From imitation to action understanding: On the evolution of mirror neurons

Eelke Spaak
e.spaak@student.ru.nl

March 2008

Thesis in partial fulfilment of the requirements for the degree of Bachelor of Science in Artificial Intelligence

Supervisor:
Dr. W.F.G. Haselager

Co-supervisor:
Dr. I.G. Sprinkhuizen-Kuyper

Radboud University Nijmegen

Abstract

The mirror neuron system (MNS) has been described as the neural basis of action understanding, as the system responsible for the human capacity to imitate and as the crucial step in the evolutionary development that led to language. Understanding the evolutionary origins of the MNS will therefore likely provide much insight into what makes us human. The involvement of the MNS in both imitation and action understanding has been firmly established. Various authors have discussed the evolutionary origins of the MNS and claimed that its function in facilitating imitation builds upon its role in action understanding and is thus a phylogenetically later development. I argue, however, that this hypothesis lacks sufficient theoretical or empirical evidence and instead present support for the reverse: the phylogenetically primary function of the MNS is imitation and the MNS evolved in direct response to a selective pressure for imitative behavior. This hypothesis was tested using evolutionary robotics simulation techniques. The simulation was conducted with embodied and simulated-world embedded artificial agents equipped with a lifetime-adapting (i.e., Hebbian learning) neural network for which the learning parameters were subject to evolution. The agents had to perform an imitation task. Analysis of the neural controller that evolved in response to this task revealed artificial neurons showing clear mirror characteristics, suggesting that, indeed, mirror neurons evolve due to a selective pressure for imitative behavior.

Contents

1 Introduction

Mirror neurons are a particular class of visuomotor neurons, originally discovered in the macaque monkey, that fire both when the monkey performs a particular action and when the monkey sees that same action being performed by another individual (DiPellegrino, Fadiga, Fogassi, Gallese, & Rizzolatti, 1992; Gallese, Fadiga, Fogassi, & Rizzolatti, 1996; Rizzolatti, Fadiga, Gallese, & Fogassi, 1996). The discovery of the mirror neuron system has given an important impetus to diverse scientific disciplines. Neurobiologists and cognitive neuroscientists unravel its physiological properties more and more, while cognitive psychologists, linguists and even philosophers continue to apply its explanatory power to increasingly diverse classes of problems. The mirror neuron system (MNS) has been described as the neural basis of action understanding, as the system responsible for the human capacity to imitate and as the crucial step in the evolutionary development that ultimately led to modern language (see Rizzolatti & Craighero, 2004).

The evolutionary origin of the mirror neuron system itself remains, however, not well understood. Identifying the circumstances under which a system that seems so essential to our higher cognitive abilities has come to evolve will likely provide much insight into what makes us human.

In an attempt to characterize the MNS's evolution, Vilayanur Ramachandran (2000) claims that mirror neurons were a "pre-adaptation" and that the extensive role they play in the modern human mind should thus be considered an exaptation (i.e., an adaptation whose current function is not the function for which it originally evolved). The absence of any direct evidence concerning the MNS's evolution does not warrant any dismissal of this claim (nor does it lend it any credibility), but Ramachandran's hypothesis does leave open the question of which selective pressure *originally* led to the MNS's evolution.

An alternative hypothesis that attempts to address this question directly has been put forward by Elhanan Borenstein & Eytan Ruppin (2005). Using a computational model of evolution, they artificially introduced a selective pressure for imitation learning. The evolutionary process resulted in agents being born with neurons that had 'mirror-like' properties. These findings suggest that, when a selective pressure for imitation learning is present, i.e., when the capacity to imitate is beneficial to an individual, mirror neurons tend to evolve to meet this pressure.

Borenstein & Ruppin's agents are, however, radically disembodied and their simulation was not conducted embedded within a realistically simulated environment. This drastically reduces the applicability of their claim to real-life biological organisms. It is essential of biological mirror neurons that they are intimately intertwined with both visual and motor processing: they respond to observed bodily motion of another individual and 'mirror' this motion by resonating within the motor system of the observer. These characteristics were not taken into account in Borenstein & Ruppin's (2005) study.

In this thesis, I investigate the evolutionary origins of the mirror neuron system. In particular, I attempt to address the question of whether mirror neurons tend to arise in a population left to evolve under an evolutionary pressure for imitation learning. First, I briefly review the literature concerning the MNS and its properties. Next, I put forward a new computational model of the evolution of imitative behavior, in which artificial agents evolved to perform well on an imitation learning task. After this, I conclude by discussing the implications this new model has for the discussion concerning the MNS's evolution.

2 The mirror neuron system

Rizzolatti & Craighero (2004) have provided an extensive review of the literature concerning the functional properties of mirror neurons. I will not repeat their entire review here, but mention the points that I deem most important for the present thesis.

As stated in the introduction, mirror neurons are visuomotor neurons, originally discovered in area F5 of the macaque monkey brain, that fire both when an individual executes a goal-directed action and when that same action is observed. For the firing of mirror neurons, only the action itself is important; the exact specifics of the visual stimuli are irrelevant. Visual presentation of either a monkey's or a human's hand, or of a scene either near to or far from the monkey, will result in the same neurons' firing, as long as the represented action is the same.

The amount of specificity to stimuli, or 'congruence', as it has been called in the mirror neuron literature, differs among the mirror neurons. Cells always firing in response to observation and execution of an action with a certain goal (e.g., grasping) are called 'broadly congruent' mirror neurons, while cells responding more specifically to only a combination of a certain goal and a means of obtaining that goal (e.g., precision grip) are called 'strictly congruent' mirror neurons. Broadly congruent mirror neurons comprise about 60% of all mirror neurons in area F5; about 30% of F5 mirror neurons are strictly congruent. The remaining 10% of F5 neurons do respond to both visual presentation and execution of actions, but the action leading to their firing during execution and the action leading to their firing during observation show no clear relationship. These neurons were dubbed 'non-congruent mirror neurons' by Gallese et al. (1996).

In addition to containing mirror neurons responding to hand actions, when its more lateral part was investigated area F5 was also found to contain so-called 'mouth' mirror neurons. These come in two types: ingestive and communicative. Ingestive mirror neurons fire both when a monkey executes food-related mouth actions and when it observes them being executed. In monkeys, lip-smacking is a communicative action, and communicative mirror neurons fire when such an action is observed. They do not show strong activity when a monkey is performing such acts itself, but they do fire when a monkey is performing ingestive acts, so in a motor respect they are the same as the ingestive mirror neurons.

It has been suggested that "the communicative mouth mirror neurons found in F5 reflect a process of corticalization of communicative functions not yet freed from their original ingestive basis" (Rizzolatti & Craighero, 2004, p. 171) and while this is certainly an interesting hypothesis, a more apparent question seems to be whether these neurons, showing a congruence even far more loose than 'broadly congruent', should be called 'mirror neurons' at all. The same issue can be raised concerning the use of the term 'non-congruent mirror neurons' mentioned in the previous paragraph. This, of course, is a question of definition, but to avoid confusion of tongues it would not hurt to address it. However, since it is not particularly relevant for the remainder of this thesis I will not attempt to do so here.

2.1 The mirror neuron system in humans

Because of the invasiveness of the technique involved, there have not been extensive single-cell recording studies of possible mirror neurons in humans, as there have been in the macaque monkey. However, there are good indications that humans also possess a mirror neuron system.

A well-known electrophysiological property of the human cerebral cortex is the so-called μ-rhythm that occurs over the somatosensory areas in rest. When subjects perform an action, neural desynchronization occurs in these areas, resulting in the μ-rhythm becoming suppressed. Long before the MNS was discovered in monkeys, this phenomenon was also found to occur when human subjects only observed actions performed by others (Gastaut & Bert, 1954), suggesting involvement of cortical motor areas in action obervation.

More recent evidence has come from motor evoked potential (MEP) recordings after transcranial magnetic stimulation (TMS). Using this technique, TMS is applied to certain cortical areas, resulting in activation of related muscles in the body. The activation of these muscles is recorded as an MEP. The change in TMS-evoked MEP under certain conditions is taken as a measure of the excitability and, consequently, of the activity of the cortical areas being stimulated. It was found that the MEP of certain muscles showed an increase when subjects were presented with visual stimuli showing an individual moving these muscles (Fadiga, Fogassi, Pavesi, & Rizzolatti, 1995). It is interesting to note that this increase occurred both when a goal-directed action was observed (e.g., grasping) and when an action with no apparent goal, or *intransitive* action, (e.g., random arm gestures) was observed. This suggests that the response characteristics of the human MNS are somewhat different from those of the monkey MNS, since the latter only shows a clear response to goal-directed actions.

In an fMRI-study with human subjects, Buccino et al. (2004) tried to characterize the exact nature of the motor resonance exhibited by the MNS. They recorded brain activity during visual presentation of either a biting animal or an animal performing a communicative act. In both the biting and the communication condition, the action was performed by animals of three different species: a human, a monkey and a dog. For the human, the communicative act was muted speech; for the monkey, it was lip smacking; for the dog, it was barking. The activation in the brain areas most likely associated with the human MNS was the same during observation of all three species' biting. In the communicative act condition, the barking dog did not elicit any MNS activation, while the monkey and the human did. These findings confirm that indeed the MNS is involved in motor resonance and that this resonance is quite strict: since humans lack the musculature required for barking, they cannot bark, so there is no barking resonance.

2.2 Possible functions of the mirror neuron system

Rizzolatti, Fogassi, & Gallese (2001) have reviewed and extended upon two main hypotheses concerning the function of the MNS that had been previously put forward in the literature: action understanding and imitation. The proposed MNS mechanism for action understanding is quite simple: when an individual observes an action, the mirror neurons representing that action fire, resulting in activation of the observer's motor system. This activation is highly similar to the activation that occurs when the observer is actually executing that action and, since the observer knows (one hopes) the consequences of his own actions, he will understand the action as it is being performed by the other.

There are two important findings in the macaque monkey that support the view that the MNS is involved in action understanding. The first is that about 15% of F5 mirror neurons also fire when a monkey hears the sound of an action (e.g., the ripping apart of a piece of paper), with any visual stimuli being absent (Kohler et al., 2002). The second piece of evidence comes from an experiment by Umiltà et al. (2001). Two conditions of their experiment are important. In the first, a piece of food was put behind a screen while the monkey could see it being put there. Subsequently, a grasping action towards this piece of food was performed by a human. Mirror neurons were found to fire during this action, even though the final part of the action was invisible, due to the screen. In the second condition, the piece of food was put behind a screen while the monkey was unable to see this. The subsequent visual presentation of the grasping action was the same. No MNS activity was found during the observation of the action, suggesting that, because it did not know it was observing a goal-directed action, the monkey did not recognize it as such, this being reflected by the absence of mirror neuron firing.

In an attempt to determine whether or not the human MNS is involved in imitation and whether this involvement is dependent upon objects being present or not, Wohlschläger & Bekkering (2002) presented subjects with visual images of an index finger touching either the ipsi- or the contralateral item of a pair of dots glued to a table; the touched dot thereby acting as the object to which the action was directed. In another, control, condition, the dots were absent, while the presented finger movements remained the same. Subjects were instructed to imitate the stimulus presented. It was found that, in the 'dots' condition, (1) movement onset time for the ipsilateral index finger was reduced and (2) error rate was higher when movement of the contralateral finger was required. No such effects were observed in the 'dots absent' condition. These findings suggest that the human MNS is involved in imitation and that its activation is, at least in part, dependent upon object-directed action.

Summarizing this section so far, ample evidence can be said to exist for the MNS's involvement in both action understanding and imitation, in monkeys as well as in humans.

2.3 Evolutionary origins of the mirror neuron system

There has been quite some discussion of the MNS in relation to evolution, but this discussion has focused almost exclusively on the role the MNS might have played in catalyzing the phylogeny of other, higher, cognitive functions, most notably communication and language (e.g., Rizzolatti & Arbib, 1998) and the 'reading' of other people's mental states (e.g, Gallese & Goldman, 1998). The body of literature on the phylogenetic origins of the MNS itself is, as I have already stated in the introduction, quite a bit slimmer.

