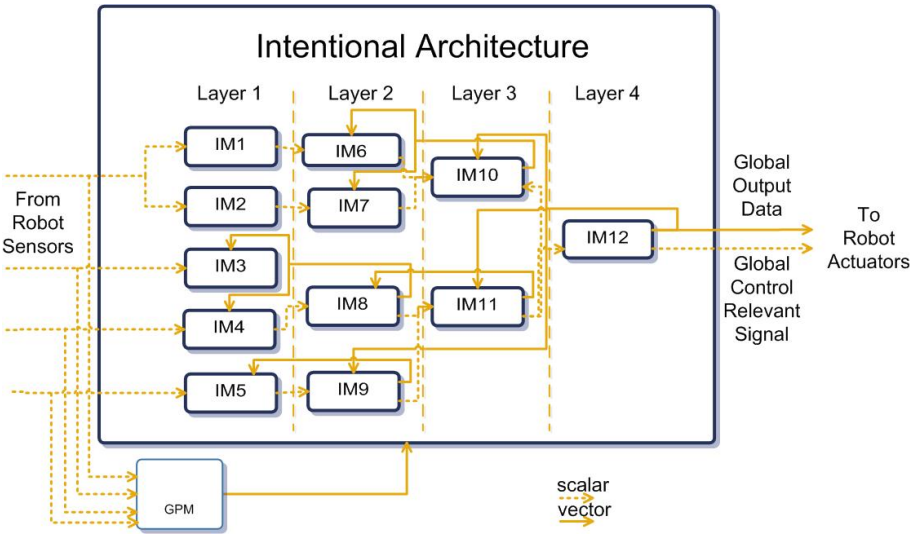


Figure 6.1: An ideal complex network of intentional modules

straction from raw input data while it is running, without having to wait for the computation of independent components. To obtain this result, a new way to abstract from inputs should be studied. New algorithms should be tested to compare results and verify which one leads to better performances. An ICA online algorithm could also be studied and implemented.

6.2.3 Hierarchical movements and behaviors learning

An improvement of the Intentional Architecture is certainly a fundamental requirement to obtain better results; the development of new goals and concepts, however, is totally useless without a motor system able to satisfy these new goals in an optimal way.

As we have shown, the motor system is able to generate movements by linear combination of motor primitives, as well as to use these movements to satisfy new and pre-defined goals. The concept of motor primitives could be improved, with more research on this topic and on the way they are used by our motor system. Movements could be created hierarchically, using the combination of motor primitives to generate a set of movements, and then combining this new set of "first order movement" to generate a more complex set of movements. This way, some tasks could be accomplished in a simple and automatic way, without complex computations.
The choice of the best movement to perform is as important as the movement

generation itself. At this time, this task is executed by the State-Action table of dynamic behaviors; but this is a very simple system, too much dependent from a satisfactory training phase. An improvement to the State-Action table can be the implementation of a neural network. By using a neural network, a behaviors should be able to predict the best movement also in an unknown state.
The agent is able to develop new movements for a specific behavior, but it is not able to create new behaviors, or to mix behaviors abilities using, for example, different combinations of actuators. A fixed number of behaviors is a significant limit to the motor system, which is unable to evolve the same way as the rest of the Intentional Architecture. In a future development, the list of behaviors needs to be dynamic: the agent should generate not only new goals, but also new complex behaviors in order to better satisfy these goals.

6.2.4 Distributed computation

By a technical point of view, the implementation of the architecture requires a low-end hardware to be executed with good performances. However, aforementioned improvements to the cognitive architecture requires certainly more computational power in order to be executed with the same performances.

One way to get more computational power is making the architecture distributed. Current implementation acts in a synchronous way, without taking advantage of threading. An intentional module that receives inputs from other modules need to wait until their data are ready. This causes a significant performance loss.
As self-explained by its name, the IDRA project is meant to be distributed, so the list of first future improvements includes the implementation of a distributed and asynchronous version of the architecture in a thread-like structure. In the far future, it would be possible to consider to implement a version of the architecture that may be spread on more computers, thus taking advantage of more memory and computing power. Such version should be totally asynchronous.

6.2.5 Multiple agents

The cognitive architecture has been tested only with NAO, a humanoid robot produced by Aldebaran Robotics. Nevertheless, one of the main features of the cognitive architecture is generality. The architecture should work the same way with all kinds of agents, whether real or virtual, humanoids or not.

According to this, the cognitive architecture needs to be tested with different types of agents. In future, different robots implementation will be put under test, to see the efficacy of the system with any robotic or virtual agent. Implementation of the cognitive architecture in a virtual agent could be of interest to test the application of the system in the field of videogames, aiming at the creation of a realistic and autonomous character.
It would also be interesting to experiment with the interaction between multiple agents. New goals and behaviors could emerge by the interaction among multiple robots, leading to group behaviors, with agents cooperating to satisfy a common goal.

Bibliography

[1] D.F. Bjorklund. *Children's Thinking: Cognitive Development and Individual Differences*. Wadsworth Publishing, 2004.

[2] R. C. Arkin. *Behavior-based Robotics*. 1999.

[3] M. Lungarella, G. Metta, R. Pfeiffer, and G. Sandini. Developmental robotics: a survey. *Connection Science*, 15:4:151–190, 2003.

[4] J. Hawkins and S. Blakeslee. *On Intelligence*. 2004.

[5] G. Dileep. *How the brain might work: A hierarchical and temporal model for learning and recognition*. PhD thesis, 2008.

[6] P. Viviani. Do units of motor action really exist? *Generation and Modulation of Action Patterns*, pages 201–216, 1986.

[7] F. A. Mussa-Ivaldi and E. Bizzi. Motor learning through the combination of primitives. *Philosophical Transactions of the Royal Society Lond. B Biological Sciences*, pages 355:1755–1769, 2000.

[8] F. A. Mussa-Ivaldi and S. A. Solla. Neural primitives for motion control. *IEEE Journal of Oceanic Engineering*, pages 29:640–650, 2004.

[9] C. B. Hart and S. F. Giszter. Modular premotor drives and unit bursts as primitives for frog motor behaviors. *The Journal of Neuroscience*, pages 24:5269–5282, 2004.

[10] P. S. G. Stein. Neuronal control of turtle hind limb motor rhythms. *Physiology A: Neuroethology, Sensory, Neural, and Behavioral Physiology*, pages 191:213–229, 2005.

[11] R.E. Kandel, J.H. Schwartz, and J.M. Thomas. *Principles of Neural Science Fourth Edition*. McGraw-Hill, 2000.

[12] O. Sporns. *Networks of the Brain*. The MIT Press, 2010.

[13] L. M. Ward. The thalamic dynamic core theory of conscious experience. *Consciousness and Cognition*, pages 20:464–486, 2011.

[14] D. S. Modha and R. singh. Network architecture of the long-distance pathways in the macaque brain. *Proceedings of the National Academy of Sciences*, pages 107(30):13485–13490, 2010.

[15] R. Adolphs and M. Spezio. Role of the amygdala in processing visual social stimuli. *Progress in Brain Research*, pages 156:363–378, 2006.

[16] S. Duncan and L. F. Barrett. The role of the amygdala in visual awareness. *Trends in Cognitive Sciences*, pages 11(5):190–192, 2007.

[17] J. S. Albus, D. T. Branch, C. Donald, and H. Perkel. A theory of cerebellar function, 1971.

[18] O. Hikosaka, K. Nakamura, K. Sakai, and H. Nakahara. Central mechanisms of motor skill learning. *Current Opinion in Neurobiology*, pages 12:217–22, 2002.

[19] R. Shadmehr and J. W. Krakauer. A computational neuroanatomy for motor control. *Experimental Brain Research*, pages 185:359–81, 2008.

[20] J. Doyon et al. Contributions of the basal ganglia and functionally related brain structures to motor learning. *Behavioral Brain Research*, Volume 199, Issue 1:61–75, 2009.

[21] Y. P. Ivanenko, R. E. Poppele, and F. Lacquaniti. Five basic muscle activation patterns account for muscle activity during human locomotion. *The Journal of Physiology*, Volume 556, Issue 1:267–282, 2004.

[22] A. E. Patla. Some characteristics of emg patterns during locomotion: implications for the locomotor control process. *Journal of Motor Behavior*, Volume 17:443–461, 1985.

[23] A. E. Patla, T. W. Calvert, and R. B. Stein. Model of a pattern generator for locomotion in mammals. *AJP - Regulatory Integrative and comparative Physiology*, Volume 248:R484–R494, 1985.

[24] C. B. Hart and S. F. Giszter. A neural basis for motor primitives in the spinal cord. *The Journal of Neuroscience*, 30(4):1322–1336, 2010.

[25] M. C. Tresch and A. Jarc. The case for and against muscle synergies. *Current Opinion in Neurobiology*, Volume 19, Issue 6:601–607, 2009.

[26] S. Schaal, J. Peters, J. Nakanishi, and A. J. Ijspeert. Learning movement primitives. *Springer Tracts in Advanced Robotics*, pages 561–572, 2005.

