Robotics report

Freek Hens
s1033509

January 27, 2023

Contents

1 Agent implementation

1.1 Color dimensionality

I noticed that the red-green-blue (rgb) values of the colors were not high dimensional enough for Nengo to separate them in vector space. Therefore I chose to increase the dimensionality of the color representation by using the mapping function below. It substracts the differences of each of the individual r, b, and b, values, and adds each result as a new dimensionality. Since I have set the dictionary dimension D to 16, the remaining 10 dimensions need to be filled. I do this by filling them with the rgb values divided by ten, to make their influence small. This way, I end up with a 16 dimensional vector representing each of the five colors.

Afterwards the colors are added to the dictionary as shown. I also included the addition of the words YES and NO to the dictionary to show that they are in the same dictionary.

The color information is transported and mapped from the vision sensor to an ensemble via connections shown at the bottom of the code snippet. The same is done for the ahead color sensors and state.

```
D = 16

def mapping(x):
    '''
    Increasing the dimensionality of the color vectors
    to create more separated representations of the colors
    as vectors.
    Input x is assumed to be a list of length 3, representing rgb
    values.
    '''
    higher_dim = []
    higher_dim.append(x[0]-x[2])
    higher_dim.append(x[0]-x[1])
    higher_dim.append(x[1]-x[2])
    higher_dim.append(x[1]-x[0])
    higher_dim.append(x[2]-x[1])
    higher_dim.append(x[2]-x[0])

    for i in range(D-6):
        # Fill the other dimensions
        higher_dim.append(x[i%3]/10)

    return higher_dim

# Vocabulary
colors = spa.Vocabulary(D)
colors.add("GREEN", mapping(col_values[1]))
colors.add("RED", mapping(col_values[2]))
colors.add("BLUE", mapping(col_values[3]))
colors.add("MAGENTA", mapping(col_values[4]))
colors.add("YELLOW", mapping(col_values[5]))
colors.parse('YES + NO')

# Setup to convert color to a higher dimension
nengo.Connection(current_color, color_conversion)
nengo.Connection(color_conversion, model.vision.input, function=mapping)
```

Due to substracting the rgb values from eachother to create higher dimensions, colors with the same values for r, g, and b (such as grey values: white, grey, black) always have a value of 0 after mapping, or a low value due to noise. This makes them not distinguishable.

The mapping is efficient, since the operations are just simple subtractions of float numbers. A similar mechanism might be present in biological intelligence to increase contrast between colors and helping distinguish them. Humans even have three types of cones: red, green, and blue ones. Though, the subtraction of these color 'values' might not occur in the human brain for recognition of colors. I do think it is plausible that this happens for finding contrasts of colors, though, perhaps for e.g. edge detection.

1.2 Seen colors

To keep in memory what colors the agent has seen, I started by trying to use a feedback in a single memory state. The idea was to remember all the colors it has passed. This did not work for more than a few tenths of a second with more than three colors though. To solve this problem, I added five states, each representing a color (figure 1), and set them to YES when the agent has passed the corresponding color using spa.Actions (of course connected by the basal ganglia and thalamus). The states have a feedback of 1 to remember their state once they have been assigned YES. The feedback synapse for this remembering is set to a low value of 0.01 to quickly switch to the assigned value.

```
model.mag_seen = spa.State(D, vocab=colors, feedback=1,
    feedback_synapse=0.01)
model.gre_seen = spa.State(D, vocab=colors, feedback=1,
    feedback_synapse=0.01)
model.yel_seen = spa.State(D, vocab=colors, feedback=1,
    feedback_synapse=0.01)
model.red_seen = spa.State(D, vocab=colors, feedback=1,
    feedback_synapse=0.01)
model.blu_seen = spa.State(D, vocab=colors, feedback=1,
    feedback_synapse=0.01)

actions = spa.Actions(
    # Store that the agent has seen a color
    "dot(vision, MAGENTA) --> mag_seen=YES",
    "dot(vision, GREEN) --> gre_seen=YES",
    "dot(vision, YELLOW) --> yel_seen=YES",
    "dot(vision, RED) --> red_seen=YES",
    "dot(vision, BLUE) --> blu_seen=YES",
)

model.bg = spa.BasalGanglia(actions)
model.thalamus = spa.Thalamus(model.bg)
```

This way of representing which colors have been visited – or seen – is time efficient, since it is just one operation to switch to YES. It is linear in space complexity, since I use single state for each color that needs to be represented. I think this way of storing colors an agent has seen is biologically plausible to a certain degree and similarity of colors, i.e. I do not think there is a separate 'state' in naturally intelligent organisms for every shade of every color. I do think it is plausible that certain (groups) of colors (or concepts in general) are stored in a naturally intelligent system in a similar fashion as my implementation.