In order to understand the origins of any biological trait, one needs to consider the function it evolved to fulfil. This function can be identical to the function the trait fulfils today, in which case the trait is called an *adaptation*, or it can be different, in which case the trait with its current function is called an *exaptation*. The two functions the MNS most likely fulfils today, as described above, are action understanding and imitation, and Ramachandran (2000) seems to claim the MNS has been exapted to fulfil these functions. However, in contrast to his view, in the absence of any clear evidence I believe our best bet is to try and identify a function the MNS is an adaptation for.

Rizzolatti has claimed that the MNS's role in action understanding phylogenetically predates its role in imitation (Rizzolatti, 2005). However, it seems highly likely that imitation without understanding occurs in many animals, including humans. For instance, a flock of birds will often fly away in its entirety after one or two individuals have started to flap their wings (Thorpe, 1963) and newborn human and monkey babies are already able to imitate mouth gestures they observe (Meltzoff & Moore, 1979). In both cases, presuming a true understanding of the observed action seems questionable. Fortunately, Rizzolatti admits just this, but his solution to the problem seems even more problematic.

The solution Rizzolatti (2005, p. 76) proposes is to make a distinction between low-level and high-level resonance mechanisms, with the low-level mechanism having evolved much earlier than the high-level one and being responsible for the type of imitatory behaviors described above. The high-level resonance mechanism would have a "cognitive meaning", while this is lacking from the low-level mechanism. The neurons comprising the latter would be located "close to" the motor system, eliciting a motor response without any understanding occurring. The high-level resonance mechanism is the basis for action understanding and is, of course, the mirror neuron system as it was found in area F5. Rizzolatti claims that this view, which he does call "hypothetical", provides a unitary account for the different types of imitative behavior, but in my view it rather introduces a distinction that need not be there. First, there is no evidence of anatomical, physiological, or any other nature that suggests an additional mirror-like low-level mechanism that would be the basis for imitation without understanding. Second, while the MNS as a motor- and perceptually grounded basis for action understanding and imitation is a very elegant view, endowing one type of mirror system with "cognitive meaning" while depriving another of it negates much of this elegance – for whence could this extra meaning come from, if the basic mechanism is identical?

Since imitatory behavior is present in phylogenetically 'lower' species, I believe it plausible the function the MNS evolved to fulfil is imitatory behavior. The MNS should thus be considered an adaptation and its function in facilitating imitation should be considered primary and as having phylogenetically predated its function in action understanding. In order to test this hypothesis, I now put forward a computational model of the evolution of adaptive agents under selective pressure for imitation learning.

3 An evolutionary robotics model of the phylogeny of imitation

Evolutionary robotics is a technique for the development of artificial agents that meet certain requirements in a robust manner (see Nolfi & Floreano, 2000). The artificial agents have a body and a controller, the latter of which is usually composed of artificial neurons. Certain properties, most commonly concerning the makeup of the neural controller, but sometimes also concerning the bodily structure of the agent, are encoded into a string representation which is referred to as a 'genotype'. In a typical, so-called 'generational', evolutionary robotics experiment, a pool of genotypes is randomly initialized, after which an agent is successively created for each genotype and left to interact with the physical or a simulated world. Based on the behavior displayed by the agent, a fitness value is calculated according to some fitness function and assigned to the genotype the agent was a reflection of. After all the genotypes in the pool have been evaluated, some are selected, based on their fitness value, and left to reproduce. During this reproduction, genetic operations, such as random mutation and/or crossing-over, will usually take place, although exact reproduction of a few genotypes is also commonly used. This results in a new pool of genotypes ready to be evaluated in the same manner as the initial one. This process is repeated until either a specified number of generations have been evaluated or an individual with certain traits has evolved.

Because of the variation introduced by the genetic operations and the fitness-dependent selection, average and maximum fitness values tend to increase over time. In other words, agents get better at what they are supposed to do. Since natural evolution also proceeds with reproduction, random variation, and selection, evolutionary robotics provides an intuitively appealing way of modelling evolutionary processes, especially when the agents studied are either implemented on a physical robot or in a realistically simulated environment.

In this section, I put forward a model of the evolutionary processes that led to imitative behavior. First, the Framsticks environment, in which the simulation was conducted, is introduced. Next, the structure of the body and controller of the agents is described, as well as the environment they inhabit. Following this, the specifics of the evolutionary algorithm are given, after which I describe the results of the evolutionary simulation and analyze the evolved agents.

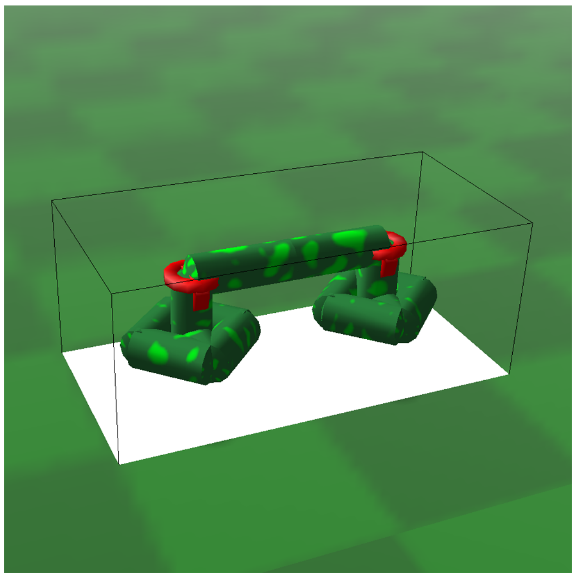
3.1 The Framsticks simulation platform

To conveniently implement the evolutionary model, the Framsticks platform was used (Komosinski, 2005). This platform has a number of properties that make it ideally suited for the study of artificial evolution. First, it realistically simulates both the physics of the actual world, allowing agents to be evaluated according to their behavior, as well as the dynamics of discrete-time neural networks of arbitrary complexity. Second, it comes in both a graphical user interface (GUI) version, that allows easy inspection of the simulation as it is being run, and a command line interface (CLI) version, that can be used for fast batch execution of multiple and/or long-lasting simulations. Third, it specifies a number of genetic 'languages' in which agents' properties, both concerning the neural controller and the bodily structure, can be described according to a strict syntax and from which simulated agents can automatically be created. Fourth, it provides automatic management of pools of genotypes, populations of creatures, and some commonly used genetic operations, allowing an experimenter to use these functions off-the-shelf. Finally, the Framsticks platform comes with a variety of different experiment definition files and artificial neuron implementations that an experimenter can use, or one can choose to implement completely new experiment definitions and/or neuron implementations in the Framscript scripting language. However, while this functionality greatly increases the flexibility of the platform, it is not without problems. In appendix A, a number of bugs and inconveniences are summed up that were encountered during the implementation of the current model.

3.2 The agent and its environment

3.2.1 Body The genotypes used in the model are specified in the Framsticks *f1* genotype language. This language allows simple specification of bodily structure in terms of parts and their joints, while also allowing complex control over the neural controller. The

Figure 1: A 3D rendering of the bodily structure of the agents. Clearly visible are the legs the agents use to propel themselves back and forth.



basic genotype of the agents was derived from the 2-legged rammer, a genotype available in the online Framsticks Experimentation Center in the genotype group called 'walking'[1]. The original version of this genotype codes for two legs and continuous walking in a single direction. The genotype was edited to allow adaptive synapses and multi-directional motion. In figure 1, a graphical depiction of the body of the agents used in the current model is shown. Clearly visible are the legs the agents use to propel themselves back and forth. Each leg is driven by a bend and a rotation muscle. The movement of the agents was simulated by the Framsticks Mechastick physics engine. The exact mechanics according to which muscles, joints, body parts and their interactions are simulated by this engine is beyond the scope of this thesis, but the interested reader is referred to the Framsticks Manual (Komosinski & Ulatowski, 2006).

3.2.2 Brain In figure 2, the layout of the agents' neural controller is shown. It is composed of four more or less separate modules: the sensors module, containing the four sensors the agent has to gather information about the world; the fully connected, fully recurrent adaptive network that is subject to evolution and in which lifetime learning can occur; a signal generator, responsible for generating a periodic signal that drives the walking motion of the agent; and a motor system, responsible for integrating the output of the adaptive network and the signal generator and actually moving the muscles accordingly. Between the adaptive network and the motor system a 'polarity conversion neuron' has

[1]The 2-legged rammer was created in 2000 by Miron Sadziak and is downloadable from `http://www.alife.pl/fec/www/index.php?PAGE=view_genotype&ID=67`

been included, converting the unipolar output (i.e., with values in the range $[0,1]$) of the adaptive network to a bipolar one (i.e., with values in the range $[-1,1]$). This conversion is done by computing $output = 2 \cdot input - 1$ and is necessary because the motor system requires bipolar activation values. The nature of the sensors and what they sense is described along with the agents' environment in section 3.2.3.

The neurons that are not in the adaptive network are all, with the exception of the polarity conversion neuron, of the Framsticks built-in 'simple neuron' type. The activation characteristics of these neurons are governed by three parameters: $force$ (real value in the range $[0,1]$), $inertia$ (real value in the range $[0,1]$), and $sigmoid$ (any real value). The $force$ parameter governs how fast the neuron responds to a change in its inputs; the $inertia$ parameter governs how fast neuron activation decays; and the $sigmoid$ parameter governs the shape of the neuron's output function. More precisely, at time t, the neuron's output is determined by the following set of equations:

$$input_t = \sum_{i=0}^{n} w_i o_{i,t-1} \tag{1}$$

$$velocity_t = velocity_{t-1} \cdot inertia + force \cdot (input_t - state_{t-1}) \tag{2}$$

$$state_t = state_{t-1} + velocity_t \tag{3}$$

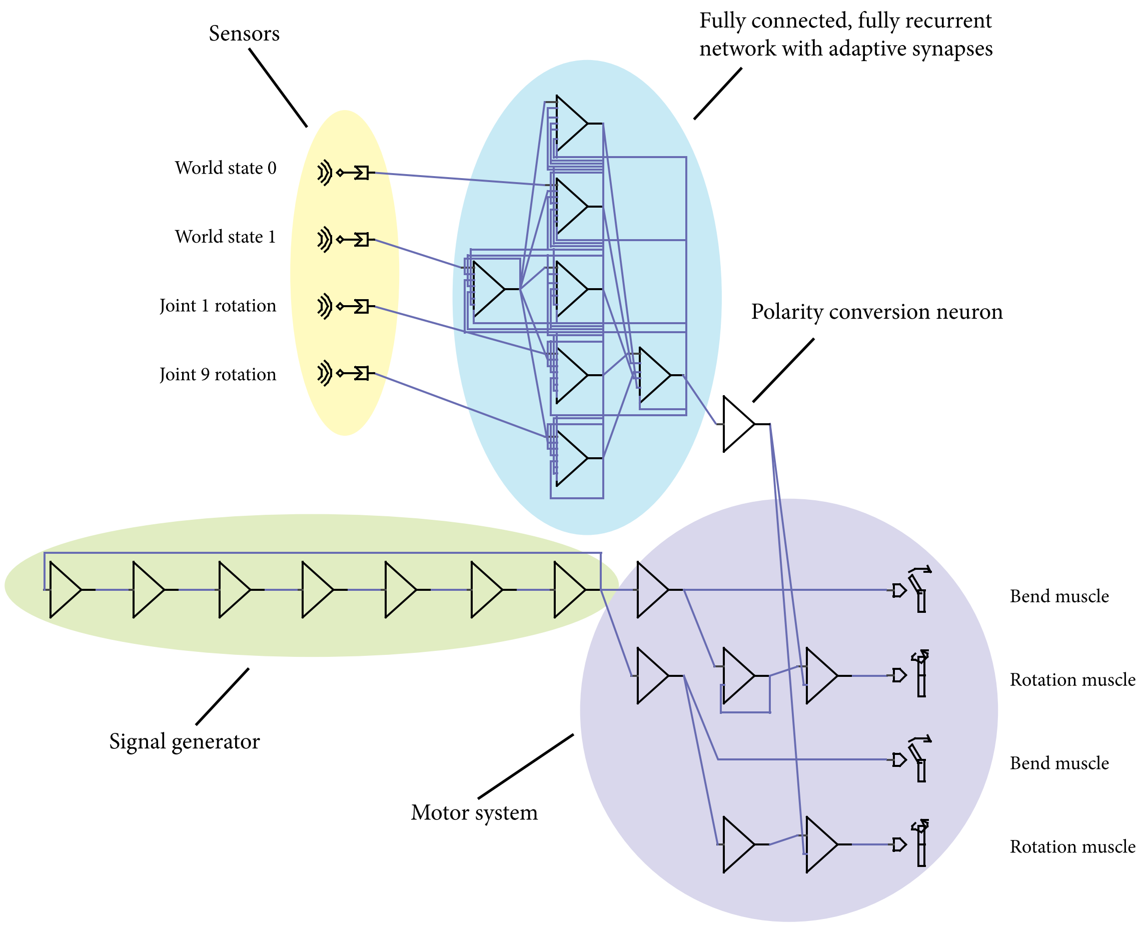
$$output_t = \frac{2}{1 + e^{-state_t \cdot sigmoid}} - 1 \tag{4}$$

Where w_i represents the weight of incoming connection i; $o_{i,t}$ represents the output of input neuron i at time t; and n represents the total number of inputs. These characteristics make neurons of this type good general-purpose non-linear neurons for use in evolutionary experiments, because both the incoming weights and the parameters governing their response characteristics can be genetically encoded. In the present model, however, all the weights concerning simple neurons were kept constant at the levels required for efficient locomotion and the parameters were kept at their default values: $inertia = 0.8$; $force = 0.04$; $sigmoid = 2.0$.