[27] A. J. Ijspeert, J. Nakanishi, and S. Schaal. Learning rhythmic movements by demonstration using nonlinear oscillators. *International Conference on Intelligent Robots and Systems*, 2002.

[28] R. S. Jackendoff and F. Lerdahl. The capacity for music: what is it, and what's special about it? *Cognition*, Volume 100, Issue 1:33–72, 2006.

[29] R. S. Jackendoff. Consciousness and the computational mind. *MIT Press*, pages 100:33–72, 1987.

[30] S. Harnad. The symbol grounding problem. *Physica, D: Nonlinear Phenomena*, Volume 42, Issue 2-3:335–346, 1990.

[31] R. Manzotti and V. Tagliasco. From "behaviour-based" robots to "motivations-based" robots. *Robotics and Autonomous Systems*, pages 51(2–3):175–190, 2005.

[32] M. G. Gazzaniga. *The cognitive neurosciences III.* 2004.

[33] E. Macaluso and J. Driver. Multisensory spatial interactions: a window onto functional integration in the human brain. *Trends in Neurosciences*, pages 28:263–271, 2005.

[34] D. Todman. Wilder penfield (1891-1976). *Journal of Neurology*, pages 255(7):1104–1105, 2008.

[35] B. J. Harrison, J. Pujol, M. Lopez-Sola, R. Hernandez-Ribas, and J. Deus et al. Consistency and functional specialization in the default mode brain network. *Proceedings of the National Academy of Sciences of the United States of America*, pages 105(28):9781–9786, 2008.

[36] K. Lorenz. *Evolution and Modification of Behavior.* The University of Chicago Press, 1965.

[37] E .A. Murray and S. P. Wise. Interactions between orbital prefrontal cortex and amygdala: advanced cognition, learned responses and instinctive behaviors. *Current opinion in neurobiology*, pages 20(2):212–20, 2010.

[38] M. Tamietto and B. de Gelder. Neural bases of the non-conscious perception of emotional signals. *Nature Reviews Neuroscience*, pages 11, 697–709, 2010.

[39] W. Chaney. *Dynamic Mind*. Houghton-Brace Publishing, 2007.

[40] W. Chaney. *Workbook for a Dynamic Mind*. Houghton-Brace Publishing, 2006.

[41] R. Manzotti. Towards artificial consciousness. *Newsletter on Philosophy and Computers*, pages 07(1):12–15, 2007.

[42] R. Manzotti. No time, no wholes: A temporal and causal-oriented approach to the ontology of wholes. *Axiomathes*, pages 19:193–214, 2009.

[43] J. Sharma, A. Angelucci, and M. Sur. Induction of visual orientation modules in auditory cortex. *Nature*, pages 404:841–849, 2000.

[44] A. Angelucci, M. Sur, and J. Sharma. Rewiring cortex: the role of patterned activity in development and plasticity of neocortical circuits. *Journal of Neurobiology*, pages 41:33–43, 1999.

[45] M. Sur, P.E. Garraghty, and A. W. Roe. Experimentally induced visual projections into auditory thalamus and cortex. *Science*, pages 242:1437–1441, 1988.

[46] S. J. Thorpe. Spike arrival times: A highly efficient coding scheme for neural networks. *Parallel processing in neural systems*, 1990.

[47] S. Sherman and R. Guillery. *Exploring the Thalamus*. Elsevier, 2000.

[48] R. Nieuwenhuys, J. Voogd, and C. van Huijzen. *The Human Central Nervous System: A Synopsis and Atlas*. Steinkpoff, 2007.

[49] M. A. Arbib. Schema theory. *The Encyclopedia of Artificial Intelligence*, pages 1427–1443, 1992.

[50] M. Jeannerod, M. A. Arbib, G. Rizzolati, and H. Sakata. Grasping objects - the cortical mechanism of visuomotor transformation. *Trends in neurosciences*, pages 18:314–320, 1995.

[51] S. Schaal, A. Ijspeert, and A. Billard. Computational approaches to motor learning by imitation. *Philosophical Transactions of the Royal Society Lond. B Biological Sciences*, pages 358:537–547, 2003.

[52] B. B. Averbeck, M. V. Chafee, D. A. Crowe, and A. P. Georgopoulos. Parallel processing of serial movements in prefrontal cortex. *Proceedings of the Nationall Academy of Sciences*, pages 99:13172–13177, 2002.

[53] J. R. Flanagan and D. J. Ostry. Trajectories of human multi-joint arm movements: Evidence of joint level planning. *Experimental Robotics I*, pages 594–613, 1990.

[54] E. V. Evarts. Relation of pyramidal tract activity to force exerted during voluntary movement. *Journal of Neurophysiology*, pages 31(1):14–27, 1968.

[55] A. G. M. Canavan, P. D. Nixon, and R. E. Passingham. Motor learning in monkeys (macaca fascicularis) with lesions in motor thalamus. *Experimental Brain Research*, pages 77:113–126, 1989.

[56] A. Hyvärinen and E. Oja. Independent component analysis: Algorithms and applications. *Neural Networks*, pages 13(4–5):411–430, 2000.

[57] A. Oursland, J. De Paula, and N. Mahmood. Case studies of independent component analysis. *Numerical Analysis of Linear Algebra*, 2004.

[58] T. M. Cover and J. A. Thomas. Elements of information theory. 1991.

[59] A. Papoulis. Probability, random variables, and stochastic processes (3rd ed.). 1991.

[60] A. Hyvärinen. New approximations of differential entropy for independent component analysis and projection pursuit. *Advances in Neural Information Processing Systems*, Volume 10:273–279, 1998.

[61] A. Hyvärinen. Fast and robust fixed-point algorithms for independent component analysis. *IEEE Transactions on Neural Networks*, pages 10(3):626–634, 1999.

Appendix A

IDRA code excerpts

In this chapter we present the code of IDRA implementation. About fourteen thousand lines of codes has been written, so we decided to describe only the main classes and functions of the implemented software.

A.1 Intentional Agent

IntentionalAgent class is the "engine" of the system, connecting each module: it manages the execution of the program, as well as the flow of data between the various parts of the system.
We present the main functions of the class: the `Init()` function, which performs the initialization of the modules, and the `loop()` function, that manages the execution flow of the program.

```
167         /// <summary>
168         /// Function initializing the variables and starting the connection with the
                robot.
169         /// </summary>
170         public void init()
171         {
172             Console.WriteLine("Initializing␣the␣application.");
173 
174             //Useful for saving in CSV format: set the decimal separator from "," to
                    "."
175             System.Globalization.CultureInfo customCulture =
176         (System.Globalization.CultureInfo)System.Threading.Thread.CurrentThread.
                CurrentCulture.Clone();
177             customCulture.NumberFormat.NumberDecimalSeparator = ".";
178             System.Threading.Thread.CurrentThread.CurrentCulture = customCulture;
179 
180       //Prepare table for net configuration
181             layersList = new Hashtable();
182             linkVecList = new Hashtable();
183             linkSigList = new Hashtable();
184             behavioursToActuators = new Dictionary<string, string[]>();
```

```
185
186     //Create Intentional Architecture
187           IA = new IntentionalArchitecture();
188           tIA = new TrainIntentionalArchitecture();
189
190     //Create body
191           body = new Body();
192
193     //Get filters and instincts list
194           filtersName = body.getFiltersNames();
195           instinctsName = IA.getInstinctList();
196       }
```

```
256       /// <summary>
257       /// Main running loop.
258       /// Each loop receives data from sensors and sends them to the intentional
              architecture,
259       /// and then receives instructions from the intentional architecture and
              sends them to actuators.
260       /// </summary>
261       /// <returns>true if success, false otherwise</returns>
262       public bool loop()
263       {
264           //Receives data from sensors.
265           iaInput = body.runSensors();
266           if (!isTrainingNetMode)
267           {
268               Console.WriteLine("Starting␣step...");
269               //Sends input data to intentional architecture.
270               IA.IAinput = iaInput;
271               //Calculate the output of the intentional architecture
272               IA.computeIAsignal();
273               //Get the output signal
274               iaOutput = IA.IAsignal;
275               //Get the output state
276               iaState = IA.IAstate;
277       //Send data to actuators
278               body.runActuators(iaState, iaOutput, false);
279           }
280           else
281           {
282               //Sends input data to intentional architecture.
283               tIA.IAinput = iaInput;
284               //Calculate the output of the intentional architecture
285               tIA.computeIAsignal();
286               //Get the output signal
287               iaOutput = tIA.IAsignal;
288               //Get the output state
289               iaState = tIA.IAstate;
290       //Send data to actuators
291               body.runActuators(iaState, iaOutput, tIA.IsTraining);
292           }
293
294           //Check if the motor training is runnig.
295           this.trainMotor = body.getTrainingStatus();
296
```

```
297             //Send data to actuators
298             Console.WriteLine("Step has ended...");
299             return true;
300         }
```