A limitation is of course that the model will become exceedingly large when more colors need to be discerned. But for this agent example this is no issue.

1.3 Color query

The first task of the agent was to answer a question about whether it has seen a certain color or not. This color query is implemented by using spa.Cortical to allow the user to 'ask' questions to the agent using a query state. The representations of the colors and whether they have been seen

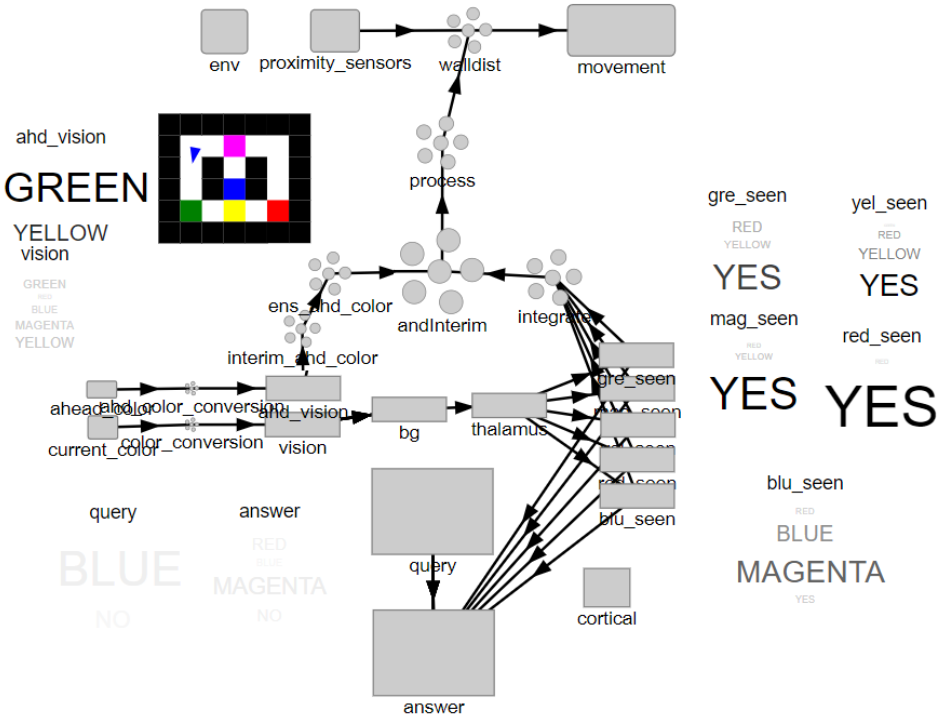

Figure 1: Illustration of how I represented which colors the agent has already seen. In this example, the agent is running along the outside wall, so it has seen green, yellow, red, and magenta. This is correctly represented by the corresponding states saying YES, and the blue state not having assigned any clear value.

are stored in the agent's cortical as shown in the code snippet. The query works by combining the value of the state of the color with the name of the color and the inverse of the query using the * operator. For example, if the query is MAGENTA, and the state of mag_seen is YES, then the answer should be YES*MAGENTA*~ MAGENTA which equals YES. In other words, we asked the agent: is the color MAGENTA seen? (By setting the query state manually to MAGENTA) And it answered with YES. Figure 2 shows this for color green, which it has visited, and figure 3 shows this for color blue, which it has not visited.

```
# For color query
model.answer = spa.State(D, vocab=colors)
model.query = spa.State(D, vocab=colors)

query = spa.Actions(
    "answer = mag_seen * MAGENTA * ~query",
    "answer = yel_seen * YELLOW * ~query",
    "answer = gre_seen * GREEN * ~query",
    "answer = red_seen * RED * ~query",
    "answer = blu_seen * BLUE * ~query",
)

model.cortical = spa.Cortical(query)
```