The seven neurons making up the adaptive network are of a completely different nature. Their incoming connections have weights that can be adapted according to Hebbian-like learning rules during the lifetime of an individual agent. This architecture is based on that proposed by Floreano & Urzelai (2000) and extended upon by Borenstein & Ruppin (2005). The exact manner in which lifetime learning can and will occur is not fixed, but rather determined through parameters subject to evolutionary optimization. Also subject to evolutionary optimization are the initial weights of these synapses, allowing for a mix of innate and acquired traits to express itself in the behavior of the agents. For a single neuron, each input synapse i is governed by four parameters:

- $w_{i,0}$ — the initial weight of the input (real value in the range $[0,1]$).
- s_i — the connection sign of the input (-1 or 1).

Figure 2: The layout of the artificial agents' neural controller. It is composed of four more or less separate modules: the sensors module, containing the four sensors the agent has to gather information about the world; the fully connected, fully recurrent adaptive network that is subject to evolution and in which lifetime learning can occur; a signal generator, responsible for generating a periodic signal that drives the walking motion of the agent; and a motor system, responsible for integrating the output of the adaptive network and the signal generator and actually moving the muscles accordingly. Between the adaptive network and the motor system a 'polarity conversion neuron' has been included, converting the unipolar output of the adaptive network to a bipolar one (see main text for details). A triangle symbol represents an artificial neuron. Note that, apart from a few manual edits to improve clarity, this graphical representation of the neural controller was created by the Framsticks GUI application and, because Framsticks has not rendered all the connections in the adaptive network symmetrically, this network might not look like a fully connected, fully recurrent network, while in fact it is.



- η_i — the learning rate for the synapse (real value in the range $[0,1]$).
- r_i — the learning rule for the synapse (integer value in the range $[0,4]$).

The connection sign is taken as a separate parameter because the learning algorithm requires positive weight values to function properly. At time t, the neuron's output is computed as follows:

$$input_t = \sum_{i=0}^{n} w_{i,k} s_i o_{i,t-1} \tag{5}$$

$$output_t = \frac{1}{1 + e^{-input_t}} \tag{6}$$

Where $o_{i,t}$ represents the output of input neuron i at time t and the subscript k is a time-dependent index whose relation to time governs how often synapses are adapted.

In the present model, the relation used was $k = \frac{t}{20}$. This results in the synaptic weights being updated every 20 time steps. This update happens according to the following formula:

$$w_{i,k} = w_{i,k-1} + \eta_i \Delta w_{i,k} \tag{7}$$

The value of $\Delta w_{i,k}$ is determined by the learning rule used for the particular synapse whose weight is being updated. The synaptic parameter r_i governs which learning rule is applied:

$r_i = 0$ — No learning:

$$\Delta w_{i,k} = 0 \tag{8}$$

$r_i = 1$ — Standard Hebbian learning:

$$\Delta w_{i,k} = (1 - w_{i,k-1}) o_{pre} o_{post} \tag{9}$$

$r_i = 2$ — Postsynaptic Hebb rule:

$$\Delta w_{i,k} = w_{i,k-1}(-1 + o_{pre}) o_{post} + (1 - w_{i,k-1}) o_{pre} o_{post} \tag{10}$$

$r_i = 3$ — Presynaptic Hebb rule:

$$\Delta w_{i,k} = w_{i,k-1} o_{pre}(-1 + o_{post}) + (1 - w_{i,k-1}) o_{pre} o_{post} \tag{11}$$

$r_i = 4$ — Covariance rule:

$$\Delta w_{i,k} = \begin{cases} (1 - w_{i,k-1}) \mathscr{F}(o_{pre}, o_{post}), & \text{if } \mathscr{F}(o_{pre}, o_{post}) > 0; \\ w_{i,k-1} \mathscr{F}(o_{pre}, o_{post}), & \text{otherwise;} \end{cases} \tag{12}$$

where $\mathscr{F}(o_{pre}, o_{post}) = \tanh(4(1 - |o_{pre} - o_{post}|) - 2)$.

In these formulas, o_{pre} is the activation of the presynaptic neuron and o_{post} is the activation of the postsynaptic neuron, both values averaged over the time steps since the last weight update (i.e., averaged over the last 20 time steps).

This type of adaptive synapses has been "based on neurophysiological findings (...) [and] these rules capture some of the most common mechanisms of local synaptic adaptation found in the nervous system of mammalians [*sic*]" (Floreano & Urzelai, 2000, p. 433).

For each adaptive synapse, each of its four properties are encoded onto the genotype and subject to evolutionary optimization. The adaptive network is fully connected and fully recurrent, i.e., each neuron receives incoming connections from each other neuron and from itself. In addition, four neurons receive sensory input, each from another sensor, and the output of one neuron is propagated to the polarity conversion neuron and, ultimately, to the motor system.

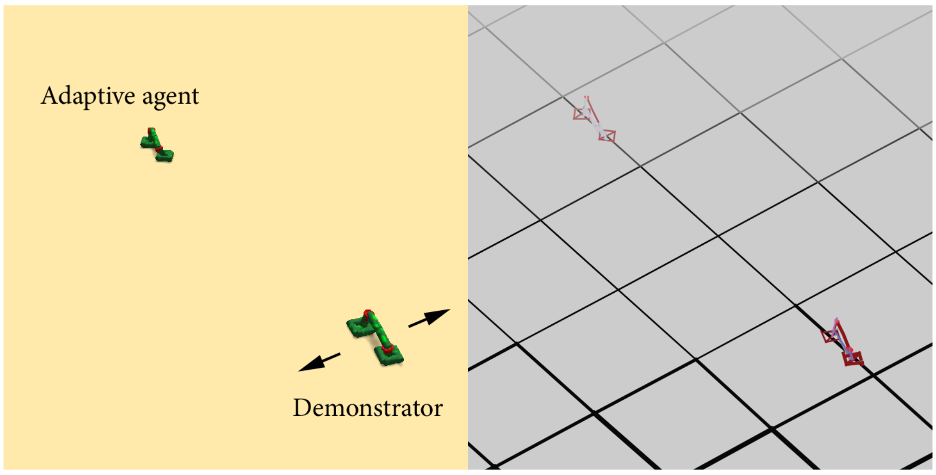
3.2.3 Task and environment The task the adaptive agents have to master during their lifetime is the execution of correct actions for given states of the world they inhabit. The correct action for each state is not initially known to the adaptive agents; they can only infer it by observing the action that is being executed by another, non-evolving, agent and learning to link this action to the state the world is currently in.

More specifically, the world the adaptive agents inhabit can be in two different possible world states, world state 0 and world state 1. Each world state has a different action associated with it, but which action is associated with which world state is not fixed. The two actions are walking (1) in north-east direction or (2) in south-west direction. The correct action for the current world state is always being executed by a demonstrator creature co-inhabiting the world with the adaptive agent (see figure 3 for a graphical representation of the artificial world).

The demonstrator creature has the exact same bodily structure as the adaptive agents. It is also equipped with a signal generator and a motor system, but it is lacking the sensors and adaptive network. Two versions of the demonstrator genotype were created: one equal to the original 2-legged rammer, always walking in north-east direction; and one edited to have it always walk in south-west direction. This allows the simulator to easily create the appropriate demonstrator creature when the world state changes.

As indicated in the previous section, the adaptive agent is equipped with two world state sensors with which it can sense the current state of the world. One sensor is responsive to world state 0, whereas the other responds to world state 1. When the actual world state is equal to the state the neuron is sensing for, the sensor's output will be 1.0. When this is not the case, the sensor's output will be 0.0.

Figure 3: A 3D rendering and a wireframe display of the artificial world inhabited by the adaptive agents. Apart from the agent subject to evolution, the demonstrator creature, capable of executing two different kinds of actions, is shown.



In addition to the world state sensors, the adaptive agents are equipped with two joint rotation sensors, sensing the rotation of certain joints in the demonstrator creature. More specifically, the rotation in the X-dimension is sensed for joints 1 and 9, respectively. These specifics were chosen after analysis of the demonstrator's locomotion revealed that these variables were the only ones showing variation with time; in other words, all the other joints were not controlled by muscles and were completely stiff. X-rotation was found to always lie within the interval $[-\frac{\pi}{2}, \frac{3\pi}{2}]$, and to make this compatible with the neural network, requiring activations in the range $[0, 1]$, this value is scaled. The scaled output of the joint rotation sensors is given by:

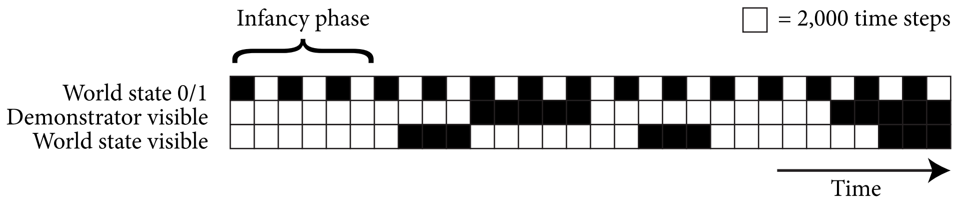
$$output = \frac{rotation}{2\pi} + \frac{1}{4} \tag{13}$$

It should be noted that, through their sensors, the adaptive agents have no privileged access to any of the internal workings of the demonstrator; only to its external behavior.

3.3 The evolutionary process

A population consisting of 20 adaptive agents, for which the properties of the adaptive network's synapses were randomly initialized, was subjected to evolutionary optimization. Each genotype was evaluated during two lifetimes, one for each possible world-state-to-action mapping. This was done to prevent the correct mapping from being genetically 'learned'. In the first lifetime, the correct action for world state 0 was walking in north-east direction and the correct action for world state 1 was walking in south-west direction, while this mapping was reversed in the second lifetime. At the beginning of each lifetime, the weights were initialized according to the initial weight values encoded in the genotype, so

Figure 4: A graphical display of the structure of the lifetime of a single agent. Shown are the world state (black represents state 0, white represents state 1), the visibility of the demonstrator creature and the visibility of the world state (black represents invisible, white represents visible). Note that, while the distribution of both world states across the world state visible/invisible condition is asymmetrical, this asymmetry does not confound the results (see section 3.4.3 for details).

