# BECCA version 0.4.0
# User's Manual

Brandon Rohrer

June 5, 2012

# Contents

# Chapter 1

# Get and run BECCA

Each chapter in this guide is designed to help you do something specific with BECCA. This chapter helps you to get a copy of BECCA on your local machine and run it on some generic worlds.

## 1.1 Where do I get the code?

You can download BECCA from

`www.openbecca.org`

or

`www.sandia.gov/rohrer`

Or you can get the latest bleeding edge version (for which any of this documentation may already be outdated) from the github repository at

`https://github.com/matt2000/becca`.

## 1.2 What tools do I need to run it?

BECCA is intended to be runnable on any hardware platform. This version relies on and was developed on

- Python 2.7
- NumPy 1.6.1[1]
- matplotlib 1.2.

BECCA has been run several different platforms (Mac OS 10.6.8, Ubuntu 12.04,[2] 32-bit Windows Vista, and 32-bit Windows 7) and IDEs/interpreters (Eclipse, PyScripter, IDLE, Stani's Python Editor, emacs, command line)[3].

## 1.3 How do I run it?

Run `benchmark.py` in your Python interpreter. The `benchmark.py` module automatically runs BECCA on a collections of worlds that are included with the download. Run it in a Python interpreter to get a report of BECCA's performance in the worlds. `benchmark.py` can be used both to compare BECCA's speed on different computing platforms and, more importantly, to compare different variants of BECCA against each other.

The worlds in `benchmark.py` are intended to be simple, but to test BECCA's fundamental learning capabilities. BECCA is an agent, in the sense that it makes

[1]The code makes use of at least one NumPy call, `count_nonzero()`, that is not supported in NumPy 1.5.x.

[2]From developer SeH: BECCA's dependency on NumPy 1.6 is provided by the latest Ubuntu 12.04 package (but not Ubuntu 11.10 which provides NumPy 1.5). A note on the website indicating this might be helpful to some users. Matplotlib is also provided, so everything seems to work fine on Ubuntu 12.04.

[3]Any notes on successes or incompatibilities would be very welcome at `openbecca.org`.

decisions in order to achieve a goal, but it is intended for use in many different settings, each of which is referred to as a world. The worlds tested in `benchmark.py` include one and two dimensional grid worlds and one and two dimensional visual worlds. The reward provided by each world gives motivation to BECCA to behave in certain ways. When it behaves correctly, it maximizes its reward. This is BECCA's one and only goal. Each world in the benchmark provides periodic updates an final reports of its progress.

Nice job. Now for the fun part.

# Chapter 2

# Write and run your first world

This chapter steps you through the process of writing and running your first world and in the process, explains BECCA's structure at a high level.

## 2.1 What is a world?

A BECCA *agent* is a thing that chooses actions. In order for those actions to have any effect, they must be coupled to a *world*, an external environment that the agent can interact with. The agent–world configuration that BECCA uses is shown in Figure 2.1. It is the canonical statement of the reinforcement learning problem: a reinforcement learning agent tries to choose actions so as to maximize the reward it receives. [3]

Due to the modularity of this architecture, you can develop and run your own worlds without having to know very much about how the BECCA agent works. The only constraints BECCA imposes on worlds are:

- A world must read in a fixed number of actions at each time step. This number is chosen by and defined in the world.

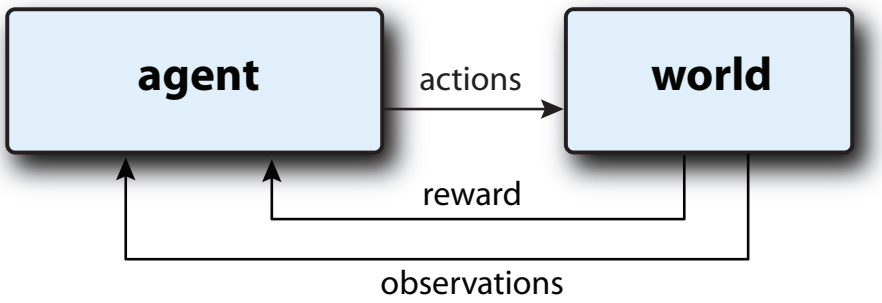


Figure 2.1: The agent-world coupling in BECCA. The agent selects actions to execute on the world, which in turn provides observations and reward feedback.