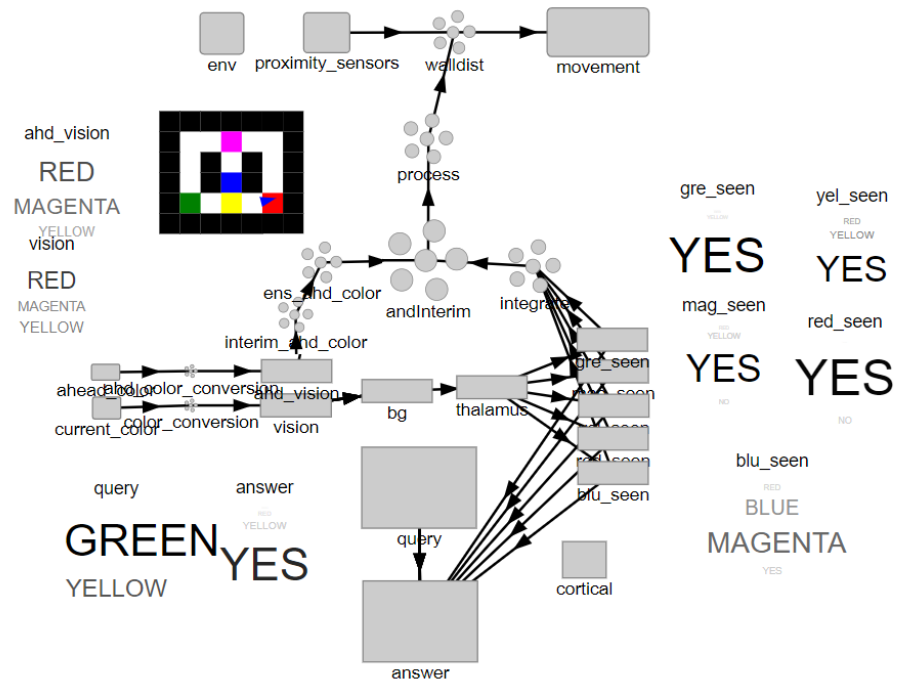

Figure 2: Setting the query state to GREEN, the model answers YES in the answer state, indicating that the agent has seen green. On the right the agents' representation of which colors have been seen as described in 1.2.

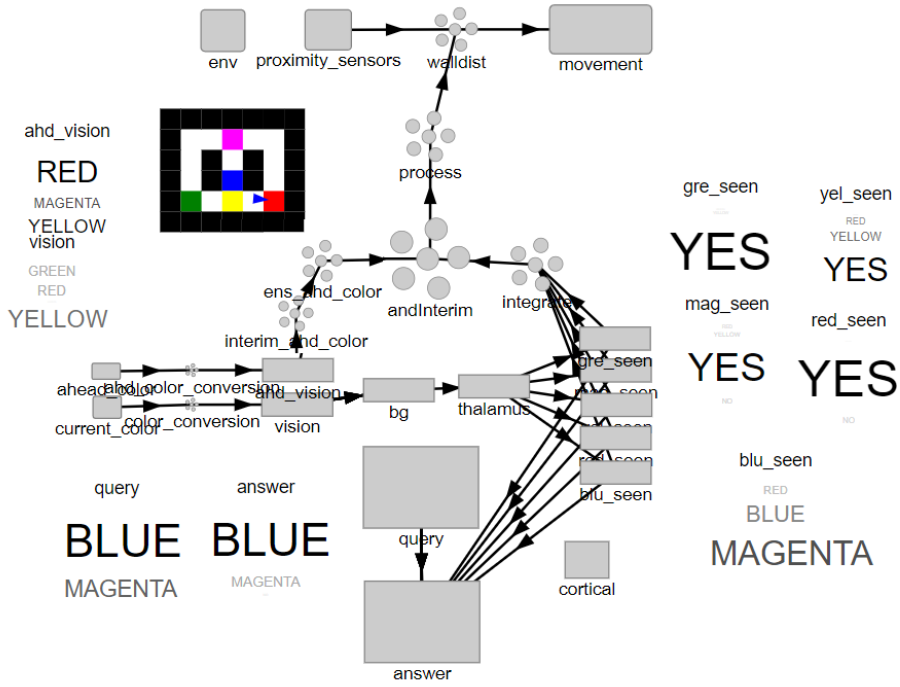

Figure 3: Setting the query state to BLUE, the model answers with BLUE in the answer state. This is not ideal as the agent should answer with NO. This situation is further described in section 1.3.