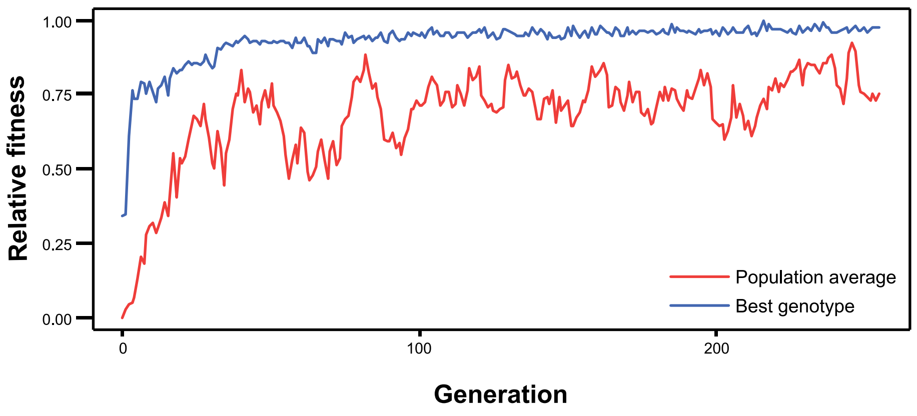


no properties acquired during the first lifetime could be expressed in the second (i.e., no Lamarckian evolution occurred). Each lifetime took 60 000 time steps, resulting in each genotype being evaluated for a total of 120 000 time steps. The world state was changed every 2 000 time steps. The world state was visible to the agent (i.e., the corresponding world state sensor's output was 1.0) in 70% of its lifetime and invisible (i.e., the output of both world state sensors was 0.0) in 30% of its lifetime. The demonstrator executing the proper action was visible to the agent (i.e., the output of the joint rotation sensors was the actual joint rotation) in 67% of its lifetime and invisible (i.e., the output of the joint rotation sensors was 0.0) in 33% of its lifetime. Furthermore, the world state and the demonstrator's joint rotation were always visible in the first 12 000 time steps of the agent's life, simulating an infancy phase. Figure 4 shows a graphical representation of the temporal structure of a single lifetime. These values ensure a good mix of different conditions is experienced during each lifetime. Note that, while the distribution of both world states across the world state visible/invisible condition is asymmetrical, this asymmetry does not confound the results (see section 3.4.3 for details).

During the evolutionary simulation of a single generation, each genotype is assigned an error score (for which lower is better), rather than a fitness value (for which, by definition, higher is better). Fitness values are, for technical reasons, linearly computed from these error scores only at the end of each generation. Except during infancy, an agent's error score is updated just before each world state change. The angle (in the range $[0, 2\pi)$) of the path traveled by the agent is computed, as well as the angle of the path traveled by the demonstrator creature. The squared difference between these values is added to the error score. When the world state is invisible, the error score is updated by instead adding half of the absolute distance traveled by the agent to it. Summarizing, the fitness function rewards agents that perform the correct action for a visible world state and do nothing at all when the world state is invisible.

At the end of a generation, the next generation is created as follows. First, two exact

Figure 5: The development of average and maximum fitness over time.



replicas are made of the best genotype in the pool. The remaining eighteen genotypes are created by selecting a parent individual based on its fitness value according to a roulette wheel selection mechanism and allowing it to reproduce. There is a 60% chance that a child genotype will be mutated and a 40% chance that it is an exact replica of its parent. A mutated genotype will have a single property value changed to a random value within the valid range for that property type.

3.4 Results

3.4.1 Fitness The evolutionary simulation was run until 256 generations had been evaluated. The development of the population's average and maximum fitness value is shown in figure 5. A linear regression analysis was performed to estimate the effect of generation on average fitness. This effect was found to be significant ($F(1, 254) = 165.201$, $p <$ .001) and strong ($r^2 = .394$). Average fitness increases with generation (standardized $\beta =$ .628). Also, a linear regression analysis was performed to estimate the effect of generation on maximum fitness. This effect was found to be significant ($F(1, 254) = 152.563$, $p < .001$) and strong ($r^2 = .375$) as well. Maximum fitness increases with generation (standardized $\beta = .613$).

These results indicate that the evolutionary simulation was successful; agents capable of imitative behavior have evolved.

3.4.2 Neurodynamics In order to determine whether or not neurons with mirror-like properties have evolved, the dynamics of the adaptive neural controller of the best individual of the last generation were analyzed. Mirror neurons are neurons that fire both when an action is executed and when that same action is only observed. Interpreted in terms of a neural network working with activation values rather than action potentials, a 'mirror neuron' is rather defined as a neuron showing the same activation pattern during execution of a particular action as during observation of that action.

During an agent's lifetime, it has to execute without observation when the world state is visible, but the demonstrator is invisible. Observation without execution occurs when the world state is invisible and the demonstrator is visible. Activation patterns of the seven adaptive neurons in both these conditions, for both world states, are shown in figure 6.

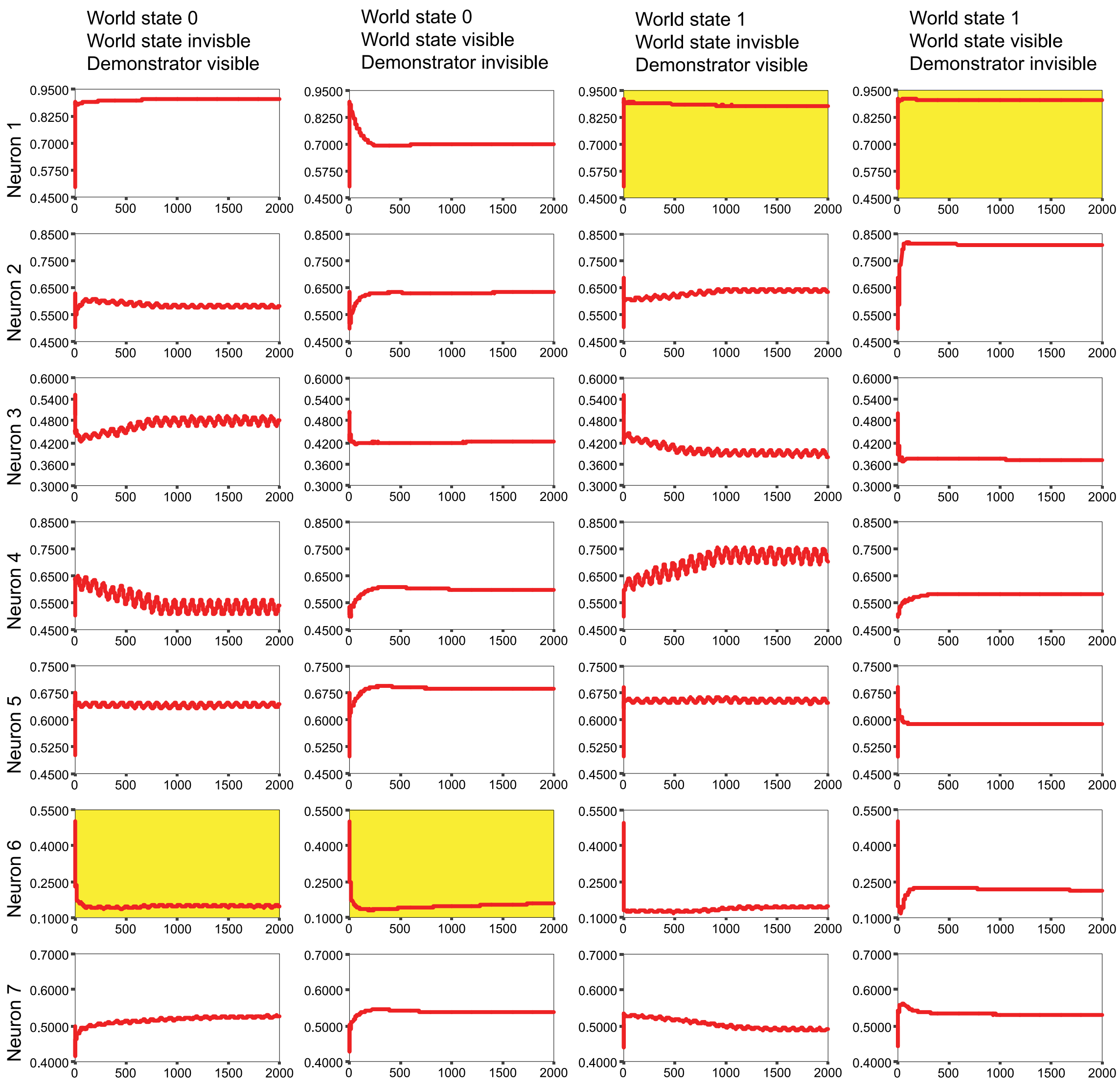
If a neuron had the exact same activation pattern in two conditions, linear regression analysis of the effect of the activation pattern in one condition on the activation pattern in the other condition would reveal a correlation of $r = 1.00$ and a regression line with an intercept of $B_{constant} = 0.00$. To quantify the mirror neuron definition in terms of statistical analyses of activation patterns, a mirror neuron is defined as a neuron showing a very strong correlation ($r > 0.85$) between its activation pattern in the 'observation, no execution' condition and its activation pattern in the 'execution, no observation' condition[2]. Furthermore, the regression line relating the two activation patterns is required to have an intercept close to zero ($B_{constant} < 0.05$). Calculating these measures for all the neurons in both world states revealed that neuron 6 satisfies the mirror neuron criteria in world state 0 and that neuron 1 satisfies the mirror neuron criteria in world state 1. No other neurons were found to satisfy the criteria. Table 1 shows all correlations and intercepts that were computed for this analysis.

3.4.3 Possible confounds The specific relative frequency of conditions mentioned in section 3.3 and shown in figure 4 results in the following distribution of world state occurrences, relative to the visibility of the demonstrator creature and the visibility of the world state. Relative to the visibility of the demonstrator, the occurrence of both world states is the same: during a single lifetime, both world states occur 10 times while the demonstrator is visible and both world states occur 5 times while the demonstrator is invisible. Relative to the visibility of the world state, however, the distribution is different: during a single lifetime, world state 0 occurs 12 times while the world state is visible and 3 times while the world state is invisible, whereas world state 1 occurs 9 times while the world state is visible and 6 times while the world state is invisible. The asymmetry of this distribution suggests it might confound the results.

To test if this is indeed the case, the error scores for each world state evaluation of the best individual of the last generation were analyzed. If the asymmetry described above were a confounding factor, the world state and/or the interaction between world state and world state visibility would have a significant effect on error score. A two-way analysis of variance was performed to estimate this effect and no significant overall effect ($F(3, 44) = 1.039, p = .385$) was found. The asymmetrical relative distribution of the two world states

[2]The exact interpretation of the relative strength of correlations is debatable. However, the most commonly accepted threshold for considering a correlation as 'strong' is $r > 0.50$ (Cohen, 1988), so defining a 'very strong' correlation as a correlation of $r > 0.85$ seems justified.

Figure 6: The activation levels of the seven neurons in the adaptive network, plotted for 2 000 subsequent time steps in which the environment parameters were the same. In each graph, the X-axis represents time and the Y-axis represents output. Notice that a different range of Y values is plotted for the different neurons. This was done to improve clarity. However, to avoid a distorted view of the results, the Y range was kept constant for each neuron between conditions. Highlighted are the neuron activations for which statistical analysis revealed they satisfy mirror neuron criteria (see main text for details).



	World state 0	World state 1
Neuron 1	$r = 0.256$	$r = 0.885$
	$B_{constant} = 0.962$	$B_{constant} = 0.030$
Neuron 2	$r = 0.204$	$r = 0.236$
	$B_{constant} = 0.507$	$B_{constant} = 0.525$
Neuron 3	$r = 0.186$	$r = 0.400$
	$B_{constant} = 0.072$	$B_{constant} = -0.069$
Neuron 4	$r = 0.387$	$r = 0.627$
	$B_{constant} = 1.166$	$B_{constant} = -0.672$
Neuron 5	$r = 0.137$	$r = 0.245$
	$B_{constant} = 0.589$	$B_{constant} = 0.491$
Neuron 6	$r = 0.910$	$r = 0.285$
	$B_{constant} = 0.033$	$B_{constant} = 0.092$
Neuron 7	$r = 0.453$	$r = 0.663$
	$B_{constant} = 0.108$	$B_{constant} = -0.333$

Table 1: The correlations between the activations of the neurons in the adaptive network in the 'observation, no execution' condition and the neuron activations in the 'execution, no observation' condition, as well as the intercepts of the regression lines relating those activation patterns.

across the world state visible/invisible condition does not confound the results.

4 Conclusion and discussion

The mirror neuron system (MNS), a system present in monkeys and most likely also in humans, forms an important basis for many cognitive functions. An understanding of its evolutionary origins will, therefore, bring us closer to understanding the human mind.

The model of the evolutionary origins of imitation put forward in the present thesis shows that an evolutionary pressure for imitative behavior results in artificial neurons emerging that have mirror-like properties. This suggests that the primary function of the MNS is imitation, rather than action understanding, and that this latter function is a phylogenetically later development that builds upon a capacity to imitate.