- A world must provide a fixed number of sensory observations at each time step. This number is also chosen by and defined in the world.
- A world must provide a scalar reward value at each time step.
- All actions and observations are real valued, equal to 0, 1, or something in between.
- The reward signal is real valued, equal to -1, 1, or something in between.

Strictly speaking, BECCA allows for two classes of observations, *sensors* and *primitives*. The only difference between the two is that the agent has to build sensors into features before it can use them, whereas it can use primitives as features immediately. Typically, observations that are likely to need some refining before it becomes useful (such as sets of pixel values) are passed in as sensors, and information that is more immediately useful (such as a contact sensor) are passed in as primitives, although BECCA puts no such constraints on them.

## 2.2 How do I make a *hello* world?

The quickest and dirtiest way to get started making your own world is to copy an existing one and modify it. This is especially effective if there already exists

something roughly similar to what you want. That's the approach we'll use to make a hello world.

1. Save `worlds/grid_1D.py` as `worlds/hello.py`
2. Replace the body of `step()` with this:

   ```
   self.timestep += 1
   print self.timestep, ' hellos to the world!'

   sensors = np.zeros(self.num_sensors)
   primitives = np.zeros(self.num_primitives)
   reward = 0
   return sensors, primitives, reward
   ```

   and save the changes.
3. Add the line

   ```
   from worlds.hello import World
   ```

   to `tester.py`. Make sure all the other lines beginning with `from worlds`... are commented out.
4. Run `tester.py`

## 2.3 What do I need to implement in my world?

`hello.py` runs because it meets a few basic requirements. This section lists them and describes the mechanics of a basic BECCA world.

Any world needs to have at least these three publicly accessible member variables,

- `num_sensors`

- `num_primitives`
- `num_actions`

these three methods,

- `step()`
- `is_alive()`
- `set_agent_parameters()`

and perhaps two optional methods,

- `is_time_to_display()`
- `vizualize_feature_set()`

each of which are all described in more detail below.

### 2.3.1 `num_sensors`

Specifies the number elements in the array of sensor information that is passed to the agent at each time step.

### 2.3.2 `num_primitives`

Specifies the number elements in the array of feature primitives that is passed to the agent at each time step.

### 2.3.3 num_actions

Specifies the number elements in the array of actions that it expects to receive from the agent at each time step. Together, these three variables define the interface between the agent and the world. They may be different for each world, but they must remain constant within a single world. [1]

### 2.3.4 step()

Advances the world by one time step. This is single most important method in a world. It specifies the behavior of the world. It accepts an array of actions and returns an array of sensors, an array of primitives, and a reward value. BECCA places no constraints on the complexity or the execution time of the `step()` method. It can incrementally advance a simulation, pass and receive network packets, or provide an interface to physical robot. The agent will wait until `step()` finishes and returns its observation and reward information before advancing to the next time step.

### 2.3.5 is_alive()

Informs the executing loop when to stop. The top level execution loop (as implemented in `benchmark.py` and `tester.py` continues to run BECCA through its agent-world-agent cycle until `is_alive()` returns false.

### 2.3.6 set_agent_parameters()

Sets the the agent's internal parameters to values specific to the world. BECCA has a quite a few constants that affect its operation, in the neoghborhood of 20 at

[1] In this version of BECCA, each of these variables must be at least 1. BECCA doesn't yet know how to handle empty sensory or primitive arrays. Worlds that have no sensors or primitives can pass a single zero value at every time step to achieve this.

last count. Most of these are set to values that seem to work for all worlds in the benchmark battery. But for development and troubleshooting, it has proven useful to be able to adjust some of BECCA's parameters manually. It is my hope that, in time, a single set of parameters will prove generally useful across a broad set of tasks. In the meantime this method provides a mechanism for world-specific knob-twiddling.

### 2.3.7 `is_time_to_display()`

Informs the executing loop when to display the feature set by returning `True`. This is for visualization purposes only and doesn't help BECCA learn in any way. This method is optional. If it doesn't exist, the execution loop just moves on.