Due to the lack of a NO as a semantic pointer for e.g. the `blu_seen` state – visible by the greyed out semantic pointers in figures 1, 2 and 3 – the agent has no clear representation for not having seen a color (example in figure 3). Due to this lack of NO, when asking the agent for a color which it has not seen, the answer will be nonsensical. But given that YES and NO in the context of having seen a color are mutually exclusive, this limitation of the implementation could be fixed by processing that the absence of YES means NO. However, unfortunately I did not have time to implement this.

I am not sure if the query system using binding and unbinding provided by the Nengo tutorials is biologically plausible, but I assume it is, since Nengo also limits certain design choice in favor of biological plausibility (e.g. basal ganglia rules or passthrough neurons, which I encountered during the project).

1.4 Changing movement

The movement of the agent needs to change based on what colors it has seen, since it may not cross a color that it has seen already. The agent came with a simple wall avoidance behaviour, which relies on sensors directly in front, and slightly to the left and right of the agent. These three values are transformed into two values, 'spd' and 'trn' which are fed into the `movement` ensemble, which controls the agent. I augmented the agents' behaviour by essentially switching the behaviour of the agent between wall avoidance and obstacle avoidance. The wall avoidance, the behaviour the agent came with, can be seen exploration. When the obstacle avoidance is switch on, the agent will stop where it is and turn around. Of course the idea being to avoid the color ahead of the agent, if it is not supposed to cross it. To make the decision of whether to avoid or not is based on the information from the 'color seen' (`mag_seen`, etc.) states and the `ahead_color` state (figure 4). An ensemble is created which integrates the information from the 'color seen' states, called 'integrate'. It stores a 5 dimensional vector containing zeros and ones corresponding to a color having been seen or not (1 means seen), where the order is: [magenta, green, yellow, red, blue]. Similarly, an ensemble called `ens_ahd_color` stores which color is most prominent in a 5 dimensional vector according to the `ahead_color` sensor. These values of these vectors are passed to the `process` ensemble, through an AND gate which stores the values in a 5 dimensional array again. Then a function is applied to check if any of these 5 values is higher than some threshold, because then the switch of the agent is flipped to obstacle avoidance mode. If they are all below this threshold, the switch is flipped back to exploration mode. This mechanism ensures that the agent does not cross a color which it has already seen.