Ramachandran (2000) claims the MNS has been *exapted* to fulfil the functions of action understanding and imitation, but since there is no direct evidence concerning the MNS's origins, the best approach to understanding these origins is to try and identify a function the MNS is an *adaptation* for. The present model provides strong support for the hypothesis that the MNS is, in fact, an adaptation for imitatory behavior.

In addition, once the function the MNS fulfils in facilitating imitation is considered as phylogenetically primary, one can avoid the highly problematic distinction between low-level and high-level resonance mechanisms that Rizzolatti (2005) has proposed in order to explain imitatory behavior in 'lower' species.

While the present model is a lot more realistic than previous models (such as that put forward in Borenstein & Ruppin, 2005), there are still some important factors that can be improved. Most notably, the artificial neurons used in the model employ continuous activation levels, whereas biological neurons use the frequency of their action potentials to transfer information. Because of this, the original definition of a mirror neuron is not applicable to the artificial neurons used in the model: the artificial neurons do not have a firing rate, while biological mirror neurons are defined by their relative firing rate in different conditions.

Also, a question of a somewhat broader scope arises when considering whether the MNS has evolved in response to a selective pressure for imitation learning: did such a selective pressure ever occur during the evolutionary history of the primates? I believe this intuitively plausible, but hard evidence would be welcome. This evidence could very well come from biology, but I think it is also possible to address the question with models in computational cognitive science. Such models should obviously not introduce a selective pressure for imitation learning explicitly, but rather use a fitness function of a more basic nature (for instance, reproductive success). Populations can then be studied to see if imitative behavior is a rewarding trait in terms of fitness.

References

Borenstein, E., & Ruppin, E. (2005). The evolution of imitation and mirror neurons in adaptive agents. *Cognitive Systems Research*, *6*, 229–242.

Buccino, G., Lui, F., Canessa, N., Patteri, I., Lagravinese, G., Benuzzi, F., et al. (2004). Neural circuits involved in the recognition of actions performed by nonconspecifics: An fMRI study. *Journal of Cognitive Neuroscience*, *16*(1), 114–126.

Cohen, J. (1988). *Statistical power analysis for the behavioral sciences.* Hillsdale, New Jersey: Lawrence Erlbaum Associates.

DiPellegrino, G., Fadiga, L., Fogassi, L., Gallese, V., & Rizzolatti, G. (1992). Understanding motor events: a neurophysiological study. *Experimental Brain Research*, *91*(1), 176–180.

Fadiga, L., Fogassi, L., Pavesi, G., & Rizzolatti, G. (1995). Motor facilitation during action observation: a magnetic stimulation study. *Journal of Neurophysiology*, *73*(6), 2608–2611.

Floreano, D., & Urzelai, J. (2000). Evolutionary robots with on-line self-organization and behavioral fitness. *Neural Networks*, *13*, 431–443.

Gallese, V., Fadiga, L., Fogassi, L., & Rizzolatti, G. (1996). Action recognition in the premotor cortex. *Brain*, *119*(2), 593–609.

Gallese, V., & Goldman, A. (1998). Mirror neurons and the simulation theory of mind-reading. *Trends in Cognitive Sciences*, *2*(12), 493–501.

Gastaut, H. J., & Bert, J. (1954). EEG changes during cinematographic presentation; moving picture activation of the EEG. *Electroencephalography and Clinical Neurophysiology*, *6*(3), 433–44.

Kohler, E., Keysers, C., Umilta, M. A., Fogassi, L., Gallese, V., & Rizzolatti, G. (2002). Hearing sounds, understanding actions: Action representation in mirror neurons. *Science*, *297*, 846–848.

Komosinski, M. (2005). Framsticks: A platform for modeling, simulating, and evolving 3D creatures. In A. Adamatzky & M. Komosinski (Eds.), *Artificial life models in software* (pp. 37–66). London: Springer.

Komosinski, M., & Ulatowski, S. (2006). *Framsticks manual.* Retrieved December 20, 2007, from `http://www.frams.alife.pl/common/Framsticks_Manual.pdf`.

Meltzoff, A., & Moore, M. (1979). Imitation of facial and manual gestures by human neonates. *Science*, *205*, 217–219.

Nolfi, S., & Floreano, D. (2000). *Evolutionary robotics: The biology, intelligence, and technology of self-organizing machines.* Cambridge, Massachusets: MIT Press.

Ramachandran, V. S. (2000). Mirror neurons and imitation learning as the driving force behind "the great leap forward" in human evolution. *Edge*, *69*.

Rizzolatti, G. (2005). The mirror neuron system and imitation. In S. Hurley & N. Chater

(Eds.), *Perspectives on imitation: From neuroscience to social science* (pp. 55–76). Cambridge, Massachusets: MIT Press.

Rizzolatti, G., & Arbib, M. A. (1998). Language within our grasp. *Trends in Neurosciences*, *21*(5), 188–194.

Rizzolatti, G., & Craighero, L. (2004). The mirror neuron system. *Annual Review of Neuroscience*, *27*(1), 169–192.

Rizzolatti, G., Fadiga, L., Gallese, V., & Fogassi, L. (1996). Premotor cortex and the recognition of motor actions. *Cognitive Brain Research*, *3*(2), 131–141.

Rizzolatti, G., Fogassi, L., & Gallese, V. (2001). Neurophysiological mechanisms underlying the understanding and imitation of action. *Nature Reviews Neuroscience*, *2*(9), 661–670.

Thorpe, W. H. (1963). *Learning and instinct in animals*. London: Methuen.

Umiltà, M. A., Kohler, E., Gallese, V., Fogassi, L., Fadiga, L., Keysers, C., et al. (2001). I know what you are doing: A neurophysiological study. *Neuron*, *31*(1), 155–165.

Wohlschläger, A., & Bekkering, H. (2002). Is human imitation based on a mirror-neurone system? some behavioural evidence. *Experimental Brain Research*, *143*(3), 335–341.

Appendix A Problems with the Framscript scripting language

The scripting language used by Framsticks has a Javascript-like quasi-object-oriented syntax, at first sight allowing it to be used directly by programmers familiar with object-oriented programming and scripting languages. However, looks can be deceiving, since object instances and static class properties can be used interchangeably in Framscript, and altering a property of one can affect the other. For instance, while

```
LiveLibrary.getGroup(0).getCreature(0)
```

will return an object of type `Creature` which can be used in subsequent calls, the following:

```
LiveLibrary.group = 0;
LiveLibrary.creature = 0;
```

will select the same creature, after which its properties can be accessed and modified through static calls making no explicit reference to a single creature:

```
Creature.genotype;
// etc.
```

This is highly undesirable from an object-oriented point of view and can often be confusing when one is not accustomed to Framscript and its internal workings.

A.1 List of bugs encountered

The following is a list of the obvious bugs I encountered while scripting the various files required for the model described in the present thesis. The description is intended for readers with some experience with Framscript. However, readers lacking such experience but having other programming experience might also be able to appreciate the bugs.

- Sometimes, a `null` value will be considered as a string with contents "null", so testing for `null` values has to be done by testing:

  ```
  (x == null || x == "null")
  ```

- Reference updates within `Vector` objects (especially within the `fields` property of a `Neuro` object) sometimes fail. When or why this happens is unclear to me, but I have worked around this by implementing a construction like the following:

  ```
  do {
      n.fields.myProperty = value;
  } while (n.fields.myProperty != value)
  ```

- Floating point values with high positive or negative exponents sometimes find the sign of their exponent and/or coefficient flipped. This is of course extremely undesirable. I suspect this is due to a floating point under- or overflow, but Framsticks does not raise an error when this occurs, which it obviously should.

Appendix B Source code listings

The following pages list the source code of the relevant Framscript files making up the model described in the present thesis. Listed are:

- `imitlearn.expdef` — The main experiment definition script, controlling the flow of the simulation.
- `hebbian.neuro` — The implementation of the adaptive neurons.

Not included are the source listings of the following files, since they are all composed of only one or two lines of trivial code and their working has been completely specified in the main body text of this thesis:

- `convertpolarity.neuro` — The implementation of the polarity conversion neuron.
- `sense_joint.neuro` — The implementation of the joint sensors.
- `sense_worldstate.neuro` — The implementation of the world state sensors.

```
################################################################################
### imitlearn.expdef
###
### imitation learning Framsticks experiment definition
### created by Eelke Spaak, Nijmegen, NL, 2007
################################################################################

expdef:
name:Experiment for the study of imitation learning
info:~
Created to study imitation learning and the neural structures required for it.
~
#REGION Experiment code ########################################################
code:~

/*** Global variables *********************************************************/

// the two different possible world state to demonstrator genotype mappings
// Vector<String>
global g_mapping1, g_mapping2;

// the currently active mapping (each genotype will be evaluated twice, once
// for each possible mapping
// int [1,2]
global g_activeMapping;

// the file to store the results of each epoch in
// File
global g_resultFile;

// coordinates for the creatures, updated on each creation
// float
global g_demoBeginX, g_demoBeginY, g_learnerBeginX, g_learnerBeginY;

/*** Initialization functions *************************************************/

/**
 * Called by Framsticks when the experiment definition file is loaded.
 */
function onExpDefLoad() {
    log("onExpDefLoad");

    initializeGroups();
    initializeParameters();

    // initialize state-to-action mapping vectors
    g_mapping1 = Vector.new();
    g_mapping1.set(0, ExpParams.action1gen);
    g_mapping1.set(1, ExpParams.action2gen);

    g_mapping2 = Vector.new();
    g_mapping2.set(0, ExpParams.action2gen);
    g_mapping2.set(1, ExpParams.action1gen);

    g_activeMapping = 1;

    log("state-to-action mapping vector initialized");

    // initialize experiment state
    ExpState.epoch = 0;
    ExpState.current = 0;
    log("experiment state initialized");
}

/**
 * Called by Framsticks when the experiment should initialize itself.
 */
function onExpInit() {
    log("onExpInit");

    // clear groups
    LiveLibrary.clearGroup(0);
    GenotypeLibrary.clearGroup(0);
    GenotypeLibrary.clearGroup(1);

    if (!areParameterSettingsCompatible()) {
        Simulator.message("parameter settings are incompatible!", 2);
    }

    initializeResultFile();
}

/**
 * Creates the necessary creature groups and genotypes groups and sets their
 * parameters to appropriate values.
 */
function initializeGroups() {
    // clear groups
    GenotypeLibrary.clear();
    LiveLibrary.clear();

    // create genotype groups
    GenotypeGroup.name = "Current generation";
    GenotypeGroup.fitness = "return this.user1;";
    GenotypeGroup.fitfun = 0;
    GenotypeLibrary.addGroup("Previous generation");
    GenotypeGroup.fitness = "return this.user1;";
    GenotypeGroup.fitfun = 0;
    log("genotype groups created");

    // create learner creature group
    CreaturesGroup.name = "Learners";
    CreaturesGroup.nnsim = 1;
    CreaturesGroup.enableperf = 1;
    CreaturesGroup.death = 0;
    CreaturesGroup.energy = 0;
    CreaturesGroup.colmask = 1;

    // create demonstrator creature group
    LiveLibrary.addGroup("Demonstrator");
    CreaturesGroup.nnsim = 1;
    CreaturesGroup.enableperf = 1;
    CreaturesGroup.death = 0;
    CreaturesGroup.energy = 0;
    CreaturesGroup.colmask = 1;

    // create 'invisible' demonstrator creature group (for use in fitness
    // calculation)
    LiveLibrary.addGroup("Invisible demonstrator");
    CreaturesGroup.nnsim = 1;
    CreaturesGroup.enableperf = 1;
    CreaturesGroup.death = 0;
    CreaturesGroup.energy = 0;
    CreaturesGroup.colmask = 1;
    log("creature groups created");
}

/**
 * Initializes the experiment parameters and simulation parameters.
 */
function initializeParameters() {
    // the fixed demonstrator genotypes
    ExpParams.action1gen = "(RRX[6:-0.783][-1:2][-1:2][-1:2][-1:2][-1:2][-1:2]("
        + "RRlIMMMX[|-1:10][@0:0.737, -1:-10](RRlllX(fffX, , , , , , fffX), , R"
        + "RlllX(fffX, , , , , , fffX)), , ), , RRX(RRlIMMMX[|-3:-10][@!:0.046,"
        + " -1:10](RRlllX(fffX, , , , , , fffX), , RRlllX(fffX, , , , , , fffX)"
        + "), , ))";
    ExpParams.action2gen = "(RRX[6:-0.783][-1:2][-1:2][-1:2][-1:2][-1:2][-1:2]("
        + "RRlIMMMX[|-1:-10][@0:0.737, -1:10](RRlllX(fffX, , , , , , fffX), , R"
        + "RlllX(fffX, , , , , , fffX)), , ), , RRX(RRlIMMMX[|-3:10][@!:0.046, "
        + "-1:-10](RRlllX(fffX, , , , , , fffX), , RRlllX(fffX, , , , , , fffX)"
        + "), , ))";

    ExpParams.popsiz = 20;              // population size
    ExpParams.creath = 0.1;             // creatures are born at this height
    ExpParams.p_nop = 8;                // chance of copying genotype unchanged
    ExpParams.p_mut = 12;               // chance of copying genotype mutated
    ExpParams.p_xov = 0;                // chance of copying genotype x-ed over
    ExpParams.num_mut = 1;              // # of mutations per reproduction
    ExpParams.num_best = 2;             // # of exact copies of best individual

    sim_params.autosaveperiod = 1;      // automatically save every X epochs

    // do not mutate the creatures' morphology
    sim_params.f1_smX = 0.0;
    sim_params.f1_smJunct = 0.0;
    sim_params.f1_smComma = 0.0;
    sim_params.f1_smModif = 0.0;

    sim_params.f1_nmNeu = 0.0;          // no new neurons
    sim_params.f1_nmConn = 0.0;         // no new connections
    sim_params.f1_nmProp = 0.0;         // no new neural property settings
    sim_params.f1_nmWei =  0.0;         // no change in connection weights
                                        // (Hebbian weights are stored as
                                        // property values rather than weights)
    sim_params.f1_nmVal = 1.0;          // do change neural property values

    ExpParams.evaltime = 60000;         // lifetime of a single individual
    ExpParams.statetime = 2000;         // time of a single state
    ExpParams.infancytime = 12000;      // duration of infancy
    ExpParams.updatetime = 20;          // do Hebbian learning after X timesteps
    ExpParams.demovisiblenum = 10;      // # evaluations w/ visible demonstrator
    ExpParams.demoinvisiblenum = 5;     // # evaluations w/ inv. demonstrator
    ExpParams.wsvisiblenum = 7;         // # evaluations w/ visible world state
    ExpParams.wsinvisiblenum = 3;       // # evaluations w/ inv. world state

    ExpParams.debug = 1;                // print debug messages to console
    World.wrldsiz = 100;                // make the world big enough

    log("experiment parameters initialized");
```

```
}

/**
 * Checks whether the various 'time' experiment parameters are compatible, i.e.
 * whether a single lifetime encompasses an integer number of world state
 * changes and whether a creature's infancy fits within its lifetime.
 */
function areParameterSettingsCompatible() {
    return ExpParams.evaltime % ExpParams.statetime == 0
        && ExpParams.infancytime < ExpParams.evaltime;
}

/**
 * Creates a file to write each epoch's results to.
 */
function initializeResultFile() {
    var fileName = "results_" + String.format("%10.0f", Math.time) + ".txt";
    g_resultFile = File.createDirect(fileName);
    g_resultFile.writeString("RESULT:\tepoch #\tmin fitness\tmax fitness\tavg f"
    + "itness\tmin err\tmax err\tavg err\tbest genotype\r\n");
    g_resultFile.flush();
}

/*** Lifetime control functions ***********************************************/

/**
 * Called by Framsticks on each time step. This function is the main controller
 * for the course of the experiment.
 */
function onStep() {
    var lifetimeIndex = Simulator.time % ExpParams.evaltime;
    var isInfant = lifetimeIndex <= ExpParams.infancytime;

    if (lifetimeIndex == 0) {
        // a new individual should be created

        if (LiveLibrary.getGroup(0).creaturecount > 0) {
            // save the old creature's data
            updateCreatureFitness();
            updateGenotypePerformanceData();
            prepareNextLifetime();
        }

        prepareNextWorldStateAndCreateDemonstrator();
        createNewLearnerCreature();

    } else if (lifetimeIndex % ExpParams.statetime == 0) {
        // the world state should change

        // no fitness is calculated during infancy
        if (!isInfant) {
            updateCreatureFitness();
        }

        prepareNextWorldStateAndCreateDemonstrator(isInfant);
        resetLearnerCreature();
    }
}

/**
```

```
 * Updates the fitness value of the learner creature.
 */
function updateCreatureFitness() {
    updateCreatureFitnessExternal();
}

/**
 * Computes the angles of the paths traveled by both the demonstrator and the
 * learner. The squared difference of these angles is added to the user2 field
 * of the current creature. Also, the radius of the path traveled by the learner
 * is added to the user3 field of the current creature. If the world state is
 * invisible, half the radius traveled is added to user2, rather than the angle
 * squared error.
 */
function updateCreatureFitnessExternal() {
    var learner = LiveLibrary.getGroup(0).getCreature(0);
    var demonstrator = LiveLibrary.getGroup(1).getCreature(0);

    if (demonstrator == null) {
        demonstrator = LiveLibrary.getGroup(2).getCreature(0);
    }

    if (learner != null && demonstrator != null) {
        var demoEta = getAngle(demonstrator.center_x - g_demoBeginX,
            demonstrator.center_y - g_demoBeginY);
        var demoR = getRadius(demonstrator.center_x - g_demoBeginX,
            demonstrator.center_y - g_demoBeginY);
        var learnerEta = getAngle(learner.center_x - g_learnerBeginX,
            learner.center_y - g_learnerBeginY);
        var learnerR = getRadius(learner.center_x - g_learnerBeginX,
            learner.center_y - g_learnerBeginY);

        if (ExpState.worldstatevisible) {
            learner.user2 += getSquaredError(demoEta, learnerEta);
        } else {
            learner.user2 += (learnerR / 2.0);
        }

        learner.user3 += learnerR;
    }
}

/**
 * Adds the current creature's user fields' values to those of the current
 * genotype.
 */
function updateGenotypePerformanceData() {
    var learner = LiveLibrary.getGroup(0).getCreature(0);

    if (learner != null) {
        GenotypeLibrary.group = 0;
        GenotypeLibrary.genotype = ExpState.current;

        if (Genotype.user2 == null || Genotype.user2 == "null") {
            Genotype.user1 = 0.0;
            Genotype.user2 = 0.0;
            Genotype.user3 = 0.0;
        }

        Genotype.user1 += learner.user1;
        Genotype.user2 += learner.user2;
        Genotype.user3 += learner.user3;
    }
}

/**
 * Deletes the current demonstrator, updates the world state and
 * creates the appropriate demonstrator creature. This function also determines
 * whether or not the demonstrator should be visible and whether or not the
 * world state should be visible.
 */
function prepareNextWorldStateAndCreateDemonstrator(isInfant) {
    LiveLibrary.clearGroup(1); // clear demonstrator group
    LiveLibrary.clearGroup(2); // clear invisible demonstrator group
    LiveLibrary.group = 1;

    // determine next world state
    if (ExpState.worldstate == 0) {
        ExpState.worldstate = 1;
    } else {
        ExpState.worldstate = 0;
    }

    // determine visibility of demonstrator and world state (only during adult
    // life)
    var demoVisible;
    if (!isInfant) {
        var stateCounter = Simulator.time % ExpParams.evaltime
            / ExpParams.statetime;

        var demoVisibleIndex = stateCounter % (ExpParams.demovisiblenum
            + ExpParams.demoinvisiblenum);
        demoVisible = (demoVisibleIndex < ExpParams.demovisiblenum);
        if (!demoVisible) {
            LiveLibrary.group = 2;
        }

        var wsVisibleIndex = stateCounter % (ExpParams.wsvisiblenum
            + ExpParams.wsinvisiblenum);
        if (wsVisibleIndex >= ExpParams.wsvisiblenum) {
            ExpState.worldstatevisible = 0;
        } else {
            ExpState.worldstatevisible = 1;
        }
    } else {
        ExpState.worldstatevisible = 1;
        demoVisible = 1;
    }

    if (g_activeMapping == 1) {
        LiveLibrary.createFromString(g_mapping1[ExpState.worldstate]);
    } else {
        LiveLibrary.createFromString(g_mapping2[ExpState.worldstate]);
    }

    moveNewBornDemonstratorCreature();
}

/**
 * Sets the active world-state-to-demonstrator-mapping and current genotype
 * identifier to appropriate values for a new learner creature lifetime. This
 * function is responsible for each genotype having two lifetimes per epoch: one
 * for each possible world-state-to-demonstrator-mapping. If necessary, this
 * function moves the experiment into a new epoch.
 */
function prepareNextLifetime() {
    if (g_activeMapping == 2) {
        g_activeMapping = 1;
        ExpState.current++;

        if (ExpState.current >= GenotypeGroup.count) {
            computeFinalFitnessValues();
            onEpochEnd();
            prepareNextEpoch();
        }
    } else {
        g_activeMapping = 2;
    }
}

/**
 * Deletes the current learner creature and creates a new one.
 */
function createNewLearnerCreature() {
    GenotypeLibrary.genotype = ExpState.current;
    LiveLibrary.group = 0;
    LiveLibrary.creature = 0;
    LiveLibrary.delete();
    LiveLibrary.createFromGenotype();

    moveNewBornLearnerCreature();

    Creature.user1 = 0.0;
    Creature.user2 = 0.0;
    Creature.user3 = 0.0;
}

/**
 * Resets the learner creature to its original position, while preserving all
 * important values (performance data and lifetime-adapted weights).
 */
function resetLearnerCreature() {
    LiveLibrary.group = 0;
    LiveLibrary.creature = 0;

    // backup relevant creature values before creature deletion
    var velocity = Creature.c_velocity;
    var vertVelocity = Creature.c_vertvelocity;
    var distance = Creature.distance;
    var lifespan = Creature.lifespan;
    var avgVelocity = Creature.velocity;
    var avgVertVelocity = Creature.vertvel;
    var user1 = Creature.user1;
    var user2 = Creature.user2;
    var user3 = Creature.user3;

    // also retrieve the hebbian neuron configurations
    var weightsVectorsVector = Vector.new();
    var summedStatesVector = Vector.new();
    var summedInStatesVectorsVector = Vector.new();
    var labelsVector = Vector.new();
    retrieveWeightsAndLabels(weightsVectorsVector, summedStatesVector,
        summedInStatesVectorsVector, labelsVector);

    LiveLibrary.delete();
    GenotypeLibrary.genotype = ExpState.current;
    LiveLibrary.createFromGenotype();
    moveNewBornLearnerCreature();

    restoreCreatureInfo(velocity, vertVelocity, distance, lifespan, avgVelocity,