### 2.3.8 `vizualize_feature_set()`

Displays the feature set when `is_time_to_display()` returns `True`. The feature set is expressed in terms of sensors, primitives, and actions by the agent. This method takes those feature representations and interprets them for the user in terms of what it knows about how those signals. For instance, if the sensors correspond to pixel values from a camera, this method would render the sensor component of each feature as an image. The agent has no information about where its sensor and primitive arrays came from or about what its actions do in the world. This method let's the world give a world-specific interpretation of those values for the user.

Worlds of course may have any number of other member variables and methods for internal use. Others than have proven useful in the benchmark battery worlds include `calculate_reward()` and `display()`.

## 2.4 What is `base_world.py`?

“Wait a second," you say. “My `hello.py` didn’t have an `is_alive()`, but you said it needed one. What’s up with that?"

These two lines from the beginning of the module help to solve that mystery:

```
from .base_world import World as BaseWorld
class World(BaseWorld):
```

`hello.py` is actually a subclass of `base_world.py`,[2] which defines a generic `is_alive()`. It also defines a generic `set_agent_parameters()` and `step()`, as well as default values for `num_sensors`, `num_primitives`, and `num_actions`. When you subclass it to create a new world you, only need to override those methods and variables that you want to change.

## 2.5 How do I run my world?

`tester.py` is the vehicle for coupling a new world with a BECCA agent and running them. You’ve already added your hello world to `tester.py` and run it. This section gives a line-by-line overview of the rest of the module and what it does. This will help make clear a few of the finer points of running a world.

1. Import the relevant modules. In particular import a `World` class from a module containing one.

   ```
   import numpy as np
   from agent.agent import Agent
   from agent import viz_utils
   from worlds.hello import World
   ```

[2] If terms like *class*, *subclass*, *inheritance*, and *override* aren’t familiar to you in this context, I’d highly recommend doing some quick reading on object-oriented programming. The concepts aren’t that tricky, but they are extremely useful when discussing OO software, such as BECCA.

2. Define the `main` method and initialize some objects. When initializing the `agent`, there are two optional variables, `MAX_NUM_FEATURES` and `agent_name`. `MAX_NUM_FEATURES` provides an upper limit on the number of features that the agent can create and can be set appropriately to prevent BECCA from using up all your RAM. `agent_name` is a text string used to identify a specific agent and is useful if you are planning to save the agent or restore it from a previously saved agent. Note that if you do this, the world used in both cases must have the same number of sensors, primitives, and actions.

```
def main():

    world = World()

    agent_name = "test";
    MAX_NUM_FEATURES = 3000
    agent = Agent(world.num_sensors,
                  world.num_primitives,
                  world.num_actions,
                  MAX_NUM_FEATURES,
                  agent_name)
```

3. Restore the agent to a previously saved agent, if desired.

```
    #agent = agent.restore()
```

4. Modify the agent's parameters according to the requirements of this particular world. Initialize actions such that the first set of commands to the world is always zeros.

```
    world.set_agent_parameters(agent)
    actions = np.zeros(world.num_actions)
```

5. Begin the execution loop. Couple the output of the world into the input of the agent and vice versa.

```
    while(world.is_alive()):
        sensors, primitives, reward = world.step(actions)
        actions = agent.step(sensors, primitives, reward)
```

6. If the necessary methods exist, and if they dictate, display the feature set to the user.

```
        try:
            if world.is_time_to_display():
                world.vizualize_feature_set(
                    viz_utils.reduce_feature_set(
                    agent.grouper),
                    save_eps=True)
                viz_utils.force_redraw()
        except AttributeError:
            pass
```

7. After the world has completed its lifetime, give a final report of the agent's performance.

```
        agent.report_performance()
        agent.show_reward_history()

        return
```

8. And finally, run `main()`

```
if __name__ == '__main__':
    main()
```

# Chapter 3

# Share your world with other BECCA users

BECCA was created so that it would be easy to use and share what you create with others. This section describes how to get your world out and get the word out.