The implementation is shown in the following code snippet. I use the information from the mapped `ahead_color` sensors, as said in 1.2, to discern which color the agent sees ahead of it. Instead of storing the colors as a 16 dimensional vector, I pass it through the `iscolor` function. This function returns a five dimensional vector with a binary encoding of the colors: one of the five values in the vector is 1, representing a certain color. At the same time, the information from the 'color seen' states, also described in 1.2, is integrated to the `integrate` ensemble. This connection performs a similar dimensionality reduction by instead of storing a 16 dimensional vector for YES or NO, `integrate` also stores a five dimensional vector, where a value is 1 when the corresponding 'color seen' state stored a YES and 0 otherwise. This is done by the `isyes` function. Now having these two five dimensional vectors containing information of the color ahead of the agent, and which colors it has seen, the AND of these is computed via the `andInterim` ensemble in the `process` ensemble (snippet lines 88 – 89). After this, agent's behaviour is changed by the `motor` ensemble, which changes the movement of the agent.

```
1 # Error https://github.com/nengo/nengo/issues/805
2 # to prevent biologically implausible situations with passthrough nodes
3 # I should add passthough nodes in between, but not enough time. (see integrate
  ↪  ensemble.)
4 model.mag_seen.output.output = lambda t,x:x
5 model.gre_seen.output.output = lambda t,x:x
```

```
model.yel_seen.output.output = lambda t,x:x
model.red_seen.output.output = lambda t,x:x
model.blu_seen.output.output = lambda t,x:x

# --------------------------------
# Processing information for avoiding
# --------------------------------

def isyes(x):
    '''returns 1 if the vector represents YES'''
    t = 0.8
    if np.dot(x, colors["YES"].v) > t:
        return 1
    else: return 0

def iscolor(x):
    '''returns a vector representing one of the colors'''
    # threshold
    t = 0.65
    # keep track which color has the highest dot product
    maxi = 0
    # and the respective representation
    cols = [0,0,0,0,0]

    magsim = np.dot(x, colors["MAGENTA"].v)
    gresim = np.dot(x, colors["GREEN"].v)
    yelsim = np.dot(x, colors["YELLOW"].v)
    redsim = np.dot(x, colors["RED"].v)
    blusim = np.dot(x, colors["BLUE"].v)

    val = 5

    # if similarity passes threshold, and is the new higest,
    # then store new max value and update the return vector.
    if magsim > t and magsim > maxi:
        maxi = magsim
        cols = [val, 0, 0, 0, 0]
    if gresim > t and gresim > maxi:
        maxi = gresim
        cols = [0, val, 0, 0, 0]
    if yelsim > t and yelsim > maxi:
        maxi = yelsim
        cols = [0, 0, val, 0, 0]
    if redsim > t and redsim > maxi:
        maxi = redsim
        cols = [0, 0, 0, val, 0]
    if blusim > t and blusim > maxi:
        maxi = blusim
        cols = [0, 0, 0, 0, val]
    return cols

# readout of mapped ahead_color and interim ensemble to make ens_ahd_color
# not a passthrough node, to enable functions on the connection.
interim_ahd_color = nengo.Ensemble(n_neurons=400, dimensions=D)
ens_ahd_color = nengo.Ensemble(n_neurons=400, dimensions=5, radius=1)
# ens_ahd_color has five neurons which are 1 when a color is ahead.
nengo.Connection(model.ahd_vision.output, interim_ahd_color)
```

```
nengo.Connection(interim_ahd_color, ens_ahd_color, function=iscolor)

# integrate ensemble to connect ahead color and current color information.
# Has 5 neurons which are 1 when a color has been seen respectively.
integrate = nengo.Ensemble(n_neurons=400, dimensions=5, radius=2)

# connecting to integrate
nengo.Connection(model.mag_seen.output, integrate[0], function=isyes)
nengo.Connection(model.gre_seen.output, integrate[1], function=isyes)
nengo.Connection(model.yel_seen.output, integrate[2], function=isyes)
nengo.Connection(model.red_seen.output, integrate[3], function=isyes)
nengo.Connection(model.blu_seen.output, integrate[4], function=isyes)

# andgate ensemble to store 5 AND values
andInterim = nengo.Ensemble(n_neurons=400, dimensions=5, radius=3)
nengo.Connection(integrate, andInterim, transform=0.5)
nengo.Connection(ens_ahd_color, andInterim, transform=0.5)

def threshold(x):
    # and gate threshold
    return x > 0.7

# process ensemble
# has 5 values which are 1 when that color has been seen and is ahead ->
# should avoid.
# e.g. mag_seen AND ahd_color = magenta -> neuron[0] = 1
process = nengo.Ensemble(n_neurons=500, dimensions=5, radius=1)
nengo.Connection(andInterim, process, function=threshold)

def avoid(x):
    ''' Decide whether to avoid the upcoming color'''
    # input x is 5D vector of values. If one is about 1, the ahead square
    # should be avoided.
    avoid = x > 0.65
    if avoid.any():
        # avoid
        return 1
    # else, dont avoid
    else:
        return 0

def movement_func(x):
    ''' original movement function'''
    turn = x[2] - x[0]
    spd = x[1] - 0.5
    return spd, turn

def act(x):
    '''input is 3D: [spd, trn, avoid?]
        output is 2D: [spd, trn] for movement ensemble
    '''
    if x[2] > 0.7:
        #avoid: stop and turn
        return [-1, -5]
    else:
        # just do object avoidance
        return x[0:2]
```

```
120
121 # ensemble to control movement: [spd, trn, avoid?]
122 motor = nengo.Ensemble(n_neurons=500, dimensions=3)
123
124 # walldist sensors connect to first 2 dims for wall avoidance control
125 # last dim is used for binary avoid? no/yes: 0 or 1
126 nengo.Connection(walldist, motor[0:2], function=movement_func)
127 nengo.Connection(process, motor[2], function=avoid)
128 # act function controls the changing of action which passes on
129 # a 2 dim vector of form: [spd, trn] again.
130 nengo.Connection(motor, movement, function=act)
```