        avgVertVelocity, user1, user2, user3, weightsVectorsVector,
        summedStatesVector, summedInStatesVectorsVector, labelsVector);
}

/**
 * Updates the currently selected creature with the provided values.
 */
function restoreCreatureInfo(velocity, vertVelocity, distance, lifespan,
    avgVelocity, avgVertVelocity, user1, user2, user3, weightsVectorsVector,
    summedStatesVector, summedInStatesVectorsVector, labelsVector) {

    Creature.c_velocity = velocity;
    Creature.c_vertvelocity = vertVelocity;
    Creature.distance = distance;
    Creature.lifespan = lifespan;
    Creature.velocity = avgVelocity;
    Creature.vertvel = avgVertVelocity;
    Creature.user1 = user1;
    Creature.user2 = user2;
    Creature.user3 = user3;

    var i;
    for (i = 0; i < Creature.numneurons; i++) {
        if (weightsVectorsVector.get(i) != null) {
            var n = Creature.getNeuro(i);

            n.fields.inWeights = weightsVectorsVector.get(i);
            n.fields.myStateSum = summedStatesVector.get(i);
            n.fields.inStatesSum = summedInStatesVectorsVector.get(i);
            n.fields.label = labelsVector.get(i);

            // the following strange construction is necessary because
            // Framsticks sometimes does not handle reference updates in Fields
            // objects well; this is a bug work-around
            if (n.fields.inWeights != weightsVectorsVector.get(i)
                || n.fields.myStateSum != summedStatesVector.get(i)
                || n.fields.inStatesSum != summedInStatesVectorsVector.get(i)
                || n.fields.label != labelsVector.get(i)) {

                LiveLibrary.delete();
                GenotypeLibrary.genotype = ExpState.current;
                LiveLibrary.createFromGenotype();
                moveNewBornLearnerCreature();

                restoreCreatureInfo(velocity, vertVelocity, distance, lifespan,
                    avgVelocity, avgVertVelocity, user1, user2, user3,
                    weightsVectorsVector, summedStatesVector,
                    summedInStatesVectorsVector, labelsVector);

                break;
            }
        }
    }
}

/**
 * Scans the currently selected creature for any hebbian neurons and stores
 * their weights, averaged activations (required for learning) and labels in the
 * specified vectors. This function guarantees the vectors to be indexed such
 * that an element will be present at an index i iff the creature's neuron at
 * index i is a hebbian neuron.
 */
function retrieveWeightsAndLabels(weightsVectorsVector, summedStatesVector,
    summedInStatesVectorsVector, labelsVector) {
    var i;
    for (i = 0; i < Creature.numneurons; i++) {
        var n = Creature.getNeuro(i);

        var iObj = Interface.makeFrom(n.fields);
        var weightsInd = iObj.findId("inWeights");

        if (weightsInd != -1) {
            var weights = iObj.get(weightsInd);
            var summedState = iObj.get(iObj.findId("myStateSum"));
            var summedInStates = iObj.get(iObj.findId("inStatesSum"));
            var label = iObj.get(iObj.findId("label"));
            weightsVectorsVector.set(i, weights);
            summedStatesVector.set(i, summedState);
            summedInStatesVectorsVector.set(i, summedInStates);
            labelsVector.set(i, label);
        }
    }
}

/**
 * Moves the currently selected creature to an appropriate location for a
 * newborn learner creature.
 */
function moveNewBornLearnerCreature() {
    placeCurrentCreatureAtOffsetFromCenter(0.0, 10.0);
    g_learnerBeginX = Creature.center_x;
    g_learnerBeginY = Creature.center_y;
}

/**
 * Moves the currently selected creature to an appropriate location for a
 * newborn demonstrator creature.
 */
function moveNewBornDemonstratorCreature() {
    placeCurrentCreatureAtOffsetFromCenter(0.0, -10.0);
    g_demoBeginX = Creature.center_x;
    g_demoBeginY = Creature.center_y;
}

/**
 * Computes the final fitness values for each of the genotypes in the current
 * population. This can only be done after an entire generation is finished,
 * since the the calculation mechanism requires the minimum and maximum to be
 * known.
 */
```

```
function computeFinalFitnessValues() {
    GenotypeLibrary.group = 0;
    GenotypeLibrary.genotype = 0;

    var minE = Genotype.user2;
    var maxE = Genotype.user2;

    // compute maximum and minimum
    var i;
    for (i = 0; i < GenotypeGroup.count; i++) {
        GenotypeLibrary.genotype = i;

        if (Genotype.user2 < minE) {
            minE = Genotype.user2;
        } else if (Genotype.user2 > maxE) {
            maxE = Genotype.user2;
        }
    }

    // fill in the fitness values
    var val = maxE + minE;
    for (i = 0; i < GenotypeGroup.count; i++) {
        GenotypeLibrary.genotype = i;
        Genotype.user1 = val - Genotype.user2;
    }
}

/*** Epoch management functions ***********************************************/

/**
 * Called by lifetime control just before an epoch ends.
 */
function onEpochEnd() {
    outputEpochResults();
    Simulator.checkpoint();

    if (ExpState.epoch >= 10000) {
        Simulator.stop();
    }
}

/**
 * Computes the results of the current epoch and outputs them to the result
 * file.
 */
function outputEpochResults() {
    GenotypeLibrary.group = 0;
    var minF, maxF, sumF, minE, maxE, sumE, bestGenotype;

    GenotypeLibrary.genotype = 0;
    minF = Genotype.fit;
    maxF = Genotype.fit;
    sumF = Genotype.fit;
    minE = Genotype.user2;
    maxE = Genotype.user2;
    sumE = Genotype.user2;
    bestGenotype = Genotype.genotype;

    var i;
    for (i = 1; i < GenotypeGroup.count; i++) {
            GenotypeLibrary.genotype = i;

            if (Genotype.fit < minF) {
                minF = Genotype.fit;
            } else if (Genotype.fit > maxF) {
                maxF = Genotype.fit;
                bestGenotype = Genotype.genotype;
            }
            sumF += Genotype.fit;

            if (Genotype.user2 < minE) {
                minE = Genotype.user2;
            } else if (Genotype.user2 > maxE) {
                maxE = Genotype.user2;
            }
            sumE += Genotype.user2;
        }

        g_resultFile.writeString("RESULT:\t" + ExpState.epoch + "\t" + minF + "\t" +
             maxF + "\t" + (sumF / ExpParams.popsiz) + "\t" + minE + "\t" + maxE +
             "\t" + (sumE / ExpParams.popsiz) + "\t" + bestGenotype + "\n");
        g_resultFile.flush();
    }

    /**
     * Called by lifetime control to move the experiment into a new epoch. This
     * function copies all current genotypes to the 'Previous generation' genotype
     * group and initializes a new generation.
     */
    function prepareNextEpoch() {
        // clear the previous generation group and copy the current to the previous
        GenotypeLibrary.clearGroup(1);
        GenotypeLibrary.group = 0;
        var i;
        for (i = 0; i < GenotypeGroup.count; i++) {
            GenotypeLibrary.genotype = i;
            GenotypeLibrary.copyGenotype(1);
        }

        // clear the current generation group
        GenotypeLibrary.clearGroup(0);
        GenotypeLibrary.group = 0;

        // copy the best genotype a number of times
        while (GenotypeGroup.count < ExpParams.num_best) {
            GenotypeLibrary.group = 1;
            selectBestGenotype();
            GenotypeLibrary.copyGenotype(0);
            log("copying best genotype exactly, genotype=" + Genotype.genotype);
            GenotypeLibrary.group = 0;
        }

        // fill the rest of the population
        while (GenotypeGroup.count < ExpParams.popsiz) {
            GenotypeLibrary.group = 1;
            selectOrCreateGenotypeForNewGeneration();
            if (Genotype.isValid) {
                GenotypeLibrary.copyGenotype(0);
            }
            GenotypeLibrary.group = 0;
    }

    clearGenotypesPerformance();

    // next epoch
    ExpState.epoch++;
    ExpState.current = 0;
}

/**
 * Selects a genotype in the GenotypeLibrary that is either an exact copy or a
 * mutated version of an existing genotype, or a genotype created by crossing
 * over two existing genotypes. Which method is used is determined by the
 * related experiment parameters.
 */
function selectOrCreateGenotypeForNewGeneration() {
    var sel;
    sel = (ExpParams.p_nop + ExpParams.p_mut + ExpParams.p_xov) * Math.rnd01;
    if (sel < ExpParams.p_nop) {
        GenotypeLibrary.genotype = selectedForCreation();
    } else {
        sel = sel - ExpParams.p_nop;
        if (sel < ExpParams.p_mut) {
            GenotypeLibrary.genotype = selectedForCreation();
            performMutation();
        } else {
            GenotypeLibrary.genotype = selectedForCreation();
            GenotypeLibrary.crossover(selectedForCreation());
        }
    }
}

/**
 * Selects a genotype based on its fitness.
 */
function selectedForCreation() {
    return GenotypeLibrary.roulette();
}

/**
 * Selects the best genotype.
 */
 function selectBestGenotype() {
    var i, best, maxF;
    maxF = 0.0;

    for (i = 0; i < GenotypeGroup.count; i++) {
        GenotypeLibrary.genotype = i;
        if (Genotype.fit > maxF) {
            maxF = Genotype.fit;
            best = i;
        }
    }
    log("selected best: " + best);
    GenotypeLibrary.genotype = best;
 }

/**
 * Mutates the currently selected genotype a number of times. This number is
 * specified by the relevant experiment parameter.
 */
function performMutation() {
    var i;
    for (i = 0; i < ExpParams.num_mut; i++) {
        GenotypeLibrary.mutate();
    }
}

/**
 * Clears the performance of all the genotypes in the current generation.
 */
function clearGenotypesPerformance() {
    GenotypeLibrary.group = 0;

    var i;
    for (i = 0; i < GenotypeGroup.count; i++) {
        GenotypeLibrary.genotype = i;
        Genotype.user1 = 0.0;
        Genotype.user2 = 0.0;
        Genotype.user3 = 0.0;
        Genotype.fit = 0.0;
    }
}

/*** Miscellaneous functions **************************************************/

/**
 * Moves the currently selected creature to the specified offset from the center
 * of the world.
 */
function placeCurrentCreatureAtOffsetFromCenter(x,y) {
    Creature.moveAbs((World.wrldsiz - Creature.size_x) / 2 + x,
                     (World.wrldsiz - Creature.size_y) / 2 + y,
                     0);
}

/**
 * Returns the squared error between two angles, taking into account that a = a
 * + 2pi = a + 4pi etc.
 */
function getSquaredError(a1, a2) {
    var e = (a1 - a2) * (a1 - a2);
    if ((a1 - a2 - Math.twopi) * (a1 - a2 - Math.twopi) < e) {
        e = (a1 - a2 - Math.twopi) * (a1 - a2 - Math.twopi);
    } else if ((a1 - a2 + Math.twopi) * (a1 - a2 + Math.twopi) < e) {
        e = (a1 - a2 + Math.twopi) * (a1 - a2 + Math.twopi);
    }
    return e;
}

/**
 * Returns the angle theta (in the range [0,2pi) ) when converting the given
 * point to polar coordinates.
 */
function getAngle(x,y) {
    if (x > 0.0 && y >= 0.0) {
        return Math.atan(y / x);
    } else if (x > 0.0 && y < 0.0) {
        return Math.atan(y / x) + Math.twopi;
    } else if (x < 0.0) {
        return Math.atan(y / x) + Math.pi;
    } else if (x == 0.0 && y > 0.0) {
        return Math.pi2; // pi/2
    } else if (x == 0.0 && y < 0.0) {
        return Math.pi + Math.pi2; // 3pi/2
    } else {
        return -1;
    }
}

/**
 * Returns the radius R when converting the given point to polar coordinates.
 */
function getRadius(x,y) {
    return Math.sqrt(x * x + y * y);
}

/**
 * Outputs a message to the console if debugging is enabled.
 */
function log(msg) {
    if (ExpParams.debug) {
        Simulator.print("DEBUG: " + msg);
    }
}

/*** Saving/loading experiment state functions ******************************/
// Note: these functions were written by the Framsticks developers.

function onExpLoad() {
    GenotypeLibrary.group = 0;
    LiveLibrary.group = 0;
    GenotypeLibrary.clearGroup(0);
    GenotypeLibrary.clearGroup(1);
    LiveLibrary.clearGroup(0);
    Loader.addClass(sim_params.*);
    Loader.addClass(CreaturesGroup.*);
    Loader.addClass(GenotypeGroup.*);
    Loader.setBreakLabel (Loader.OnComment, "onExpLoad_Comment");
    Loader.setBreakLabel (Loader.BeforeUnknown, "onExpLoad_Unknown");
    Loader.run();
    if (GenotypeGroup.count > 0) {
        Simulator.print("Experiment loaded (" + GenotypeGroup.count +
            " genotypes)");
    }
}

function onExpLoad_Unknown() {
    if (Loader.objectName != "org") {
        return;
    }
    GenotypeLibrary.genotype = -1;
    Loader.currentObject = Genotype.*;
    Interface.makeFrom(Genotype.*).Interface:setAllDefault();
    Loader.loadObject();
    GenotypeLibrary.copyGenotype(0);
}

function onExpSave() {
    File.writeComment("'imitation_learning.expdef' data");
    File.writeObject(sim_params.*);
    GenotypeLibrary.group = 0;
    File.writeObject(GenotypeGroup.*);
    LiveLibrary.group = 0;
    File.writeObject(CreaturesGroup.*);
    GenotypeLibrary.group = 0;
    var i = 0;
    while (i++ < GenotypeGroup.count) {
        GenotypeLibrary.genotype = i;
        File.writeNameObject("org", Genotype.*);
    }
    Simulator.print("Experiment saved (" + GenotypeGroup.count + " genotypes)");
}

~
#END_REGION

#REGION Experiment parameters ##################################################

prop:
id:action1gen
name:Genotype for action 1 demonstrator
type:s 1

prop:
id:action2gen
name:Genotype for action 2 demonstrator
type:s 1

prop:
id:demovisiblenum
name:Number of subsequent world state evaluations with visible demonstrator
type:d 0 10

prop:
id:demoinvisiblenum
name:Number of subsequent world state evaluations with invisible demonstrator
type:d 0 10

prop:
id:wsvisiblenum
name:Number of subsequent world state evaluations with visible world state
type:d 0 10

prop:
id:wsinvisiblenum
name:Number of subsequent world state evaluations with invisible world state
type:d 0 10

prop:
id:updatetime
name:Number of time steps after which Hebbian neurons should 'learn'.
type:d 1 100000

prop:
id:popsiz
name:Gene pool size
type:d 1 2000

prop:
```

```
id:p_nop
name:Unchanged
type:f 0 100
group:Selection

prop:
id:p_mut
name:Mutated
type:f 0 100
group:Selection

prop:
id:p_xov
name:Crossed over
type:f 0 100
group:Selection

prop:
id:num_best
name:Number of exact copies of best individual
type:d 0 20

prop:
id:num_mut
name:Number of mutations per reproduction
type:d 1 50

prop:
id:evaltime
name:Evaluation time
type:d 100 100000

prop:
id:statetime
name:Timesteps per state
type:d 1 100000

prop:
id:infancytime
name:Duration of infancy phase
type:d 1 100000

prop:
id:creath
name:Initial elevation
type:f -1 50
help:~
Vertical position (above the surface) where newborn creatures are placed.
~

prop:
id:debug
name:Debug messages?
type:d 0 1 ~No~Yes
help:Print debug messages to the console, yes or no.

#END_REGION

#REGION Experiment state ######################################################