## 3.1 Where do I put my world module so that others can find it?

The most convenient place to share BECCA content is GitHub. Open a GitHub account if you don't have one already, create a repository, and populate it with your world's code.

The official BECCA's core code base is hosted on GitHub in one of Matt Chapman's repositories:

`https://github.com/matt2000/becca`

It contains a tagged version of this code, plus all the incremental commits that are working toward the next version. The core contains the full agent, plus

a battery of worlds for benchmarking, and a couple of utility modules. The plan is to keep the BECCA core small and trim with only the necessaries for getting a new user up and running.

Contributed code and modifications will be available separately. The details for how this will happen are still taking shape as our community is still young and small, but for now contributed modules will be listed with brief descriptions on the `openbecca.org` site. This list will be as exhaustive as possible, until the volume of contributions forces us to find another way to do things.

## 3.2 How do I tell them about it?

If you didn't record it, it didn't happen. When you do something new, make a record of it in some way so you can show everybody how cool it is. There are lots of good ways:

- Make a video
- Grab screenshots
- Draw diagrams
- Write an appendix to this users guide[1]
- Write a conference paper[2]
- Write a paragraph.

Then, whatever form your documentation takes, share it around. Right now, the best way to broadcast notifications to other BECCA users is to post in the Google Group,

[1] If you're new to LaTeX, just copy one of the users guide chapter files and modify it to tell your story. The `.tex` files are in the BECCA GitHub repository under `doc/user_manual/`.

[2] I've had good luck with the Artificial General Intelligence (AGI) and Biologically Inspired Cognitive Architectures (BICA) conference series. Also the combined International Conference on Development and Learning/Epigenetic Robotics Conference (ICDL/EpiRob) is good and the brand new AAAI Spring Symposium on Integrating Artificial Intelligence looks to be fantastic.

`https://groups.google.com/forum/?fromgroups#!forum/becca_users`

Sign up if you haven't yet. Incidentally, subscribing to the group's posts is also the best way to hear about others' contributions to the body of BECCA code.

This is subject to change. The preferred way to advertise new content may eventually migrate to an `openbecca.org` forum.

# Chapter 4

# Modify your agent code

This section is to help you get started hacking your own version of the Becca agent. It's still very new, and there's a lot of room for improvement, so don't be afraid to jump in and start rewiring it.

## 4.1 How is the agent code structured?

This section won't give the justification and theory behind every part of the agent, but it will give an overview of the structure so that you have some idea about where to start making changes. A more thorough description definitely belongs in this guide, and I plan to include it in later versions. For now, the best I can do is refer you to some previously published descriptions: [2, 1]. These are no longer entirely accurate, unfortunately, but give a pretty good ideal about what is going on.

Figure 4.1 gives an overview of the major classes are some of their important members. In the code, each class is contained in a module with a lowercase version of the same name. A brief description of the major functions of each class is given below.