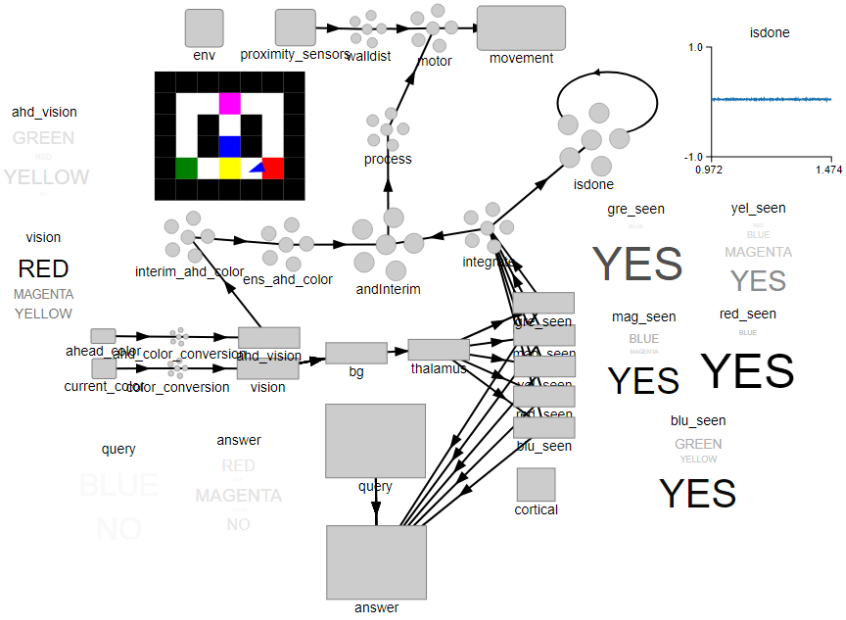

Figure 4: The final model of the agent, which improves control of the agent and adds obstacle avoidance. Also it shows whether the agent has completed the task in the `isdone` ensemble.

Nengo has explicitly limited the ability to preform functions on the connections of passthrough nodes. They have done this to disallow the creation certain biologically implausible implementations: "*Unlike other types of nodes, we explicitly disable the function argument when connecting from passthrough nodes. The reason for this is to ensure that users know they are making a direct connection and not a decoded connection*" [1]. A way around this limitation is to apply the identity function to the output of the passthrough nodes. This is done at lines 4 – 8 of the code snippet for the 'color states'. I circumvented this issue of passthrough nodes for the 'ahead vision' state by adding an interim ensemble `interim_ahd_color` which acts as a passthrough to the `ens_ahd_color` ensemble. I did not apply similar interim passthrough nodes to the 'color seen' states due to time restrictions.

Due to not implementing passthrough nodes between the 'color seen' states and the integrate ensemble (but instead performing a function on the connection between the two), I have made my implementation less biologically plausible, since the output from states should just be a readout and be passed to an ensemble or node where the computation takes place. Aside from that, I think the implementation I chose is not biologically implausible, as having representations of colors that

lie ahead together with representations of colors that one has seen before is not an uncommon mechanism for processing information [6]. Next to that, performing an AND operation in one way or another also does not seem implausible to me.

1.5 Color sequence

The second task of the agent was to visit a sequence of colors in order. Unfortunately, due to time restrictions, I only implemented it such that the agent checks whether it has visited all of the colors in the given sequence.

The color sequence is given to the agent by changing the values of the list `color_sequence` in the code. The ensemble `isdone` checks which colors the agent has seen and crosses them of the list. When all the items of the list are crossed off, the value of `isdone` changes from 0 to 1, with a feedback to remember it. This can be seen in the graph on the top-right of figure 4, which shows the final version of my implementation. The code snippet below shows the implementation.

```
# ---------------------------------
# Input color sequence to follow
# ---------------------------------

# Input sequence of colors to search
color_sequence = [GREEN, RED, BLUE]

def check(x):
    '''input is 5D array of colors seen
    Function checks whether the color_sequence has been achieved.'''
    t = 0.8 # Threshold
    length = len(color_sequence) + 1
    for c in color_sequence:
        # color c is a D dimensional color vector
        # iscolor makes it the same 5d representation as x
        cols = iscolor(c)
        if np.dot(cols, x) > t:
            length -= 1
    if length == 0:
        # all asked colors have been seen
        return 1
    else: return 0

isdone = nengo.Ensemble(500, 1)
nengo.Connection(isdone, isdone, transform=1, synapse=0.01)
nengo.Connection(integrate, isdone, function=check)
```

1.6 Other environments

On the given environment, the agent performed the task decently. The color recognition and query work (irrespective of the map), and the I see change in movement when the agent has to avoid a color. It does not always succeed in avoiding it though, since when it turns, the switch flips back to normal wall avoidance mode, and then it can happen that the agent just runs over to-be-avoided color anyway.