```

```
state:
id:notes
name:Notes
type:s 1
help:~
You can write anything here
(it will be saved to the experiment file)
~

state:
id:epoch
name:Generation number
type:d
flags:16

state:
id:current
name:Evaluating genotype
type:d
flags:16

state:
id:worldstate
name:Current world state
type:d
flags:16

state:
id:worldstatevisible
name:Is world state visible
type:d
flags:16

#END_REGION


```

```
################################################################################
### hebbian.neuro
###
### Framsticks neuron definition for hebbian learning neuron
### created by Eelke Spaak, Nijmegen, NL, 2007
################################################################################

class:
name:Hebb
longname:Hebbian
description:~
Neuron that exhibits Hebbian learning (in the adaptation of Floreano & Urzelai
(2000)). Output is between 0.0 and 1.0 (unipolar). This neuron takes a maximum
of 11 inputs.
~
prefinputs:-1
prefoutput:1
#REGION Neuron code ############################################################
code:~

/**
 * Called by Framsticks to make the neuron initialize itself.
 */
function init() {
    Fields.inWeights = Vector.new();
    Fields.inStatesSum = Vector.new();

    var i;
    for (i = 0; i < Neuro.getInputCount; i++) {
        Fields.inWeights.add(Fields.["w" + i]);
        Fields.inStatesSum.add(0.0);
    }

    // assign a random label to this neuron to help identify it during
    // analysis and debugging.
    Fields.label = "hebbneuron" + Math.random(1000);

    Fields.myStateSum = 0.0;
    Fields.updateInterval = ExpParams.updatetime;
}

/**
 * Called by Framsticks when this neuron needs to update its state.
 */
function go() {

    // determine activation level
    var weightedSum;
    weightedSum = 0.0;

    var i;
    for (i = 0; i < Neuro.getInputCount; i++) {
        var inState = Neuro.getInputState(i);
        var sign = Fields.["sign" + i];

        if (sign == 0) {
            weightedSum -= inState * Fields.inWeights.get(i);
        } else {
            weightedSum += inState * Fields.inWeights.get(i);
        }

        // store state of incoming neuron
        Fields.inStatesSum.set(i, Fields.inStatesSum.get(i) + inState);
    }

    var myState = sigmoid(weightedSum);

    // store state
    Fields.myStateSum += myState;

    // update incoming weights if necessary
    if (Simulator.time % Fields.updateInterval == 0 && Simulator.time > 0) {
        updateWeights();
    }

    // update neuron state
    Neuro.state = myState;
}

/**
 * Applies hebbian learning to each of this neuron's incoming weights. The
 * valued used for pre- and postsynaptic activation are averaged over a
 * specified time interval (not necessarily 1, so weights are not necessarily
 * updated each time step).
 */
function updateWeights() {
    var avgState = Fields.myStateSum / Fields.updateInterval;
    Fields.myStateSum = 0.0;

    var i;
    for (i = 0; i < Neuro.getInputCount; i++) {
        var avgInState = Fields.inStatesSum.get(i) / Fields.updateInterval;
        adaptWeight(i, avgInState, avgState);
        Fields.inStatesSum.set(i, 0.0);
    }
}

/**
 * Adapts the incoming weight at the specified index using hebbian learning with
 * the specified values for pre- and postsynaptic activation and the learning
 * parameters specified in this neuron's properties.
 */
function adaptWeight(index, preSynapticActivation, postSynapticActivation) {
    var deltaW, oldWeight, eta, r;
    oldWeight = Fields.inWeights.get(index);
    eta = Fields.["eta" + index];
    r = Fields.["r" + index];

    if (r == 0) {

        // no learning
        deltaW = 0.0;

    } else if (r == 1) {

        // plain hebb rule
        deltaW = (1.0 - oldWeight) * preSynapticActivation
            * postSynapticActivation;

    } else if (r == 2) {

        // postsynaptic hebb rule
        deltaW = oldWeight * (-1.0 + preSynapticActivation) *
            postSynapticActivation + (1.0 - oldWeight) * preSynapticActivation
            * postSynapticActivation;

    } else if (r == 3) {

        // presynaptic hebb rule
        deltaW = oldWeight * preSynapticActivation * (-1.0
            + postSynapticActivation) + (1.0 - oldWeight)
            * preSynapticActivation * postSynapticActivation;

    } else if (r == 4) {

        // covariance rule
        var F = tanh(4.0 * (1.0 - Math.abs(preSynapticActivation
            - postSynapticActivation)) - 2.0);
        if (F > 0.0) {
            deltaW = (1.0 - oldWeight) * F;
        } else {
            deltaW = oldWeight * F;
        }

    } else {
        Simulator.message("Hebbian r should be in the interval [0,4]!",
            2);
    }

    var newWeight = oldWeight + eta * deltaW;
    Fields.inWeights.set(index, newWeight);

    if (Fields.inWeights.get(index) > 1.0) {
        Fields.inWeights.set(index, 1.0);
    }
    // Framsticks has very buggy floating point handling with exponents larger
    // than +- 4, so, unfortunately, the following is necessary
    if (Fields.inWeights.get(index) < 1.0e-4) {
        Fields.inWeights.set(index, 0.0);
    }

}

/**
 * Returns the hyperbolic tangent of the specified number.
 */
function tanh(x) {
    if (x == 0.0) {
        return 0.0;
    }
    return (Math.exp(x) - Math.exp(-x)) / (Math.exp(x) + Math.exp(-x));
}

/**
 * Returns the sigmoid (or logistic) function of the specified number.
 */
function sigmoid(x) {
    return 1.0 / (1.0 + Math.exp(-x));
}

```

```
/**
 * Outputs a message to the console if debugging is enabled.
 */
function log(msg) {
    if (ExpParams.debug) {
        Simulator.print("DEBUG: " + msg);
    }
}

~
#END_REGION

#REGION Neuron properties #####################################################
prop:
id:eta0
name:learning rate for incoming connection 0
type:f 0.0 1.0

prop:
id:eta1
name:learning rate for incoming connection 1
type:f 0.0 1.0

prop:
id:eta2
name:learning rate for incoming connection 2
type:f 0.0 1.0

prop:
id:eta3
name:learning rate for incoming connection 3
type:f 0.0 1.0

prop:
id:eta4
name:learning rate for incoming connection 4
type:f 0.0 1.0

prop:
id:eta5
name:learning rate for incoming connection 5
type:f 0.0 1.0

prop:
id:eta6
name:learning rate for incoming connection 6
type:f 0.0 1.0

prop:
id:eta7
name:learning rate for incoming connection 7
type:f 0.0 1.0

prop:
id:eta8
name:learning rate for incoming connection 8
type:f 0.0 1.0

prop:
id:eta9
```

```
name:learning rate for incoming connection 9
type:f 0.0 1.0

prop:
id:eta10
name:learning rate for incoming connection 10
type:f 0.0 1.0

prop:
id:r0
name:learning rule for incoming connection 0
type:d 0 4

prop:
id:r1
name:learning rule for incoming connection 1
type:d 0 4

prop:
id:r2
name:learning rule for incoming connection 2
type:d 0 4

prop:
id:r3
name:learning rule for incoming connection 3
type:d 0 4

prop:
id:r4
name:learning rule for incoming connection 4
type:d 0 4

prop:
id:r5
name:learning rule for incoming connection 5
type:d 0 4

prop:
id:r6
name:learning rule for incoming connection 6
type:d 0 4

prop:
id:r7
name:learning rule for incoming connection 7
type:d 0 4

prop:
id:r8
name:learning rule for incoming connection 8
type:d 0 4

prop:
id:r9
name:learning rule for incoming connection 9
type:d 0 4

prop:
id:r10
```

```
name:learning rule for incoming connection 10
type:d 0 4

prop:
id:sign0
name:sign of incoming connection 0 (0=neg, 1=pos)
type:d 0 1

prop:
id:sign1
name:sign of incoming connection 1 (0=neg, 1=pos)
type:d 0 1

prop:
id:sign2
name:sign of incoming connection 2 (0=neg, 1=pos)
type:d 0 1

prop:
id:sign3
name:sign of incoming connection 3 (0=neg, 1=pos)
type:d 0 1

prop:
id:sign4
name:sign of incoming connection 4 (0=neg, 1=pos)
type:d 0 1

prop:
id:sign5
name:sign of incoming connection 5 (0=neg, 1=pos)
type:d 0 1

prop:
id:sign6
name:sign of incoming connection 6 (0=neg, 1=pos)
type:d 0 1

prop:
id:sign7
name:sign of incoming connection 7 (0=neg, 1=pos)
type:d 0 1

prop:
id:sign8
name:sign of incoming connection 8 (0=neg, 1=pos)
type:d 0 1

prop:
id:sign9
name:sign of incoming connection 9 (0=neg, 1=pos)
type:d 0 1

prop:
id:sign10
name:sign of incoming connection 10 (0=neg, 1=pos)
type:d 0 1

prop:
id:w0
```

```
name:initial weight of incoming connection 0
type:f 0.0 1.0

prop:
id:w1
name:initial weight of incoming connection 1
type:f 0.0 1.0

prop:
id:w2
name:initial weight of incoming connection 2
type:f 0.0 1.0

prop:
id:w3
name:initial weight of incoming connection 3
type:f 0.0 1.0

prop:
id:w4
name:initial weight of incoming connection 4
type:f 0.0 1.0

prop:
id:w5
name:initial weight of incoming connection 5
type:f 0.0 1.0

prop:
id:w6
name:initial weight of incoming connection 6
type:f 0.0 1.0

prop:
id:w7
name:initial weight of incoming connection 7
type:f 0.0 1.0

prop:
id:w8
name:initial weight of incoming connection 8
type:f 0.0 1.0

prop:
id:w9
name:initial weight of incoming connection 9
type:f 0.0 1.0

prop:
id:w10
name:initial weight of incoming connection 10
type:f 0.0 1.0

prop:
id:inWeights
name:incoming inWeights
type:o
flags:0

prop:
```

```
id:label
name:label
type:s
flags:0

prop:
id:updateInterval
name:weights updating interval
type:d
flags:0

prop:
id:myStateSum
name:sum of neuron states, used in updating weights
type:f
flags:0

prop:
id:inStatesSum
name:array of sum of incoming connection states, used in updating weights
type:o
flags:0

#END_REGION


```