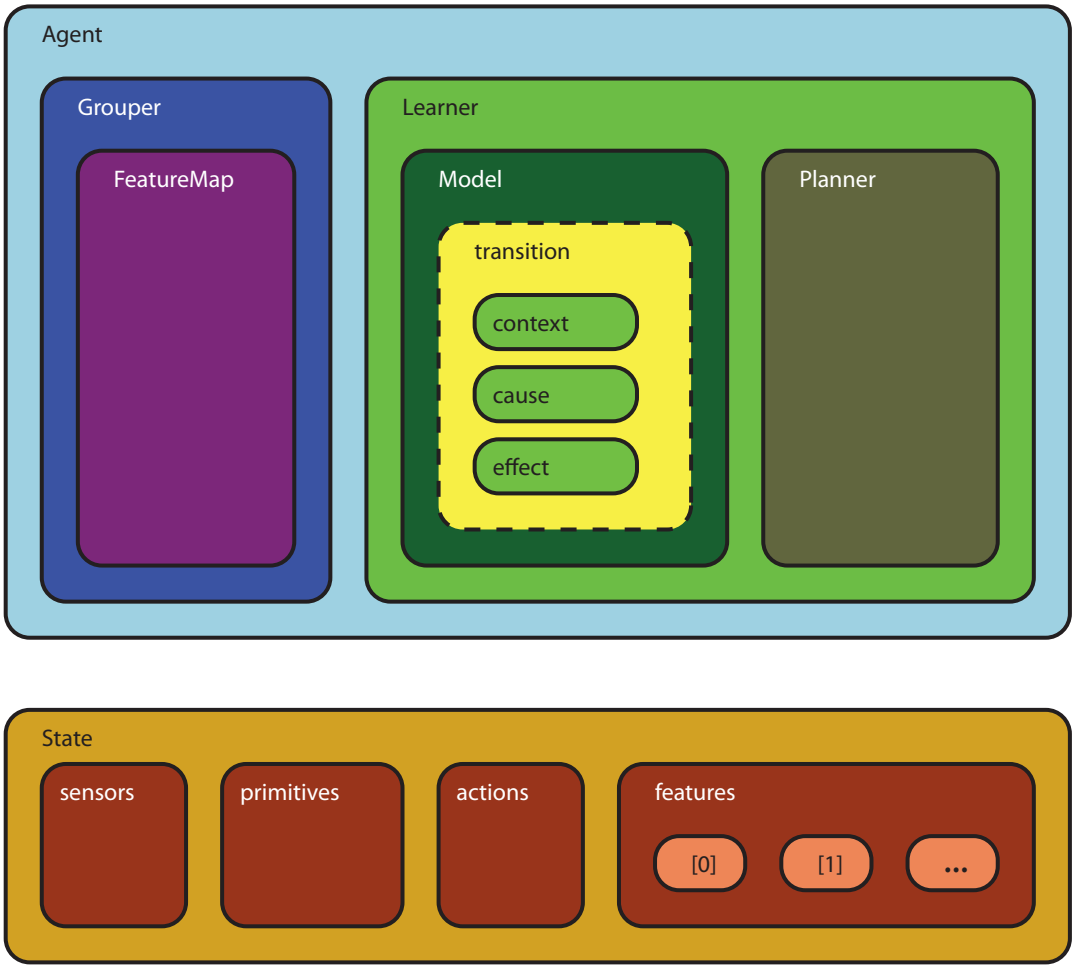


Figure 4.1: The class structure of BECCA's agent code.

### 4.1.1 `State`

- Represents the activity of inputs, features, and actions at a point in time. These are captured in the member variables `sensors`, `primitives`, `actions`, and `features`, all of which are numpy arrays, except for `features`, which is a list of numpy arrays.

### 4.1.2 `Agent`

- Performs executive functions, such as saving and reporting.
- Calls `Grouper.step()` and `Learner.step()` at each time step.

### 4.1.3 `Grouper`

- Forms inputs into groups.
- Finds features in each of those groups.
- Determines which features are active at each time step.
- Feeds those features back as additional inputs.

### 4.1.4 `FeatureMap`

- Lists the inputs that contribute to each feature.

### 4.1.5 `Learner`

- Chooses which feature to attend to.
- Maintains working memory using attended features.
- Calls `Model.step()` and `Planner.step()` at each time step.

### 4.1.6 `Model`

- Contains a library of `transition`s, each consisting of three `State`s: a `context`, a `cause`, and an `effect`. `transition`s are conceptual only, and are not actually member variables.
- Finds similar transitions in a library.
- Records new transitions in the library.

### 4.1.7 `Planner`

- Based on the `Agent`'s current state and history, chooses an action likely to bring reward.
- Refers to the `Model` in order to make predictions.

### 4.1.8 Utility modules: `utils` and `viz_utils`

- Perform general math chores that may be used by multiple classes.
- Visualize the internal states of classes for the user.

## 4.2 Is my agent better than the core agent?

The most natural question to ask after changing the agent is whether the new version is better than the old one. The answer will of course depend on what you mean by “better". The choice of performance measure has a great deal of influence on the performance levels of individual agents. The only necessary characteristic of a performance measure is that is produce a numerical value when applied to an agent. If you have a specific performance measure in mind, code it up and use it. If not, consider `benchmark.py`.