A.2 Intentional Architecture

The IntentionalArchitecture class manages the network of Intentional Modules, allowing their communication and computing the output of the net. We present the code that performs these operations, respectively with the `createIMNet(...)` function and the `computeIAsignal()` function.

```
269         /// Create the net of Intentional Modules
270         /// </summary>
271         /// <param name="layersList">Hashtable containing the list of layers and IM
                for each layer</param>
272         /// <param name="linkVecList">Hashtable containing the list of all vector
                links</param>
273         /// <param name="linkSigList">Hashtable containing the list of all signal
                links</param>
274         /// <param name="netFileName">Name of the net</param>
275         public void createIMNet(Hashtable layersList, Hashtable linkVecList,
276           Hashtable linkSigList, string netFileName)
277         {
278             ims = new Hashtable();
279             ArrayList vectorSenderList = new ArrayList();
280             ArrayList controlSenderList = new ArrayList();
281             foreach (string s in layersList.Keys)
282             {
283                 if (s.Contains("Layer"))
284                 {
285                     foreach (string name in ((ArrayList)layersList[s]))
286                     {
287                         vectorSenderList.Clear();
288                         foreach (string link in linkVecList.Keys)
289                             if (((string[])linkVecList[link])[1].Contains(name))
290                             {
291                                 vectorSenderList.Add(((string[])linkVecList[link])[0]);
292                             }
293                         controlSenderList.Clear();
294                         foreach (string link in linkSigList.Keys)
295                             if (((string[])linkSigList[link])[1].Contains(name))
296                             {
297                                 controlSenderList.Add(((string[])linkSigList[link])[0])
                                      ;
298                             }
299                         ims.Add(name, new IntentionalModule(name, vectorSenderList,
300                 controlSenderList, loadICS(netFileName, name)));
301                     }
302                 }
303             }
304             //Set the name of the last IM in the net to get the correct output
305             int max = -1;
```

```
306           foreach (string imName in ims.Keys)
307           {
308               int layerNumber = Convert.ToInt16(imName.Split('.')[1]);
309               if (layerNumber > max)
310               {
311                   lastImName = imName;
312                   max = layerNumber;
313               }
314           }
315
316
317           //Set the vector links between IM using the delegate method
318           foreach (string link in linkVecList.Keys)
319           {
320               string sorg = ((string[])linkVecList[link])[0];
321               string dest = ((string[])linkVecList[link])[1];
322               if (sorg.Contains("IM") && dest.Contains("IM"))
323                   ((IntentionalModule)ims[sorg]).InputDelegate +=
324         new vectorInputDelegate(((IntentionalModule)ims[dest]).getVectorInput);
325           }
326
327           //Set the control links between IM using the delegate method
328           foreach (string link in linkSigList.Keys)
329           {
330               string sorg = ((string[])linkSigList[link])[0];
331               string dest = ((string[])linkSigList[link])[1];
332               if (sorg.Contains("IM") && dest.Contains("IM"))
333                   ((IntentionalModule)ims[sorg]).ControlDelegate +=
334         new ControlSignalDelegate(((IntentionalModule)ims[dest]).getControlInput);
335           }
336       }
```

```
102       /// <summary>
103       /// Compute the output of the Intentional Architecture
104       /// </summary>
105       public void computeIAsignal()
106       {
107           //Computes phylogenetic signal.
108           phyMod.computePhySignal(iaData);
109
110           //Sends phylogenetic signal to all the intentional module.
111           foreach (string imName in ims.Keys)
112               ((IntentionalModule)ims[imName]).PhySignal = phyMod.PhySignal;
113
114           //Pass the new data from sensors to IM
115           foreach (string imName in ims.Keys)
116             ((IntentionalModule)ims[imName]).getVectorInput(iaData, "Filter");
117
118           //Send the correct output
119           iaSignal = ((IntentionalModule)ims[lastImName]).RelevantSignal;
120           iaState = ((IntentionalModule)ims[lastImName]).Output.OfType<float>().
                ToArray();
121
122           //Prepare the data to be displayed
123           moduleValues.Clear();
124           foreach (string imName in ims.Keys)
```

```
125         {
126             IntentionalModule im = (IntentionalModule)ims[imName];
127             float[] temp = new float[130];
128             for (int i = 0; i < temp.Length; i++)
129                 temp[i] = -1;
130             temp[0] = im.getOnthogeneticSignal();
131             temp[1] = im.RelevantSignal;
132             im.getCategoriesCorrelation().CopyTo(temp, 2);
133             moduleValues.Add(imName, temp);
134             }
135     }
```

A.2.1 Categorization

The process of categorization is one of the most important part of the architecture: creation of new goals depends on the neural activation of categories. Categorization is performed in CategorizationModule class, using the `categorize()` function.

```
159     /// <summary>
160     /// Creates new categories
161     /// </summary>
162     public void categorize()
163     {
164         List<Cluster> clusters = new List<Cluster>();
165 
166   //Categorize only relevant and different information
167         if (catMemory.Count < maxNumOfCluster &&
168             relevantSignal > relevantSignalThreshold)
169         {
170     //Get the input data in single format
171             float[] temp = new float[input.Count];
172             for (int i = 0; i < input.Count; i++)
173             {
174                 temp[i] = Convert.ToSingle(input[i]);
175             }
176 
177     //Create a point and perform clustering
178             Point p = new Point(temp);
179             clusters = kmeans.DoKMeans(p);
180 
181             catMemory.Clear();
182             foreach (Cluster c in clusters)
183             {
184                 ArrayList tempArr = new ArrayList(c.Centroid.PointData);
185                 catMemory.Add(tempArr);
186             }
187         }
188     }
```

A.2.2 Hebbian Learning

Goals generation is managed by the OntogeneticModule class. Hebbian learning is at the base of the goals development process, and it is implemented with the `hebbianLearning()` function.

```
99      /// <summary>
100     /// Hebbian learning function. This function ensures that whenever a certain
            level
101     /// of correlation (defined by thresholdFixing) is established,
102     /// the system stops looking for a positive correlation
103     /// and that signal is definitely assumed to be relevant. (=1.0)
104     /// </summary>
105     public void hebbianLearning()
106     {
107         for (int i = 0; i < weights.Count; i++)
108         {
109             if ((float)weights[i] > thresholdFixing)
110             {
111                 weights[i] = 1.0f;
112             }
113             else
114             {
115                 float j = (float)(input[i]);
116                 weights[i] = (float)weights[i] + learningRate *
117           (hebbianSignal * j - ((float)weights[i]) * j * j);
118             }
119         }
120     }
```

A.3 Motor System

Movements are generated through combination of motor primitives and the best movement to perform is selected according to a predefined policy, aiming at maximizing the relevant signal. These operations are executed by two classes: OutputSynthesis class and Behaviour class.
In OutputSynthesis class there is the `getMovement(...)` function, which compute the activation profiles that are sent to the agent.

```
73      /// <summary>
74      /// Get the activation profiles of joint values to generate the robot's
            movement.
75      /// </summary>
76      /// <param name="state">Current state of the environment</param>
77      /// <param name="relSig">Relevant signal associated to the current state.</
            param>
78  /// <param name="isNetTrainingMode">true if system is in net training mode, false
        otherwise</param>
79      /// <returns>List of joint values for the actuators</returns>
80      public Dictionary<string, float[]> getMovement(float[] state, float relSig,
            bool isNetTrainingMode)
```

```
81      {
82          float[,] weights;
83          bool[] BehaviorsTrainingStatus = new bool[behaviourList.Count];
84          Dictionary<string, float[]> movements = new Dictionary<string, float[]>();
85
86      //Get the cluster corresponding to the current state
87          int pointCluster = km.getPointClusterId(new Point(state));
88
89          foreach (Behaviour behavior in behaviourList)
90          {
91       //IF the net is not training, get a value from behavior
92       //ELSE generate a random value
93              if (!isNetTrainingMode)
94                  weights = behavior.getBehaviourMovements(pointCluster, relSig,
                        counter);
95              else
96              {
97                  Console.WriteLine("Training␣mode,␣generating␣random␣weights");
98                  weights = generateRandomWeights(behaviorsToActuators[behavior.
                        Name].Length,
99                       primitives.GetLength(0));
100             }
101
102             //IF there is a good movement to perform, combines primitives to get
                     the joint values
103             //ELSE add a new random movement and try it
104             if (weights != null)
105             {
106                 Console.WriteLine("Movement␣OK.");
107             }
108             else
109             {
110                 Console.WriteLine("No␣good␣movements␣in␣the␣list,␣adding␣a␣new␣one
                        ");
111
112                 behavior.addMovement(generateRandomWeights(behaviorsToActuators[
                        behavior.Name].Length,
113                           primitives.GetLength(0)));
114                 weights = behavior.getBehaviourMovements(pointCluster, relSig,
                        counter);
115             }
116
117      //Generate a movement by linear combbination of primitives with weights
118             generateMovement(behavior, weights, ref movements);
119
120             if (!isNetTrainingMode)
121                 BehaviorsTrainingStatus[behaviourList.IndexOf(behavior)] =
                        behavior.Training;
122         }
123
124         if (!isNetTrainingMode)
125         {
126             if (!BehaviorsTrainingStatus.Contains(true))
127                 this.isTrainingMotorMode = false;
128         }
129
```

```
130         this.counter++;
131
132     //Return the activation profiles of movements
133         return movements;
134       }
```

