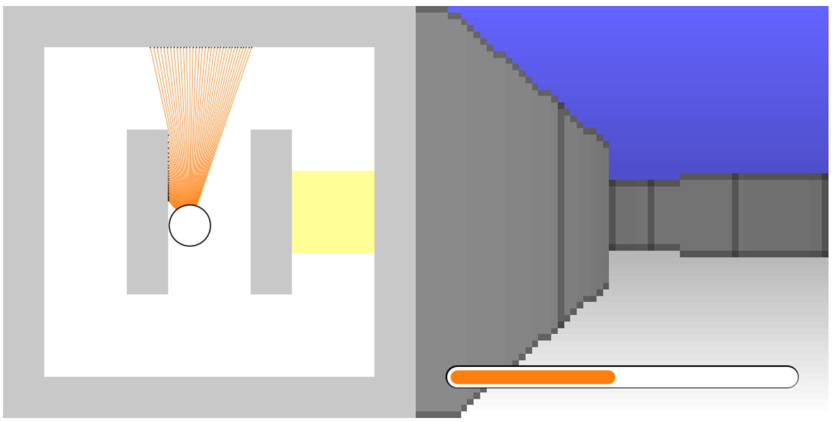
Contents

1 Programming with programmatoid and neuronoid

Let us consider the following digital experimental setup, as described in braincraft challenge presentation[1].

1.1 Simplified problem statement



Reference: The braincraft challenge[2].

Variables

Input θ_l Average of the *left* sensors input, between 0 and 1 when very close, 30 sensor field.

θ_r Average of the *right* sensors input, idem.

ε Energy indicator, between 1 when full and 0 when dead.

Output $d\theta$ Orientation relative increment, between -1 for -5 leftward and 1 for 5 rightward.

Internal λ Orientation preference 0 if left, 1 if right, $\lambda = 0$ at start.

With respect to the original we consider only two leftward and rightward sensor, and re-normalize the input/output values.

[1] `https://github.com/rougier/braincraft/blob/master/README.md#introduction`
[2] `https://github.com/rougier/braincraft`

Computation unit: neuronoid

By "neuronoid" we name the not very biologically plausible[3] simplest biological neuron or neuron small ensemble inspired by mean-field modelisation[4] of the Hodgkin–Huxley neuronal axon model[5].

$$\begin{array}{rr}\tau \frac{\partial v_i}{\partial t}(t) + v_i(t) = z_i(t), & z_i(t) \stackrel{\rm def}{=} h\left(\sum_j w_{ij}\, v_j(t) + w_i\right), \\ h(v) \stackrel{\rm def}{=} \frac{1}{1+\exp(-4\,v)}, & H(v) \stackrel{\rm def}{=} \left\{\begin{array}{lll} 1 & \mbox{if} & v > 0 \\ 0 & \mbox{if} & v < 0 \end{array}\right.\end{array}$$

where v_i is the membrane potential, so that:

$$\begin{array}{rcl} v(t) &=& 1/\tau \int_0^t z(s)\, exp(-(t-s)/\tau)\, ds + v(0)\, exp(-t/\tau) \\ &=& z(0) + (v(0) - z(0))\, e^{-t/\tau}\big|_{z(t)=z(0)}. \\ &=& z(t)|_{\tau=0} \\ &\simeq& (1-\gamma)\, v(t-\Delta t) + \gamma\, z(t-\Delta t)) \\ &=& \sum_{s=0}^{t-1} z(s)\, \gamma\, (1-\gamma)^{t-s+1}\, z(s) + v(0)\, (1-\gamma)^t \\ &=& z(0) + (v(0) - z(0))\, (1-\gamma)^t|_{z(t)=z(0)} \\ &=& z(t)|_{\gamma=1}. \end{array}$$

writing also the corresponding discrete approximation using an Euler schema with $0 < \gamma < 1, 0 < \tau$:

$$\begin{array}{c} \gamma \stackrel{\rm def}{=} 1 - \exp(-1/\tau) \Leftrightarrow \tau = 1/\log(1/(1-\gamma)) \\ \lim_{\gamma \rightarrow 0} \tau = +\infty, \lim_{\gamma \rightarrow 1} \tau = 0. \end{array}$$

Here, $h(\cdot)$ is the normalized sigmoid with $h(-\infty) = 0, h(0) = 1/2, h'(0) = 1, h(+\infty) = 1$, which is the mollification of the Heaviside function $H(\cdot)$, as detailed below.

It is thus a very common 1st order "neuronoid" model, but with an adjustable bias (or offset) w_i.

1.2 Programmatoid and neuronoid computation

Programmatoid computation

We name "programmatoid" computation the conception of an input-output straight-line program[6] implementing test operator on numerical value expressions using the Heaviside function, considering 1 as the true value and 0 as the false value. The $H(v)$ implements the test of the positive sign of v, while for $v_i \in \{0, 1\}$:

programmatoid conjunction The $v_0 = H(v_1 + v_2 + \cdots)$ formula performs a *or* operation.

