BECCA version 0.4.0

User's Guide

Brandon Rohrer

June 11, 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 it 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Or you can get the latest bleeding edge version (for which any of this documentation may already be outdated) from the github repository at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/editors (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. It gives a report of BECCA's performance in the worlds. The benchmark 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 designed to be simple tests of BECCA's fundamental learning capabilities. BECCA is an agent, in the sense that it makes decisions in order to achieve a goal, but it was created to be used in many different settings. The worlds tested in `benchmark.py` include one and two dimensional

[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). 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`.

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 and 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 writing and running of your first world. In the process it 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]

The architecture is modular. 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.

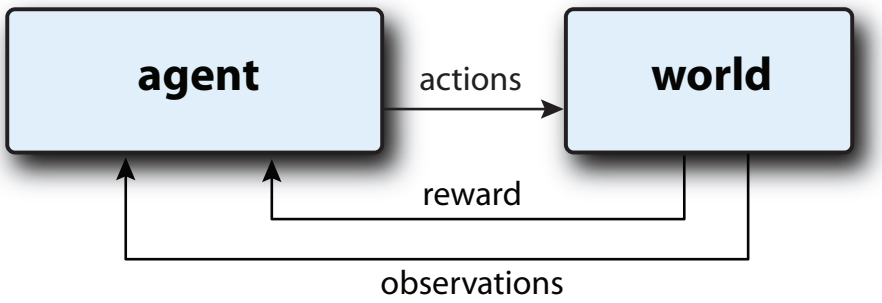

Figure 2.1: The agent-world coupling in BECCA. The agent selects actions to execute on the world, which in turn provides observations and reward feedback.

- 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 they become 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.

2.2 How do I make a *hello* world?

The quick and dirty 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!'

   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 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 defines how the world will respond to the agent. 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 lets the world give a world-specific interpretation of those values to the user.

Worlds of course may have any number of other member variables and methods for internal use. Others that 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
   ```

[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 (OO) programming. The concepts aren't that tricky, but they are extremely useful when discussing OO software, such as BECCA.