The choice of the best movement is done in Behaviour class, using the `getBehaviourMovements(...)` function, which returns the weights to be used in the linear combination of motor primitives.

```
93        /// <summary>
94        /// Gets movement from a behaviour
95        /// </summary>
96        /// <param name="inputState">The state coming from the intentional
              architecture.</param>
97        /// <param name="relevantSignal">The relevant signal coming from the
              intentional architecture.</param>
98      /// <param name="counter">counter of iterations</param>
99        /// <returns>A vector of weights for linear combination of primitives</
              returns>
100       public float[,] getBehaviourMovements(int inputState, float relevantSignal,
              int counter)
101       {
102           //Useful stuff
103           float rs = relevantSignal;
104           //float[] inSt = inputState;
105           int index;
106           float[,] movement;
107
108           //int key = computeHash(inSt);
109           Console.WriteLine("Current␣state␣cluster:␣{0}", inputState);
110
111           if (!bootStrapDone && tableInput.Keys.Count == 0)
112               return bootStrap(inputState);
113
114           //IF the state is not in the table, add it to that.
115           Console.WriteLine("Checking␣if␣its␣a␣known␣state...");
116
117           if (!tableInput.ContainsKey(inputState))
118           {
119               addState(inputState);
120               Console.WriteLine("Unknown␣state.");
121           }
122           else
123           {
124               Console.WriteLine("Alredy␣known␣state.");
125           }
126
127           //Update the values of the table.
128           //IMPORTANT: we need to update the values of the PREVIOUS state,
129           //based on the values of the CURRENT relevant signal,
130       //which depends from the LAST action performed.
131           updateTable(tMinusOneState, inputState, performedActionIndex, rs);
132
133           //Compute the index of the movement to be performed
134           index = getMovementIndex(inputState);
```

```
135
136         //IF index is -1, it is necessary to create a new movement (and test it)
137         if (index < 0)
138         {
139             Console.WriteLine("No suitable movement found");
140
141             return null;
142         }
143
144         //Get the array of weights corresponding to the selected movement
145         movement = movements.ElementAt(index);
146
147         //Current state = next previous state.
148     //Current index = last performed action index at next step.
149         tMinusOneState = inputState;
150         performedActionIndex = index;
151
152         tableDimsCounter.Add(new int[] { tableInput.Keys.Count, movements.Count
                 });//0,0
153         if (counter > this.thresholdTrain && isTrainingMotorMode)
154         {
155             saveTable();
156             saveCSV();
157             this.isTrainingMotorMode = false;
158         }
159
160         return movement;
161       }
```

A.4 Robot Interface and NAO robot

One of the main features of the implementation is its generality and adaptability to every type of agent. RobotInterface class gives an interface which allows to use the software with any agent.

```
10    /// <summary>
11    /// Interface defining the various methods called by Body,
12  /// and that have to be implemented in the Robot agent.
13    /// </summary>
14    /// <author>Burrafato Marco</author>
15    /// <author>Florio Luca</author>
16    /// <author>Franchi Alessio Mauro</author>
17    public interface RobotInterface
18    {
19
20        /// <summary>
21        /// Returns the input of a generic Sensor in an array format.
22        /// </summary>
23        /// <param name="sensorName">The name of the sensor.</param>
24        /// <returns>The sensor's input signal</returns>
25        Sinput getSensorInput(String sensorName);
26
27        /// <summary>
```

```
28      /// Sets the value for a specific actuator.
29      /// </summary>
30      /// <param name="actuatorsActivations">List of actuators and activation
            values</param>
31      /// <returns>1 if success, 0 if failure</returns>
32      bool setActuatorOutput(Dictionary<string, float[]> actuatorsActivations);
33
34      /// <summary>
35      /// Close the connection to the robot.
36      /// </summary>
37      /// <returns>True if the connection is closed, false otherwise.</returns>
38      bool disconnectFromRobot();
39
40      /// <summary>
41      /// Creates a connection between the software and the robot.
42      /// </summary>
43      /// <param name="ipAddress">Ip Address of the robot.</param>
44      /// <param name="port">Connection port of robot.</param>
45      /// <returns>True if connection estabilished, false otherwise.</returns>
46      bool connectToRobot(string ipAddress, int port);
```

The following is the implementation of the interface for a NAO robot.

```
54      /// <summary>
55      /// Returns the input of a generic Sensor in an array format.
56      /// </summary>
57      /// <param name="sensorName">The name of the sensor.</param>
58      /// <returns>The sensor's input signal.</returns>
59      public Sinput getSensorInput(String sensorName)
60      {
61          Sinput result;
62          switch (sensorName)
63          {
64              case "CameraTop":
65                  videoProxy.setParam(18, 0);
66           //Get the image from the Nao robot;
67                  img = (ArrayList)videoProxy.getImageRemote(videoProxyID);
68                  result = formatImage(img);
69                  break;
70              case "CameraBottom":
71                  videoProxy.setParam(18, 1);
72           //Get the image from the Nao robot;
73                  img = (ArrayList)videoProxy.getImageRemote(videoProxyID);
74                  result = formatImage(img);
75                  break;
76              case "HeadYaw":
77                  float[] yaw = new float[1];
78                  yaw[0] = motionProxy.getAngles("HeadYaw", false)[0];
79                  result = new Sinput(yaw);
80                  break;
81              case "HeadPitch":
82                  float[] pitch = new float[1];
83                  pitch[0] = motionProxy.getAngles("HeadPitch", false)[0];
84                  result = new Sinput(pitch);
85                  break;
86              case "RightArm":
```

```
87                  string[] joints = new string[] { "RShoulderPitch", "RShoulderRoll"
                        ,
88                          "RElbowYaw", "RElbowRoll" };
89                  float[] values = motionProxy.getAngles(joints, false).ToArray();
90                  Dictionary<string, float> jointValues = new Dictionary<string,
                        float>();
91                  jointValues.Add("RShoulderPitch", values[0]);
92                  jointValues.Add("RShoulderRoll", values[1]);
93                  jointValues.Add("RElbowYaw", values[2]);
94                  jointValues.Add("RElbowRoll", values[3]);
95                  result = new Sinput(jointValues);
96                  break;
97              case "US␣Sensor␣2"://TODO: add other types of sensors
98              case "US␣Sensor␣4":
99              case "MicroFront":
100                 result = new Sinput();
101                 break;
102             default:
103                 result = new Sinput();
104                 break;
105         }
106         return result;
107     }
```

```
138     /// <summary>
139     /// Sets the value of a specific actuator.
140     /// </summary>
141     /// <param name="actuatorsActivations">Actuators and corresponding
            activations</param>
142     /// <returns>1 if success, 0 if failure</returns>
143     public bool setActuatorOutput(Dictionary<string, float[]>
            actuatorsActivations)
144     {
145         bool result = true;//TODO
146         //List<String> joints;
147         //List<float> angles;
148
149         //Movememt quantum
150         float quantum = 0.05f;
151         //Take one value on two
152         int size = 25;
153         //Array of angles to interpolate
154         float[] listOfAngles;
155         //Interval time to perform interpolation
156         float[] listOfTimes;
157         //Names of joints
158         List<string> names = new List<string>();
159         //List of angles for each joint
160         List<float[]> values = new List<float[]>();
161         //List of times for each joints
162         List<float[]> times = new List<float[]>();
163         //Flag for agonist muscle movement
164         int agonist;
165
166         try
167         {
```

```
168             //We do this for each actuator
169             foreach (string name in actuatorsActivations.Keys)
170             {
171 
172                 //Initialization of variables
173                 names.Add(name);
174                 float time = 0.0f;
175                 float vel = 0.0f;
176                 listOfTimes = new float[size];
177                 listOfAngles = new float[size];
178 
179                 //IF the value of activation is negative, its not an agonist
180                 if (actuatorsActivations[name][5] < 0)
181                     agonist = -1;
182                 else
183                     agonist = 1;
184 
185                 //Compute time for interpolation
186                 for (int i = 0; i < listOfAngles.Length; i++)
187                 {
188                     //IF an activation is under 0.1, we set minimum velocity to 0.1
189                     if (Math.Abs(actuatorsActivations[name][i *
190                   (actuatorsActivations[name].Length / size)]) < 0.1f)
191                         vel = 0.1f;
192                     else
193                         vel = Math.Abs(actuatorsActivations[name][i *
194                   (actuatorsActivations[name].Length / size)]);
195 
196                     float tempMovement = agonist * quantum * (i + 1);
197                     float tempPosition = motionProxy.getAngles(name, true)[0];
198 
199         //Populate list of angles
200                     listOfAngles[i] = agonist * quantum * (i + 1);
201 
202                     //Time = space/speed (4 is a scale value, otherwise times will
                          be too high)
203                     time += (quantum / vel) / 4;
204 
205                     //Populate list of times
206                     listOfTimes[i] = time;
207                 }
208 
209                 //Add angles and times to lists
210                 values.Add(listOfAngles);
211                 times.Add(listOfTimes);
212             }
213 
214             //Perform movement
215             motionProxy.angleInterpolation(names, values, times, false);
216 
217             Console.WriteLine("Movement␣done");
218         }
219         catch
220         {
221             Console.WriteLine("Error␣in␣Movement!");
222             result = false;
```

```
223             }
224
225             return result;
226         }
```

```
272         /// <summary>
273         /// Close the connection to the robot.
274         /// </summary>
275         /// <returns>True if the connection is closed, false otherwise.</returns>
276         public bool disconnectFromRobot()
277         {
278             try
279             {
280                 videoProxy.unsubscribe("GVM_nao");
281             }
282             catch
283             {
284                 Console.WriteLine("Error␣in␣closing␣connection!");
285                 return false;
286             }
287             Console.WriteLine("Connection␣closed!");
288
289             //save data in csv
290             saveCSV();
291
292             return true;
293
294         }
```

```
228         /// <summary>
229         /// Creates a connection between the software and the robot.
230         /// </summary>
231         /// <param name="ipAddress">Ip Address of the robot.</param>
232         /// <param name="port">Connection port of robot.</param>
233         /// <returns>True if connection estabilished, false otherwise.</returns>
234         public bool connectToRobot(string ipAddress, int port)
235         {
236             Console.WriteLine("Trying␣to␣connect␣to␣robot...");
237             try
238             {
239                 motionProxy = new MotionProxy(ipAddress, port);
240                 behaviourProxy = new BehaviorManagerProxy(ipAddress, port);
241                 poseProxy = new RobotPoseProxy(ipAddress, port);
242                 textToSpeechProxy = new TextToSpeechProxy(ipAddress, port);
243                 videoProxy = new VideoDeviceProxy(ipAddress, port);
244                 touchProxy = new SensorsProxy(ipAddress, port);
245                 memoryProxy = new MemoryProxy(ipAddress, port);
246                 audioProxy = new AudioDeviceProxy(ipAddress, port);
247             }
248             catch
249             {
250                 Console.WriteLine("Cannot␣connect␣to␣NAO!");
251                 return false;
252             }
253
254             if (!subscribe()) return false;
```

```
255
256             Console.WriteLine("Everything␣connected!");
257             return true;
258         }
```