[3] https://en.wikipedia.org/wiki/Biological_neuron_model#Relation_between_artificial_and_biological_neuron_models

[4] https://inria.hal.science/cel-01095603v1

[5] https://en.wikipedia.org/wiki/Hodgkin-Huxley_model

[6] https://en.wikipedia.org/wiki/Straight-line_program

programmatoid disjunction The $v_0 = v_1\, v_2 \cdots$ formula performs a *and* operation.

programmatoid negation The $v_0 = (1 - v_1)$ formula performs a negation.

so that we can combine any boolean expression on any test of value sign, thus value comparison, and value interval inclusion, switches between two expressions, etc. This defines real semi-algebraic[7] sets of degree 1.

Using local feedback we can also design several functions detailed in the Appendix of this section, while we also can combine neuronoid and programmatoid functions to design temporal functions.

Mollification of the Heaviside function

The Heaviside function is related to a conditional expression by a simple relation:

$$H(v) = \mathit{conditional\ value} > 0 \ ?\ 1 : \mathit{conditional\ value} < 0 \ ?\ 0 : H(0),$$

where $\mathit{condition}\ ?\ \mathit{value\ if\ true} : \mathit{value\ if\ false}$ is a conditional expression.

The Heaviside function approximates sigmoid with huge slope at zero, i.e.:

$$\forall v \neq 0, H(v) = \lim_{W_\infty \rightarrow +\infty} h(W_\infty\, v),\, h'(W_\infty\, v)|_{v=0} = W_\infty$$

while the convergence is also obtained for $v = 0$ in the distribution sense with $H(0) = h(0) = 1/2$. More precisely:

$$|H(\cdot) - h(\cdot)|_{\mathcal{L}_1} = O\left(\frac{1}{W_\infty}\right),$$

and, for $0 < \epsilon_\infty \ll 1 < v_\infty$:

$$h(W_\infty\, v_\infty) = 1 - \epsilon_\infty \Leftrightarrow v_\infty = \frac{-\log(\epsilon_\infty)}{4\, W_\infty} + O\left(\epsilon_\infty\right).$$

A step further, the local variation of sigmoid writes:

$$\begin{array}{rcl} h(v) & = & h(v_0) + \left[\frac{4}{\exp(v_0)^8} + O\left(\frac{1}{\exp(v_0)^{12}}\right)\right] (v - v_0) + O\left((v - v_0)^2\right) \\ & \simeq & h(v_0) + \frac{v - v_0}{\exp(v_0)^8/4}. \end{array}$$

Neuronoid implementation of programmatoid computation

We call "neuronoid" computation the conception of an input-output transform based on feed-forward and recurrent combination of neuronoids, as defined previously.

By successive combination, any programmatoid computation involving the $H(\cdot)$ function can be approximated by a neuronoid, with $\tau \simeq 0$.

A step ahead, we considering neuronoid with $\tau > 0$ it seems obvious that we can designed temporizing mechanisms, oscillators and sequence generator, sleep sort mechanism, etc. (not detailed here because not used at this stage, see appendix).

[7] `https://en.wikipedia.org/wiki/Real_algebraic_geometry`

A putative controller

Navigation

Heuristic : The bot runs at constant velocity, (i) ahead by default and (ii) if passing in the preferred orientation is possible, makes a quarter turn.

Programmatoid solution :

$$\begin{array}{rcl} d\theta & = & \underbrace{\gamma\,(\theta_l - \theta_r)}_{\substack{\text{linear correction to} \\ \text{maintain direction ahead}}} + \underbrace{\frac{\pi}{2}\,(\Delta\theta_r - \Delta\theta_l)}_{\text{direction change}} \\ \Delta\theta_r & = & \lambda\, H(\beta - \theta_r) \\ \Delta\theta_l & = & (1 - \lambda)\, H(\beta - \theta_l) \end{array}$$

where:

$\Delta\theta_*$ raises from 0 to 1 when the left sensor detects a passing, i.e., the fact tat the side wall is not close anymore.

γ is a feedback loop gain $0 < \gamma < 1$ to be adjusted high enough to correct the direction, small enough to avoid oscillations.

β is a threshold below which the side sensor input corresponds to no side wall but a passing.

in words: the bot navigates ahead thanks to the linear correction parameterized by γ and perform a quarter turn in the preferred direction as soon a passing is detected.