```
from agent import viz_utils
from worlds.hello import World
```

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 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?

I haven't finished writing up the justification and theory behind every part of the agent. This section will give just 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 shows the major classes and 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.

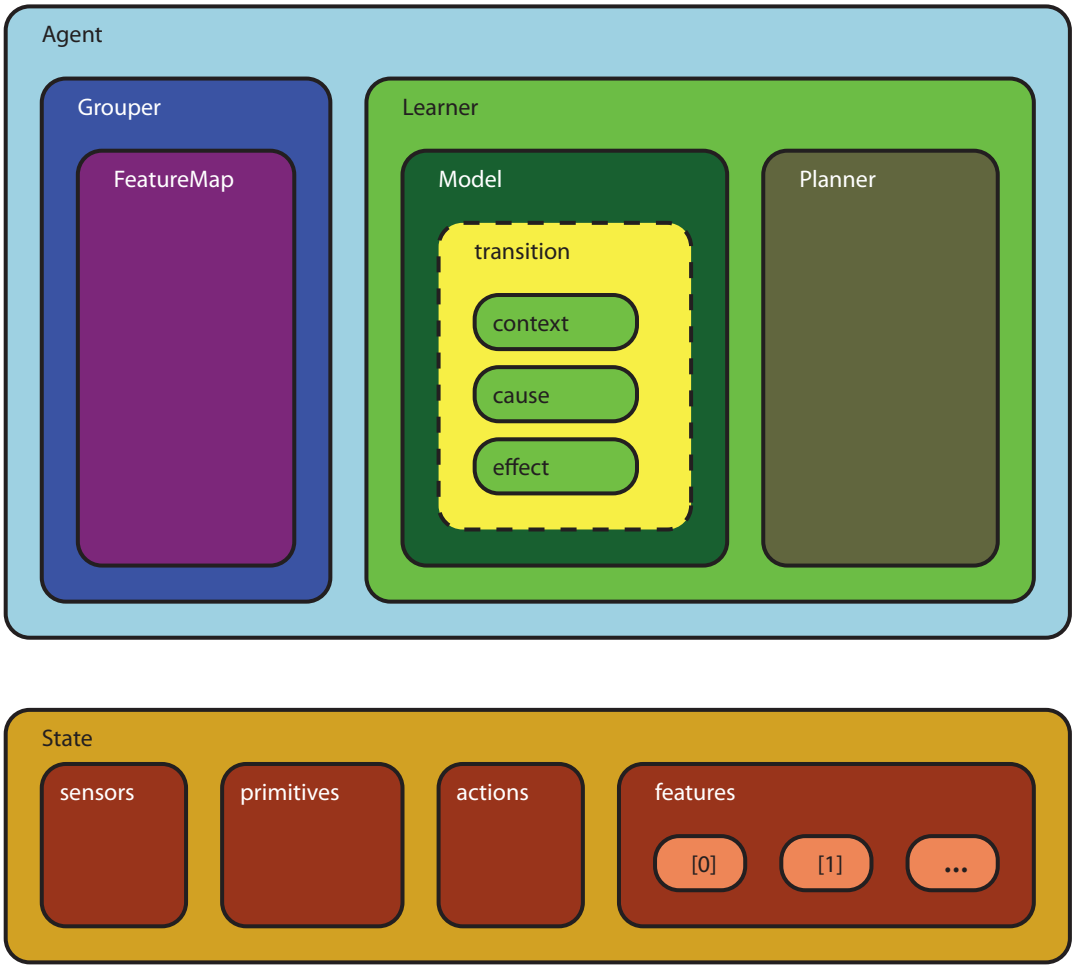

Figure 4.1: The class structure of BECCA's agent code.

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`

- Performs general math chores. May be used by multiple classes.
- Visualizes the internal states of classes for the user.

4.1.9 TCP/IP communication implementation: `server.py`

SeH wrote this module, creating a TCP/IP interface to the BECCA agent. It allows you to hook up a simulation or physical robot world that is speaking something other than Python. This extension enlarges the set of things that BECCA can talk to a great deal.

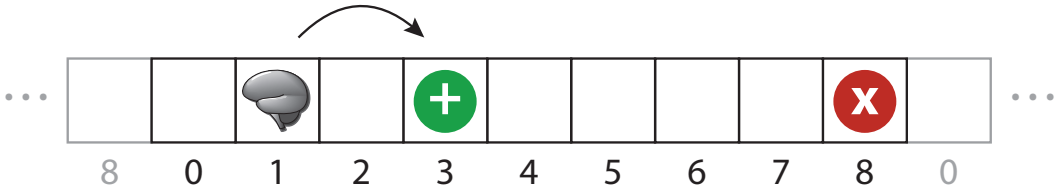

Figure 4.2: The one-dimensional grid world.

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.[1] 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`.

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 in this version of the benchmark are very basic. New worlds are added only when BECCA can perform well on all the old ones, each new world has brought to light more of BECCA’s bugs. This process is great for ironing out the kinks in BECCA, but a little slow. As BECCA matures, the worlds in the benchmarking battery will also grow in number and sophistication. Since the benchmark is likely to change with each release of BECCA, the version number of each benchmark 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 step in increments of one in either direction. Occasionally 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.)

[1] This is also a natural question to ask of other reinforcement learning agents. If you care to, you can code up any RL agent in Python and test it against BECCA on the benchmark. If you do, I'd be really curious to hear your results.

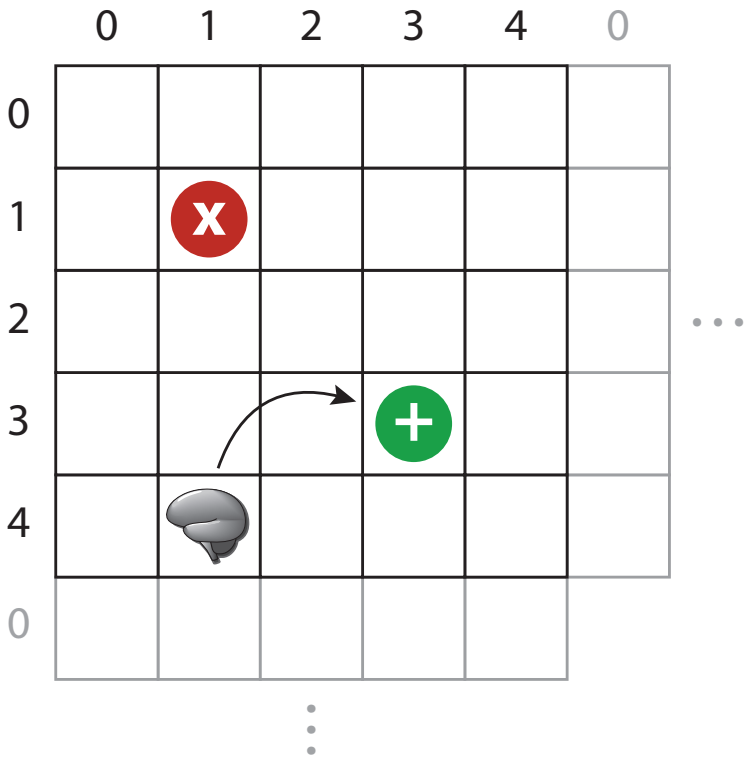

Figure 4.3: The two-dimensional grid world.

4.2.4 `grid_2D.py`

This is similar to `grid_1D.py`, but extended to two dimensions, with 5 rows and 5 columns. Each dimension wraps around, similar to the one dimensional version. This gives the world a toroidal topology and makes it impossible to fall off of it or run into any walls. The agent can make steps of 1 or 2 columns or rows in any of the four compass directions. A location in the lower right portion of the world gives a reward of 1/2, and a second location in the upper left portion of the world gives a punishment of -1/2. And, as with the one dimensional world, every step incurs a cost of 1/100. The agent's position in the world is represented using an array of 25 primitives, one for each possible location.