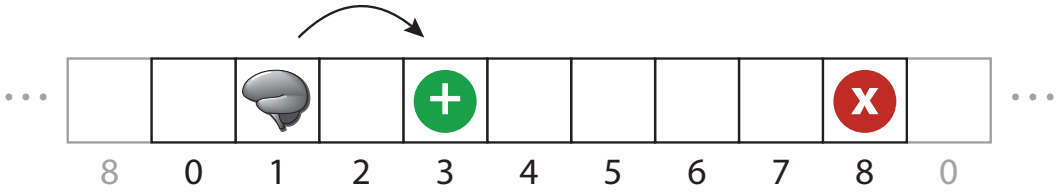


Figure 4.2: The one-dimensional grid world.

The worlds in `benchmark.py` are intended to be a testing battery that requires a broad learning capability to do well on. Admittedly, the battery members are currently very basic. Their complexity has been limited by Becca's performance as its bugs have become apparent. As Becca matures, the worlds in the benchmarking battery will also grow in number and sophistication. Since it is intended to be a changing metric, the version of Becca that a benchmark is released under will be an important identifier.

The worlds in this version of the benchmark are described briefly below.

### 4.2.1 `grid_1D.py`

As the name implies, this is a one-dimensional grid world. There are nine discrete states arrayed in a line, and the agent steps between them in increments of 1, 2, 3, or 4 steps in either direction. Stepping right from the rightmost state lands the agent in the leftmost state and vice versa, making the world into a ring. The fourth state from the left gives the agent a reward of 1/2 and the far right state inflicts a punishment (negative reward of -1/2). Every step the agent takes also incurs a cost of 1/100. This world was designed to be simple, yet non-trivial, and has proven very useful in troubleshooting BECCA.

### 4.2.2 `grid_1D_ms.py`

The `ms` stands for 'multi-step'. This world is identical to the `grid_1D.py` world, except that the agent may only take steps of one state at a time. Occa-

sionally random perturbations place the agent in new positions and it must make its way back to its goal using multiple steps, rather than in a single step. This world requires multi-step planning and is a challenge for some learning agents.

### 4.2.3 `grid_1D_noise.py`

The `noise` refers to the fact that the primitives reported by this world include a number of noise elements, whose values are randomly generated at each time step. Other than that, it is identical to the `grid_1D.py` world. Many reinforcement learning algorithms do not have a mechanism for handling noise or irrelevant data and this world poses a challenge to them. (BECCA of course handles them comfortably.)

### 4.2.4 `grid_2D.py`

### 4.2.5 `grid_2D_dc.py`

### 4.2.6 `image_1D.py`

### 4.2.7 `image_2D.py`

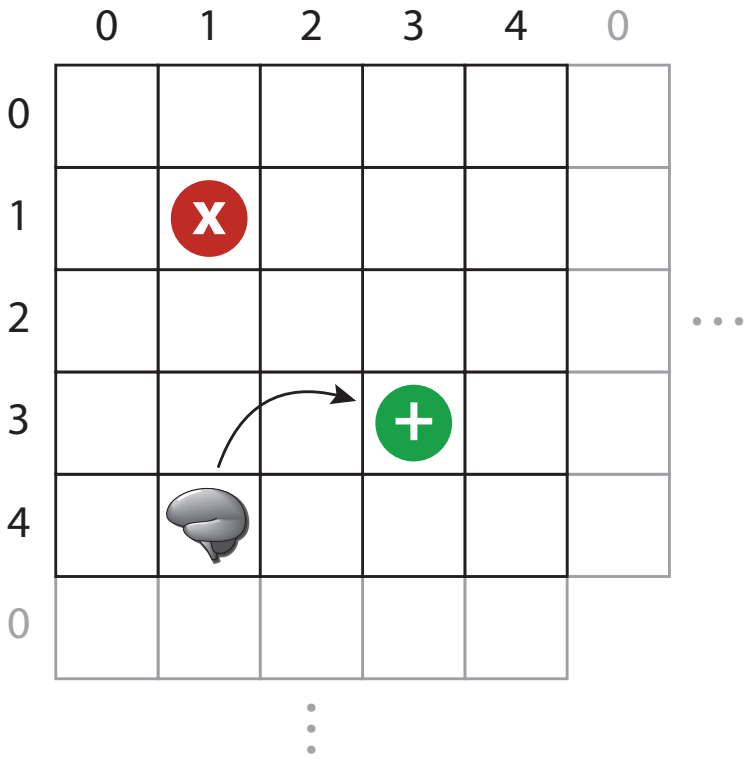


Figure 4.3: *
The two-dimensional grid world.