I tried a different environment in which all colors lay on one line and some colors appear twice 5. It is visible that during the wall avoiding behaviour of the agent, in which the speed relies on the distance to the walls, the agent speeds up very much. So much that the color avoidance system cannot stop the agent before it runs over the color.

Figure 5: An alternative map on which the agent was tested.

1.7 Conclusion

Admittedly, the movement of the agent is quite unstable. This could be improved greatly by adjusting the threshold values and weights for the decision making processes and the movement ensembles, though I did not have enough time for this. Also, the second task is not implemented correctly. I do think the core logic of the agent is good, though, and I am proud to see aspects of the tasks back in the behaviour of the agent.

2 Reflection

Reflection on the possibilities and limitations within cognitive robotics today in context of this project.

- To what degree are simple agents like we have built here useful for cognitive models (think about minimal cognition approaches)

  The (simple) models that are made are useful for testing theories (e.g. NEF, spiking networks) as a proof of concept and seeing emergent behaviour. I think the Nengo environment using the GUI, while impressive, is not well suited for developing large scale models. Also, the model I created was starting to run into performance issues due to the amount of neurons and computations that were happening. The simple networks like the one created here can also be used as building blocks for more complex models.

- What would be concrete implementational issues with theories we have seen throughout the course. Be concrete with examples.

  Developmental robotics has a practical issue with implementing: the theory is based on that human children are not born with the abilities of adult humans [2]. Instead, they need to learn tasks and develop strategies over many years. Developmental robotics is based on this idea that maybe that is the same for artificial systems. But to test and implement this, many years of (hard- and software) development and learning is needed from the artificial system [3]. The development is also very intricate and involves interactions between e.g. genetics, environment and experience. This can be very hard to model.

- What features should a cognitive architecture contain? Should we favour emergent or cognitivist approaches? What are the advantages and disadvantages of each?

  A cognitive architecture should be modular, i.e. subsystems should be divided by function and be specialized for different tasks (e.g. my implementation having a separate vision processing area and motor control area). It should also interface with the environment via sensors and actuators to allow it to operate in the with the (real) world, as proclaimed by Embodied Cognition [4]. Even though the top-down approach of the cognitivist architectures can provide a deeper understanding of the processes, I would say that emergent behaviour is more favorable, since they are more biologically plausible, robust and flexible than cognitivist approaches. However, they may lead to the behaviour being less understandable due to the black boxes.

- What might fundamentally have been different if the task in this assignment had used Dynamic Field Theory (DFT) instead?

  Though both NEF, SPA and DFT have similarities, like both processing information using (groups of) neurons, there are also differences:

  DFT assumes that cognitive processes are distributed across multiple regions of the brain rather than localized in specific areas [5]. This means that different aspects of a task or perception are represented by activity patterns in different populations of neurons, whereas in my implementation, I chose to separate e.g. the processing of visual information and the motor actions.

  Also, semantic pointers (or other representations of concepts) would have instead been modelled by different activations in the neurons, instead of vectors. I imagine that then the concept of similarity between internal representations would become more difficult to model.

  The concepts of self-organization and attractor dynamics are not apparent in the NEF and SPA architectures, while they are fundamental to DFT [5]. This means that the behaviour of the agent would be emergent from the connections between populations of neurons, instead of dedicated, assigned, areas of computation (e.g. vision processing).

References

[1] Nengo. *Connections in depth.* Accessed 24/01/2023. https://www.nengo.ai/nengo/connections.html#passthrough-nodes

[2] Lungarella, M., Metta, G., Pfeifer, R., & Sandini, G. (2003). Developmental robotics: a survey. Connection science, 15(4), 151-190.

[3] Asada, M., Hosoda, K., Kuniyoshi, Y., Ishiguro, H., Inui, T., Yoshikawa, Y., ... & Yoshida, C. (2009). Cognitive developmental robotics: A survey. IEEE transactions on autonomous mental development, 1(1), 12-34.

[4] Foglia, L., & Wilson, R. A. (2013). Embodied cognition. Wiley Interdisciplinary Reviews: Cognitive Science, 4(3), 319-325.

[5] Erlhagen, W., & Schöner, G. (2002). Dynamic field theory of movement preparation. Psychological review, 109(3), 545.

[6] Bracci, S., Caramazza, A., & Peelen, M. V. (2015). Representational similarity of body parts in human occipitotemporal cortex. Journal of Neuroscience, 35(38), 12977-12985.