Appendix B

ICAlib library documentation

The Independent Component Analysis - ICA - is an algorithm designed to recover a set of independent signals from a set of the same signals mixed togheter. It is based on the assumption that every signal can be reconstructed basing on the combination a set of predefined signals, the independent components precisely [56, 57]. ICA is often used when there are two or more mixed signals in input and we want to know their original form, separating them from each other.

Figure B.1 shows an example of four different functioning signals mixed and reconstructed using our C# implementation of the ICA algorithm. Figure B.1a shows four basic signals that we used to obtain a normal noisy signal. The signals we used for testing are:

```
1//a sinusoid
2functionsToReturn[0, (int)i] = (double)Math.Sin(i/2);
3//a funny curve
4functionsToReturn[1, (int)i] = (double)Math.Pow((((i % 23) - 11) / 9), 5);
5//a saw tooth
6functionsToReturn[2, (int)i] = (float)(((i % 27) - 13) / 9);
7//an impulsive noise
8functionsToReturn[3, (int)i] = (float)(random.Next(0, 1) * 2 - 1) * Math.Log(((
     double)random.Next(1, 10)) / 10);
```

Figure B.1b shows four signals obtained from the four signals in figure B.1a with a weighted sum using random weights. These signals represent the normal mixed signals that could come as a sensory input and originate from different sources. Figure B.1c shows four different signals identified by the ICA algorithm, starting just from the signals shown in B.1b. As we can see the algorithm is able to isolate the independent components present in

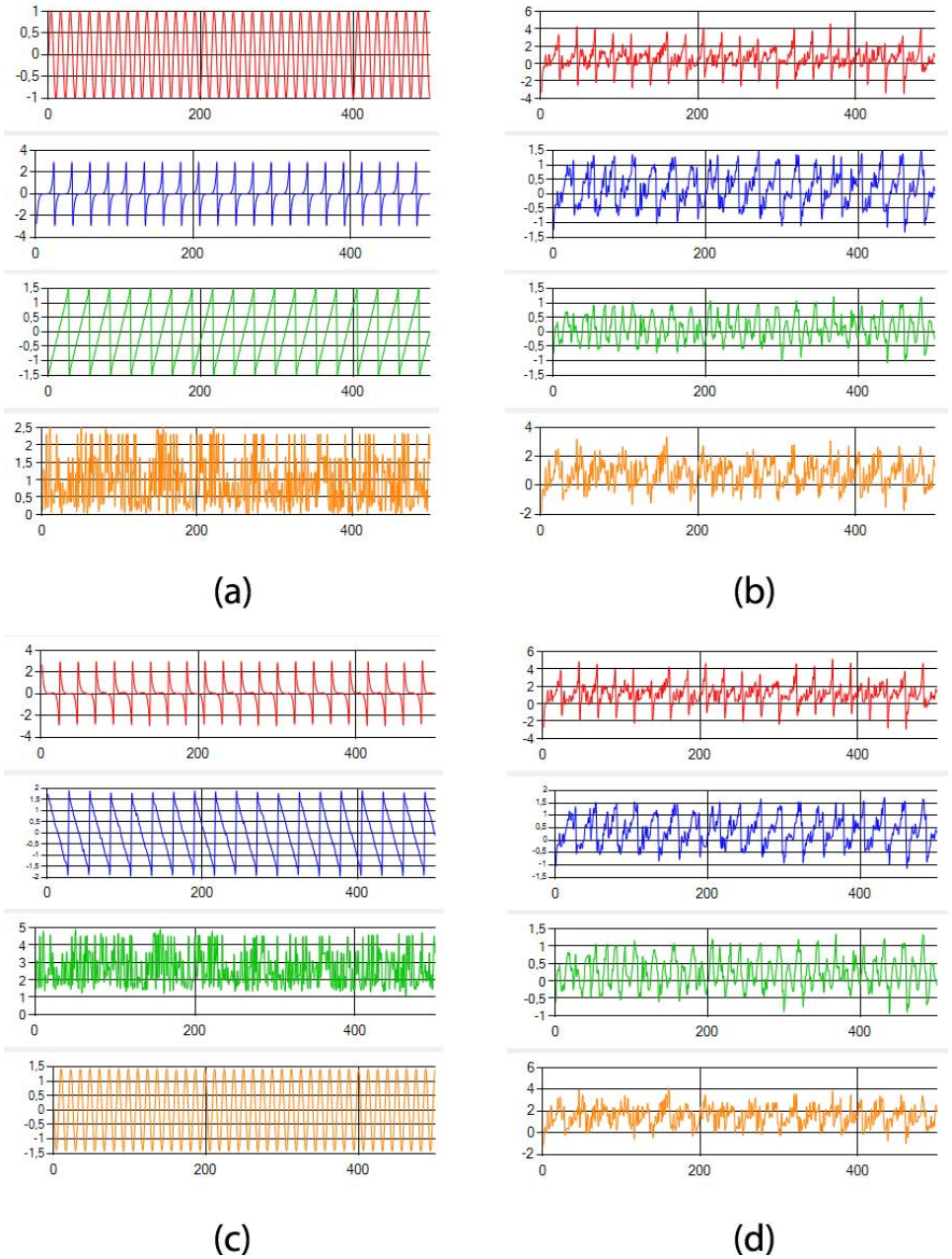


Figure B.1: ICA signals: (a) original signals; (b) mixed signals; (c) independent components; (d) reconstructed signals

the noisy input, and they indeed correspond to the four original signals that we mixed.
As explained in the next section, the independent components are obtained from the incoming signals applying a determined weights matrix. Using the inverse process we can re-obtain the original signals. Figure B.1d shows the incoming signals reconstructed using the independent components and the specific weight matrix we used before.
As we can see from Figure B.1c, however, the algorithm is not able to identify the correct amplitude of the independent component: while in Figure B.1a, for example the sinusoid amplitude is between -1 and +1, the amplitude of the corresponding independent component is between -1,4 and +1,4.
This is a known ambiguity of ICA [56], and it is due to the fact that, being both S and A unkwnown in Eq. B.2, any multiplier applied to s_i could be nullified by applying the same divisor to a_i.
For this reason, in our implementation of the Intentional Architecture we normalize the values of the independent components before applying them to the incoming input; that is, we assume each independent component has a variance $E(s_i^2) = 1$. Note that this leaves the ambiguity of the sign: we could have still the s_i signal multiplied by -1.