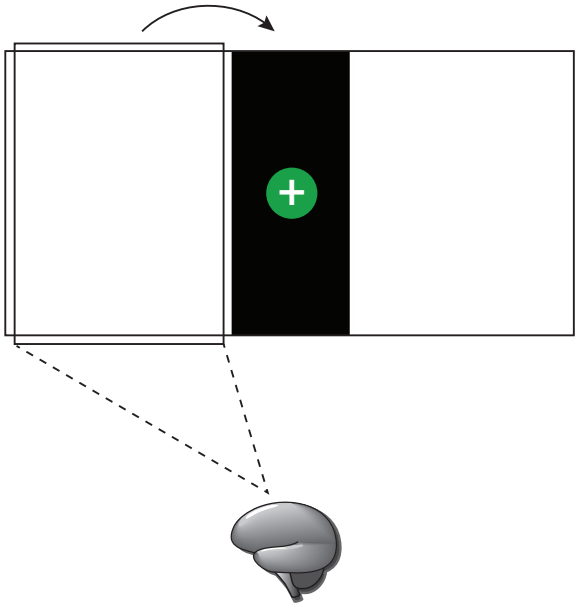

Figure 4.4: The one-dimensional image world.

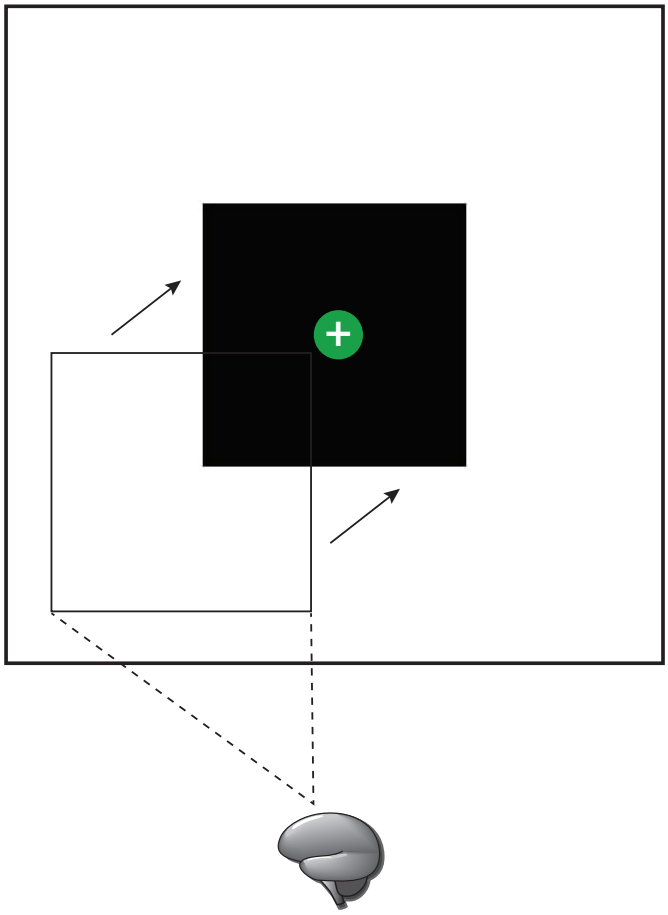

Figure 4.5: The two-dimensional image world.

# Chapter 5

# Share your agent with other BECCA users

# Bibliography

[1] B. Rohrer. Biologically inspired feature creation for multi-sensory perception. In *Second International Conference on Biologicall Inspired Cognitive Architectures*, 2011.

[2] B. Rohrer. A developmental agent for learning features, environment models, and general tasks. In *First Joint International Conference on Development and Learning and Epigenetic Robotics*, 2011.

[3] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, Massachusetts, 1998.