Neuronoid implementation : The mollification of this system uses 3 neuronoids and writes:

$$\begin{array}{rcl} d\theta & = & h\left(\gamma\,(\theta_l - \theta_r) + \frac{\pi}{2}\,(\Delta\theta_r - \Delta\theta_l)\right) \\ \Delta\theta_r & = & h(W_\infty\,(\beta - \theta_r) + 2\,W_\infty\,(\lambda - 1)) \\ \Delta\theta_l & = & h(W_\infty\,(\beta - \theta_l) - 2\,W_\infty\,\lambda) \end{array}$$

as easily verified considering the four cases $\beta \lesseqgtr \theta_*$ versus $\lambda \in \{0, 1\}$.

With respect to the programmatoid solution, the output value is "saturated" by the $h(\cdot)$ function while the $\Delta\theta_*$ almost binary values are approximated by the mollification of the Heaviside function. This part of the system is feed-forward thus without convergence or stability issue.

Direction choice

Heuristic : The initial direction is right, but as soon as an input contradicts this assumption, it is turned left once, and this remains.

Case 1 If the energy is to low, it means we turn in the wrong direction, so it changes.

Case 2 If there is a blue color on the left it means we must change from right to left.

Case 3 If there is a red (or yellow, etc) color somewhere, then change the turn direction if i see it again on the left.

Programmatoid solution :

$$
\begin{array}{rcl}
\lambda &=& \lambda + \Delta\lambda_1 + \Delta\lambda_2 + \Delta\lambda_3 + \cdots \\
\Delta\lambda_1 &=& (\lambda - 1)\, H(\alpha - \varepsilon) \\
\Delta\lambda_2 &=& (\lambda - 1)\, H(\iota - I_{\mbox{left blue color sensor}}) \\
\Delta\lambda_3 &=& (\lambda - 1)\, (\Upsilon_{\mbox{again red color}} + \Upsilon_{\mbox{again yellow color}} + \cdots) \\
\cdots && \\
\Upsilon_{\mbox{again this color}} &=& \Upsilon_{\mbox{seen this color}}\, H(\iota - I_{\mbox{left this color sensor}}) \\
\Upsilon_{\mbox{seen this color}} &=& \Upsilon_{\mbox{seen this color}} + (1 - \Upsilon_{\mbox{seen this color}})\, H(\iota - I_{\mbox{this color sensor}}) \\
\cdots &&
\end{array}
$$

where:

α is a the energy threshold, corresponding to the energy consumption during one turn.

ι is some color detection threshold.

$\Delta\lambda_1$ raises to one if the energy decreases below a threshold.

$\Delta\lambda_2$ raises to one if the blue color is seen on the left.

$\Delta\lambda_3$ raises to one if some color is seen again.

$\Upsilon_{\mbox{again this color}}$ raises to one a previously seen color is seen again on the left.

$\Upsilon_{\mbox{seen this color}}$ raises to one a color is seen for the first time.

$I_{\mbox{left this color sensor}}$ combines color sensor input.

$I_{\mbox{this color sensor}}$ combines left and right color sensor input.

Neuronoid implementation : The mollification uses 3 neuronoids for cases 1 and 2 plus 3 neuronoids by color for case 3. The derivation of the neuronoid equations is straightforward after the previous one.

Appendix: a few programmatoid and neuronoid components

Conditional expression

For two inputs $v_{i1}(t) \in \{0,1\}, v_{i0}(t) \in \{0,1\}$, an output $v_o(t) \in \{0,1\}$ and a control $v_l \in \{0,1\}$, the equation, for $W_\infty > 1$, and $0 < W_\epsilon \ll 1$:

$$
\begin{aligned}
v_o(t+1) &= v_l(t) == 1 \; ? \; v_{i1}(t) \; : \; v_{i0}(t) & (1)\\
&= v_l(t)\, v_{i1}(t) + (1 - v_l(t))\, v_{i0}(t) & (2)\\
&= H(v_{i1}(t) - W_\infty\, (1 - v_l(t)) - W_\epsilon) + H(v_{i0}(t) - W_\infty\, v_l(t) - W_\epsilon) & (3)\\
&\simeq h(W_\infty\, v_{i1}(t) - 2\, W_\infty\, (1 - v_l(t))) + h(W_\infty\, v_{i0}(t) - 2\, W_\infty\, v_l(t)) & (4).
\end{aligned}
$$

- It is obvious to verify that line (1) and (2) are equivalent.
- The fact line (2) and (3) are equivalent is verified by this truth table:

$v_l(t)$	$v_{i1}(t)$	$v_{i0}(t)$	$v_{i1}(t) - W_\infty\,(1 - v_l(t)) - W_\epsilon$	$v_{i0}(t) - W_\infty\, v_l(t) - W_\epsilon$	$v_o(t+1)$
1	0	0	$-W_\epsilon < 0$	$-W_\infty - W_\epsilon < 0$	$0 + 0 = 0 = v_{i1}(t)$
1	0	1	$-W_\epsilon < 0$	$1 - W_\infty - W_\epsilon < 0$	$0 + 0 = 0 = v_{i1}(t)$
1	1	0	$1 - W_\epsilon > 0$	$-W_\infty - W_\epsilon < 0$	$1 + 0 = 1 = v_{i1}(t)$
1	1	1	$1 - W_\epsilon > 0$	$1 - W_\infty - W_\epsilon < 0$	$1 + 0 = 1 = v_{i1}(t)$
0	0	0	$-W_\infty - W_\epsilon < 0$	$-W_\epsilon < 0$	$0 + 0 = 0 = v_{i0}(t)$
0	0	1	$-W_\infty - W_\epsilon < 0$	$1 - W_\epsilon > 0$	$0 + 1 = 1 = v_{i0}(t)$
0	1	0	$1 - W_\infty - W_\epsilon < 0$	$-W_\epsilon > 0$	$0 + 0 = 0 = v_{i0}(t)$
0	1	1	$1 - W_\infty - W_\epsilon < 0$	$1 - W_\epsilon > 0$	$0 + 1 = 1 = v_{i0}(t)$

- The approximation of line (3) at line (4) can be quantified as follows, for $v_{i1}(t) \in [0,1]$, $v_{i0}(t) \in [0,1]$, and $v_l(t) \in \{[0, \epsilon_\infty], [1 - \epsilon_\infty, 1]\}$, $1 \ll W_\infty$.

  - For $v_l(t) \geq 1 - \epsilon_\infty$:

  $$v_o(t+1) \leq h(W_\infty\, v_{i1}(t) - 2\, W_\infty \epsilon_\infty)) + h(W_\infty\, (2\, \epsilon_\infty - 1)),$$

  calcul pour $v_{i1}(t) > 1 - \epsilon_\infty$ et $v_{i1}(t) < \epsilon_\infty$

  - For $v_l(t) \leq \epsilon_\infty$:

  $$v_o(t+1) \geq h(W_\infty\, (2\, \epsilon_\infty - 1)) + h(W_\infty\, v_{i0}(t) - 2\, W_\infty\, \epsilon_\infty).$$

To explain the design choices, let us notice that:

- The W_∞ value is used at the programmatoid level to ensure that given the $v_l(t)$ binary value we will have have the desired value at 0. It allows to replace an expression including a product by a binary function, by a sum. The rationale is that there is no explicit multiplication between two variables at the neuronoid level, thus allowing one a straightforward neuronoid approximation.
- The W_ϵ value is used at the programmatoid level to avoid the ambiguous 0 value and ensure that for $v \simeq 0$ we obtain $H(v - W_\epsilon) = 0$. It is not useful at the neuronoid level, since the $h(\cdot)$ function is continuous.

RS input/output gate

For an input $v_i(t) \in \{0,1\}$, an output $v_o(t) \in \{0,1\}$, and a control $v_l \in \{0,1\}$, the equation:

$$\begin{aligned} v_o(t+1) &= v_l(t)\, v_i(t) + (1 - v_l(t))\, v_o(t) \\ &= v_l(t) == 1 \;?\; v_o(t) \,:\, v_i(t), \end{aligned}$$

implements a 1 bit memory, i.e., also called a RS gate.

From the previous

monostable binary value The $v_0 = v_0 + (1 - v_0)\, H(v_1)$ formula sets v_0 to 1 for ever, as soon v_1 has raised once to 1.

and by combination RS gates, bistable mechanisms, etc. (not detailed here because not used at this stage, see appendix).

To be done.

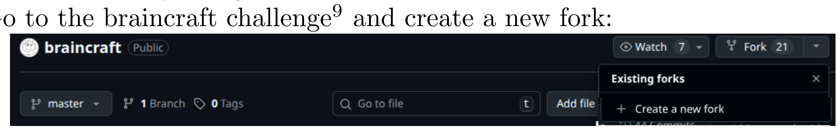
2 Using the braincraft challenge setup

Reference: The braincraft challenge[8].

You must be familiar with basic `git` usage and basic `python` programmation.

Installation of the setup

- Connect to `https://github.com` with your login.
- Go to the braincraft challenge[9] and create a new fork:



- Download the repository in SSH read/write mode:

-In the braincraft local git directory, run `make test`

\+ You may have to run `make install`, before.

\+ You are advised to use a virtual environment, running `make venv`

Running at the programmatic level

The `def next(input)` function that takes the `input = depth_l, color_l, depth_r, color_r, energy` date structure and output the `d_o` where a stands for the orientation is to be defined.

[8] `https://github.com/rougier/braincraft`

[9] `https://github.com/rougier/braincraft`

The `programmatic.py` code provides all functions to evaluate the implementation, without training phase.

Running at the artificial neural network level

The `def weights()` function returning the fixed pre-compiled parameters `weights = win, w, wout` is to be defined.

The `neuronoid.py` code provides all functions to evaluate the implementation, without training phase.