B.1 Principles of ICA functioning

Each incoming signal X_i can be viewed as a linear combination of predefined independent components:

$$X_i = \sum_{j=0}^{N} a_{ij} S_j \tag{B.1}$$

where S_j is an independent component signal, and a_{ij} is a specific weight to apply to a component in order to obtain the incoming signal. The sistem can be described as

$$X = AS \tag{B.2}$$

where each row of X is a signal X_i; each row of S is an independent component signal S_i; and A is an nxn mixing matrix that generates X from S. The goal of ICA is to find S and A given X.

We can consider each signal as a random variable; we want to find a matrix

$$W = A^{-1} \tag{B.3}$$

where

$$S = WX \tag{B.4}$$

this indicates that each independent signal S_i can be expressed as a linear combination of the signals in X.
Note that A must be invertible for $S = WX$ to be valid.

The ICA algorithm starts with the hypotesis that each row of S is statistically independent; that is, given two components s_1 and s_2 the information on the values of s_1 can not give any information on the values of s_2, and vice versa.
This can be expressed by:

$$P(s_1|s_2) = P(s_1) => P(s_1|s_2) = \frac{P(s_1, s_2)}{P(s_2)} => P(s_1, s_2) = P(s_1) * P(s_2) \tag{B.5}$$

where P(s_1,s_2) is the joint density function of s_1 and s_2.

Another property of independent random variables is that

$$E(f_1(s_1), f_2(s_2)) = E(f_1(s_1))E(f_2(s_2)) \tag{B.6}$$

for any given function f_1,f_2.

Another, weaker form of independence is uncorrelation. Two variables s_1 and s_2 are uncorrelated if their covariance is zero:

$$Cov(s_1, s_2) = E(s_1 * s_2) - E(s_1)E(s_2) = 0 \tag{B.7}$$

Another fundamental restriction in ICA is that the independent component can not be Gaussian.
The Central Limit Theorem states indeed that the weihgted sum of independent random variables tends towards a Gaussian distribution. Thus, $x_i = a_{i1}s_1 + a_{i2}s_2$ is closer to a gaussian distribution than s_1 or s_2.
This implies that we can find the combination of independent signals forming X only if they do not represent a Gaussian distribution already. This can be seen also from the Gaussian distibution graph, as shown in Figure B.2: the density is completely simmetric, making it impossible to determine any information about the direction of the columns in the mixing matrix A.
In order to find the independent signals, we need some way to measure the non-gaussianity of the S matrix.
To simplify things, we manipulate the data so that we have an incoming signal X with mean equal to zero and variance equal to 1. Then, we measure its

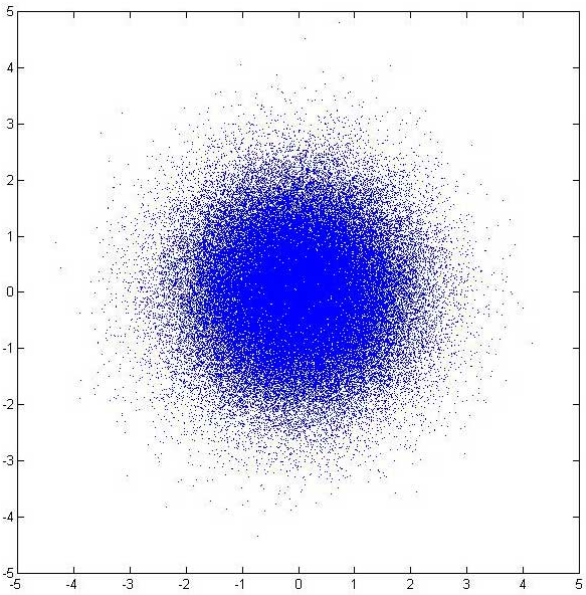


Figure B.2: An example of Gaussian distribution

non-gaussianity using negentropy. Negentropy is based on the information-theoretic quantity of differential entropy. Differential entropy for a continuous random vector variable y can be defined as [58, 59]:

$$H(y) = -\int f(y) log f(y) \tag{B.8}$$

where a_i is one of all the possible values of Y.
Negentropy J can be defined as

$$J(y) = H(y_{gauss}) - H(y) \tag{B.9}$$

where y_{gauss} is a Gaussian random variable of the same covariance matrix as y. Negentropy is always greater or equal to zero, and it is zero if and only if y has a Gaussian distribution. Thus negentropy is a statistical optimal estimator of non-gaussianity.
For our purpose, we will use an approximation of negentropy [60]:

$$J(y) \approx \sum_{i=1}^{p} k_i (E(G_i(y)) - E(G_i(v)))^2 \tag{B.10}$$

where k_i are some positive constants, and v is a Gaussian variable of mean zero and variance one - that is, it is standardized as well as y. G_i are some

non-quadratic functions. The choice of G is crucial for robust estimators and good approximations of negentropy; good choices are [61]:

$$g_1(u) = tanh(a_1 u) \quad , \quad g_2(u) = u\ e^{(-u^2/2)} \tag{B.11}$$

where $1 \leq a_1 \leq 2$ is a constant, usually equals to 1.
In our work, we used the hyperbolic tangent.

B.2 The algorithm

For our purposes, we implemented in C# language the FastICA algorithm presented by A. Hyvärinen and previously implemented in Matlab R language [61]. This algorithm is composed of several steps:

- *Preprocessing*: before applying the ICA algorithm to the incoming data X, we need to perform some actions to simplify the computation:
  - *Centering*: we assume that each X_i input has a mean value of zero. Thus, we subtract the mean from each row of X:

    $$X_i = X_i - E(X_i) \tag{B.12}$$

    Note that this implies that also each S_i will have zero mean, as we can see from Eq. B.2.
  - *Covariance*: we compute the covariance matrix of X, that is:

    $$Cov(X) = E(XX^T) \tag{B.13}$$

    The covariance matrix will be squared and symmetric. After this, we can compute the Singular Value Decomposition (SVD) on the covariance matrix, in order to obtain the diagonal matrix of eigenvalues D and the orthogonal matrix of eigenvectors E:

    $$Cov(X) = DED^T \tag{B.14}$$

  - *Whitening*: we transform each x vector linearly so that we obtain a new vector x_w that is white, i.e. its components are not correlated and their variance is equal to one. This also means that the covariance matrix of X_w is is the identity matrix:

    $$Cov(X_w) = E(X_w X_w^T) = I \tag{B.15}$$

Whitening can be performed with:

$$X_w = ED^{-1/2}E^T X \tag{B.16}$$

where $D^{-1/2} = diag(d_1^{-1/2}, ..., d_n^{-1/2})$. Whitening transforms the mixed signals X in a new matrix, A_w; the original matrix can be obtained from Eq. B.2 and B.16 as:

$$X_w = ED^{-1/2}E^T AS = A_w S \tag{B.17}$$

The new whitened matrix A_w is orthogonal; therefore:

$$Cov(X_w) = E(X_w X_w^T) = A_w E(SS^T)A_w^T = A_w A_w^T = I \tag{B.18}$$

and then whitening reduces the number of parameters we need to estimate.

- *FastICA algorithm*: for each independent component that we want to find, we define a weight vector w (notice that this implies the number of independent components must be known a piori by the algorithm; varying the number of IC can bring to different results. Here we assumed that the number of IC is equal to the number of incoming inputs). The algorithm learning rule finds a direction, that is converges to a vector w such that $s = w^T x$ maximizes the non-Gaussianity.
  The main loop of the algorithm is:

  1. Create a matrix of random weight values W, of the same dimensions of A_w;
  2. Since we have different signals, we have to define a vector w_i for each row of S; furthermore we have to ensure that different vectors do not converge to the same maxima; that is, we need to decorrelate each output $w_i^T x_i$ at every iteration.
     This can be performed using the Gram-Schmidt decorrelation:

$$\begin{aligned} w_{ij} &= \frac{w_{ij}}{\sqrt{\sum_{i=0}^{n} w_{ij}^2}} && \forall column\ j = 0, ..., n \\ w_{ik} &= w_{ij} - (w_{ik} * \sum_{i=0}^{n} w_{ik}^2) && \forall column\ k = j+1, ..., n \end{aligned} \tag{B.19}$$

  3. Check convergence as

$$\begin{aligned} &minAbsCos = min(abs(diag(W^T W_{old}))); \\ &if(1 - minAbsCos < epsilon) \\ &\quad break; \end{aligned} \tag{B.20}$$

where W_{old} is the value of W at the preceding step, and *epsilon* is a predefined value.

4. Update the weights of W: we orthogonalize W with respect to the previous components:

$$W = E(X\ G(X^T W)) - E(G'(W^T X))W \tag{B.21}$$

and then we normalize W

$$W = W/||W|| \tag{B.22}$$

5. Go back to point 1.
6. $S = WX_w$;

B.3 Conclusions

The implementation of this algorithm reveals a number of interesting properties [61]:

- the convergence is cubic or, at least, quadratic, while other ICA implementations based on gradient descent have a linear convergence; thus, the convergence here is much faster.
- In contrast to gradient based algorithms, there is no need for a step size parameter, making the algorithm simpler.
- The method can improve its performances choosing a suitable nonlinear function g. Functions shown in Eq. B.11 are good examples of optimal functions for robustness and minimum variance.
- the algorithm is able to find any IC of any non-Gaussian distribution using any nonlinear function G, differently from other algorithms that need an estimate of the probability distribution and a related nonlinearity function.
- it has many advantages of neural networks: it is parallel, distributed, computationally and memory inexpensive.

Appendix C

KMeansLib library documentation

K-Means is a clustering algorithm that allows the user to divide a group of objects in K partitions on the base of their values. These attributes could be represented as vectors, and they would form a vectorial space. The goal of the K-Means clustering algorithm is to minimize the total variance in clusters.
Each cluster is identified by a centroid. The algorithm is iterative:

1. create K partitions and randomly assign to each partition some points;
2. compute the centroid of each partition;
3. create a new partition assigning each point to the cluster with the nearest centroid;
4. Compute the centroid of the new clusters;
5. repeat until convergence.

In order to use this algorithm to perform clustering for categorization, we implemented a C# library.

C.1 KMeansLib

KmeansLib library allows the user to perform K-Means "standard", that is clustering on a set of points all at once, as well as "online", that is giving to the set of clusters one point per time and then recomputing the clusterization with the new point. The main function is `DoKMeans(List<Point> points, int clusterCount)`: it perform the standard K-Means algorithm described above.

```
286         /// <summary>
287         /// Compute K-Means clustering for a set of points.
288         /// </summary>
289         /// <param name="points">List of points to be clustered</param>
290         /// <param name="clusterCount">Desired number of clusters</param>
291         /// <returns>List of generated clusters</returns>
292         public List<Cluster> DoKMeans(List<Point> points, int clusterCount)
293         {
294             //divide points into equal clusters
295             List<Cluster> allClusters = new List<Cluster>();
296             List<List<Point>> allGroups = SplitList<Point>(points, clusterCount);
297 
298             //Set the maximum number of clusters equal to the desired number of
                    clusters
299             this.maxNumClusters = clusterCount;
300 
301             foreach (List<Point> group in allGroups)
302             {
303                 Cluster cluster = new Cluster(group, clusterIdCounter++);
304                 allClusters.Add(cluster);
305             }
306 
307             //start k-means clustering
308             int movements = 1;
309             while (movements > 0)
310             {
311                 movements = 0;
312 
313                 foreach (Cluster cluster in allClusters) //for all clusters
314                 {
315                 //for all points in each cluster
316                     for (int pointIndex = 0; pointIndex < cluster.Count; pointIndex++)
317                     {
318                         Point point = cluster[pointIndex];
319 
320                         int nearestCluster = FindNearestCluster(allClusters, point,
                                true);
321                         if (nearestCluster != allClusters.IndexOf(cluster)) //if point
                                has moved
322                         {
323                             if (cluster.Count > 1) //each cluster shall have minimum
                                    one point
324                             {
325                                 Point removedPoint = cluster.RemovePoint(pointIndex);
326                                 allClusters[nearestCluster].Add(removedPoint);
327                                 movements += 1;
328                             }
329                         }
330                     }
331                 }
332             }
333 
334             this.clusters = allClusters;
335 
336             return (allClusters);
337         }
```

To perform online K-Means, we override this function, so that the argument is a point. The distance from each centroid is computed, and if there is no centroid whose distance is less than a threshold, a new cluster is created. The set of cluster is then updated according to the new topology:

```
221         /// <summary>
222         /// Perform KMeans clustering
223         /// </summary>
224         /// <param name="point">Point to add to clusters</param>
225         /// <returns>List containing all the clusters</returns>
226         public List<Cluster> DoKMeans(Point point)
227         {
228             Console.WriteLine("Computing␣KMeans␣clustering...");
229             List<Cluster> allClusters = clusters;
230 
231             //IF there is no clusters, create the first
232             if (allClusters.Count == 0)
233             {
234                 Cluster newCluster = new Cluster(point, clusterIdCounter++);
235                 this.clusters.Add(newCluster);
236 
237                 this.pointCluster = newCluster.ID;
238                 Console.WriteLine("KMeans␣clustering␣done.");
239 
240                 return this.clusters;
241             }
242 
243             //Fiind the nearest cluster to the point
244             int nearestCluster = FindNearestCluster(this.clusters, point, false);
245             Console.WriteLine("Nearest␣cluster␣index:␣{0}", nearestCluster);
246 
247 
248             //IF the nearest cluster is not in distance threshold, create a new
                    cluster
249             //ELSE add the point to the nearest existing cluster
250             if (nearestCluster == -1)
251             {
252                 Cluster newCluster = new Cluster(point, clusterIdCounter++);
253                 this.clusters.Add(newCluster);
254                 //Update the distribution of points accordng to the new topology of
                        clusters
255                 this.clusters = UpdateClusters(this.clusters);
256                 this.pointCluster = newCluster.ID;
257             }
258             else
259             {
260                 this.pointCluster = clusters[nearestCluster].ID;
261 
262                 //IF the nearest cluster is not full, add the poit to it.
263                 foreach (Cluster cluster in allClusters)
264                     if (nearestCluster == allClusters.IndexOf(cluster) &&
265                 (maxNumPoints == 0 || cluster.Count < maxNumPoints))
266                     {
267                         cluster.Add(point);
```

```
268
269                     //IF the cluster is full, set the flag
270                     if (maxNumPoints != 0 && cluster.Count == maxNumPoints)
271                         cluster.IsFull = true;
272                 }
273
274             //IF the set of cluster is not fixed, update the position of points
275             if (maxNumClusters != 0 && clusters.Count == maxNumClusters)
276                 this.clusters = UpdateClusters(this.clusters);
277             else
278                 this.clusters = allClusters;
279         }
280
281         Console.WriteLine("KMeans␣clustering␣done.");
282
283         return (this.clusters);
284     }
```

We have implemented another useful function, `getDistances(Point p)`, used to compute the distances from a point to each centroid. This function is used, both in the Categorization Module and in the movement generation system, to compute the vector of neural activation.

```
349     /// <summary>
350     /// Get the distance of the point to each cluster.
351     /// </summary>
352     /// <param name="p">Point to be checked</param>
353     /// <returns>Distances of point p from each cluster centroid.</returns>
354     public float[] getDistances(Point p)
355     {
356         float[] distances = new float[this.clusters.Count];
357
358         for (int index = 0; index < this.clusters.Count; index++)
359         {
360             distances[index] = FindDistance(p, this.clusters[index].Centroid);
361         }
362
363         return distances;
364     }
```

C.2 Kmeans Simulator

In order to test the implemented K-Means library, we have realized a simple software that uses this library to perform clustering. The program allows the user to utilize the aforementioned functions for clustering one point at a time, or to create a defined number of clusters from a list of points.

C.2.1 Graphical interface

In the top left corner there are the settings for the algorithm: the threshold of distance from clusters, the maximum number of clusters and the maximum number of points per cluster.
Under the settings for K-Means algorithm, there are commands to add a single point at specific coordinates or to generate a random point. There is also a list about which points can be added. Under the list, a parameter could be set to specify the number of clusters to generate.
In the right side of the window, a graph is generated to show the computed clusters and their centroids. Each cluster has an ID and an unique color (Figure C.1).

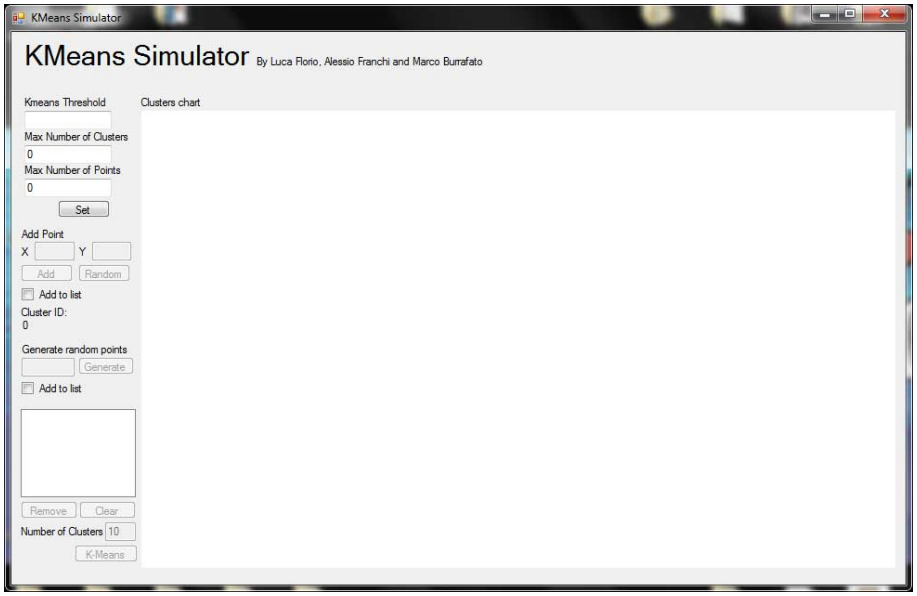


Figure C.1: KmeansLib simulator

C.2.2 Program usage

At the beginning, parameters for K-means algorithm must be set. The user can specify three parameters:

- Kmeans Threshold sets the threshold that represents the maximum distance from the centroid of a cluster. If the distance of the point is above this threshold, a new cluster should be created;
- Max Number of Clusters sets the maximum number of clusters to be

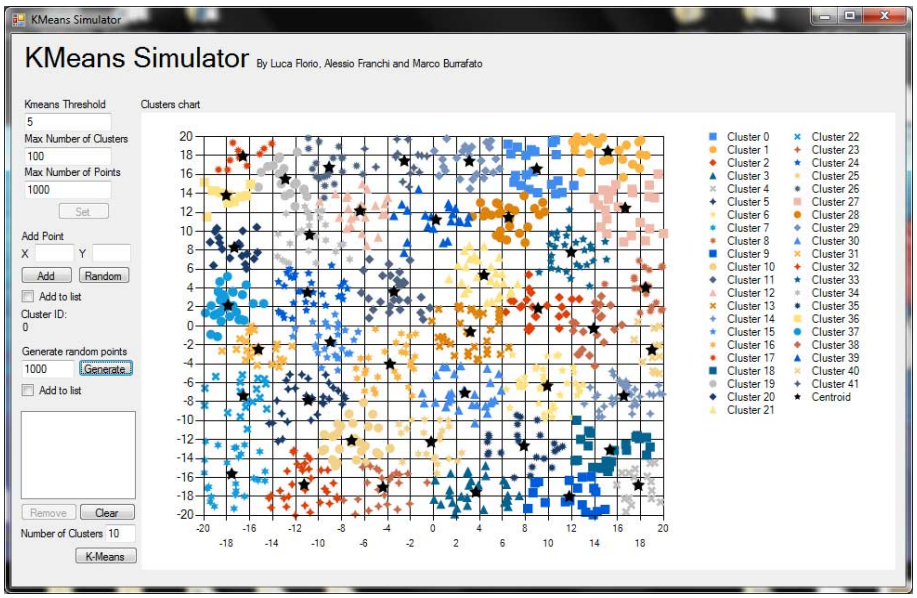


Figure C.2: K-Means clustering

generated. If the limit is reached, no new cluster is generated, but new points can be added to the nearest cluster;

- Max Number of Points sets the maximum number of points per cluster. If the limit is reached, no new points are added to that cluster, but the index of the nearest cluster is computed.

Once this parameters are set, the user can start performing clustering.
The user can generate a point by manually set the X and Y value of the point, as well as by using the random generator of points. There is also a command to generate a specified number of random points. Clustering could be performed in two different ways: in "online mode", where points are added one by one, or in "standard mode" where points are added to a list and then clustering is done on that list.
To perform clustering in standard mode, the points must be added to the list, by checking the "Add to list" checkBox. Once the number of clusters has been set, the "K-Means" button starts the computation of clustering.
Results are shown in the graph. Each cluster has a unique ID and color. The centroid of each cluster is represented as a black star (Figure C.2).

Appendix D

Behavior Simulation program documentation

Behavior Simulation is a software developed in order to study dynamic behaviors before their implementation in the system.
The implemented software simulates an input and a relevant signal, and computes the correct movement through linear combination of motor primitives. These primitives can be of two types: Gaussian functions or cosine functions.

D.1 Graphical interface

The graphical interface of the software in composed by a single window. In the top left corner there are two settings panels: the first is for primitives settings, the second is for K-Means clustering settings. There is also a checkbox labeled as "Nao", that if checked allows the user to test movements on a NAO robot. In the top right corner a chart allow to view the previously set primitives. Under the settings panels, there is the input state panel, which allows the user to interact with the system sending commands to it. Next to the input state panel, there is a listbox which prints useful information about the state of the system. In the bottom left corner we can find the State-Action table, associating a state and a movement to a relevant signal. In the bottom right corner there is a chart where the generated movement is drawn as a linear combination of primitives (Figure D.1).

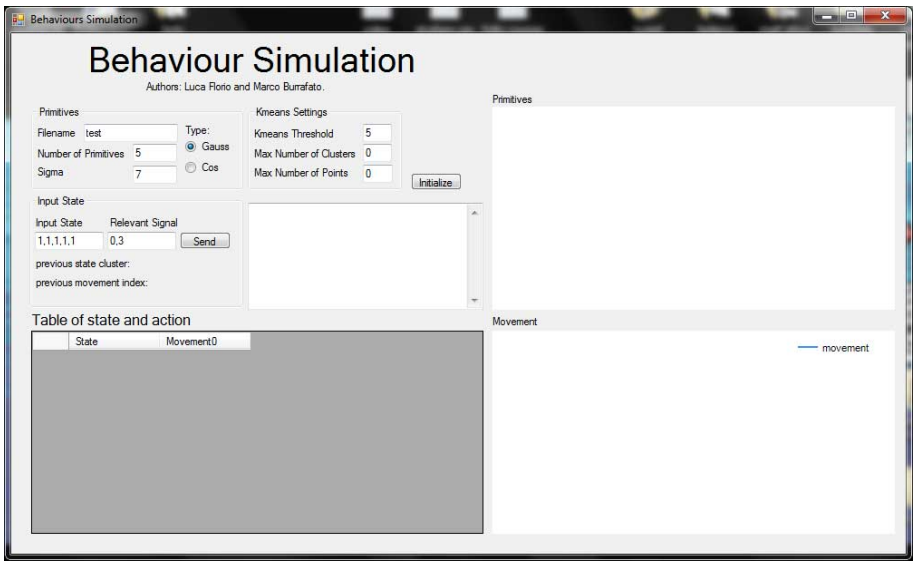


Figure D.1: Graphical interface

D.2 Program usage

Primitives settings allow the user to chose the XML behavior file to load, the shape of primitives and their number. In case of Gaussian primitives, the sigma value could be set manually.
K-Means settings allow to chose a threshold for the maximum distance from clusters, as well as the maximum number of clusters or the maximum number of points per cluster (setting these parameters as zero would mean to set them with no limits).
A checkbox labeled as "Nao", near K-Means settings, could be check to use a real or virtual Nao robot to perform movements.
Once the parameters are set, the initialize button starts the simulation and prints generated primitives in the top right chart.
During the simulation, the user can send an input to the program through the input state box. The user has to insert a five number input state (each number has to be separated by a comma, e.g. 1,1,1,1,1) and a floating point relevant signal associated to that state. Once the input is sent, the program performs clustering and updates the state-movement table in the bottom left corner. The best movement is selected, depending on state and signal, with this policy:

- if there is a movement not tried yet, that movement is chosen;
- if there is a movement already tried, with an associated relevant signal

higher than 0.8, that movement is chosen;

- if all the movements have been performed, but no one has an associated relevant signal higher than 0.8, a new movement is generated and tested.

A movement is intended as a linear combination of primitives.
After the best movement is selected, its shape is printed in the bottom right chart. If a NAO robot is connected, the movement is also performed by the robot (Figure D.2).

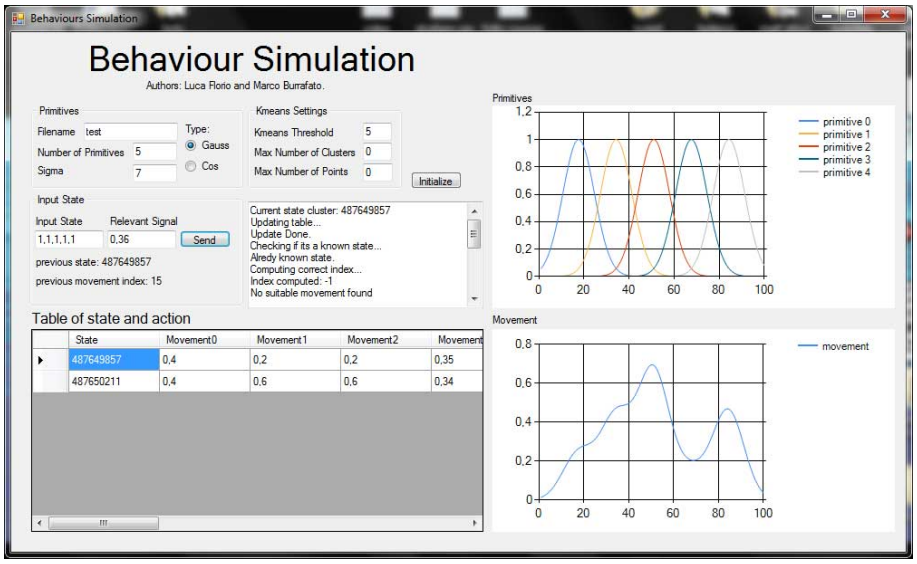


Figure D.2: Example of usage with Gaussian primitives