4.2.5 grid_2D_dc.py

The dc stands for 'decoupled' and refers to the fact that the column and row position of the agent are represented separately, each in an array of 5 primitive elements, giving this world a primitive array size of 10 rather than 25. This forces BECCA to consider multiple inputs simultaneously when making decisions, making it a slightly more difficult world than grid_2D.py.

4.2.6 image_1D.py

The two major differences between the image worlds and the grid worlds are:

1. In the image worlds position is continuous, rather than discrete.
2. In the image worlds observations are arrays of sensors, rather than arrays of primitives.

In the one dimensional case, the agent is looking at a mural (albeit a very boring one) and can shift its gaze right and left. It is rewarded for staring at the center of the mural.

The agent can move in increments of 1/4, 1/8, 1/16, and 1/32 of the mural width to both the right and left until it reaches the limits of the mural. All actions are also subject to an additional 10% random noise. The agent's field of view is half as wide as the mural. The reward region is 1/16 as wide as the mural and positioned at its center. When the center of the agent's field of view falls within the reward region, the agent receives a reward of 1/2. Within its field of view, the agent averages pixel values in a 10×10 grid to create a coarsely pixelated version of the mural. Each coarse pixel can be between 0 (all black) and 1 (all white). The pixel values and their complements (1 - the values) are passed to the agent in a 200 element sensor array.

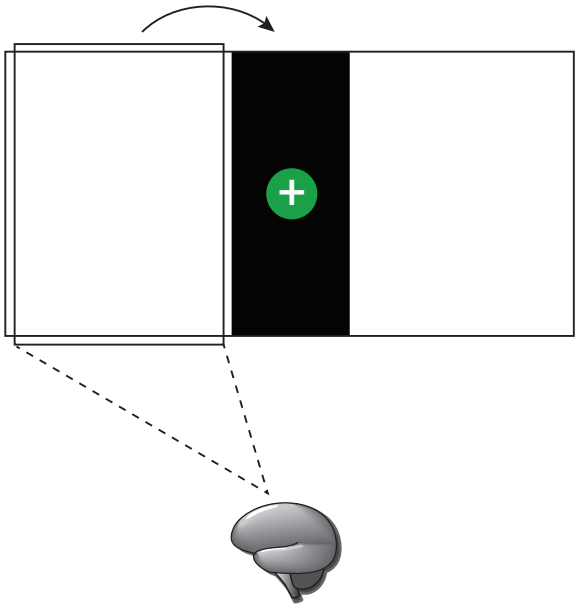

Figure 4.4: The one-dimensional image world.

4.2.7 `image_2D.py`

This world is similar in many ways to the one dimensional version, just extended to a second dimension. In it, the agent must center its gaze both horizontally and vertically to receive the full reward. One difference is that it pixelizes its world into a 5×5 grid (resulting in a sensor array of 50 elements). A second difference is that the agent receives a reward of 1/4 if its gaze is centered horizontally and an additional reward of 1/4 if its gaze is centered vertically.

4.2.8 `watch.py`

The watch world is not yet part of the battery. Right now it's a debugging tool. In it, the agent is exposed to image segments taken from a library of natural images, and it builds groups and feature representations from those image segments. It's helping to work out the kinks in the feature creation heuristic, although I hope to integrate it into a task for future versions of the benchmark.

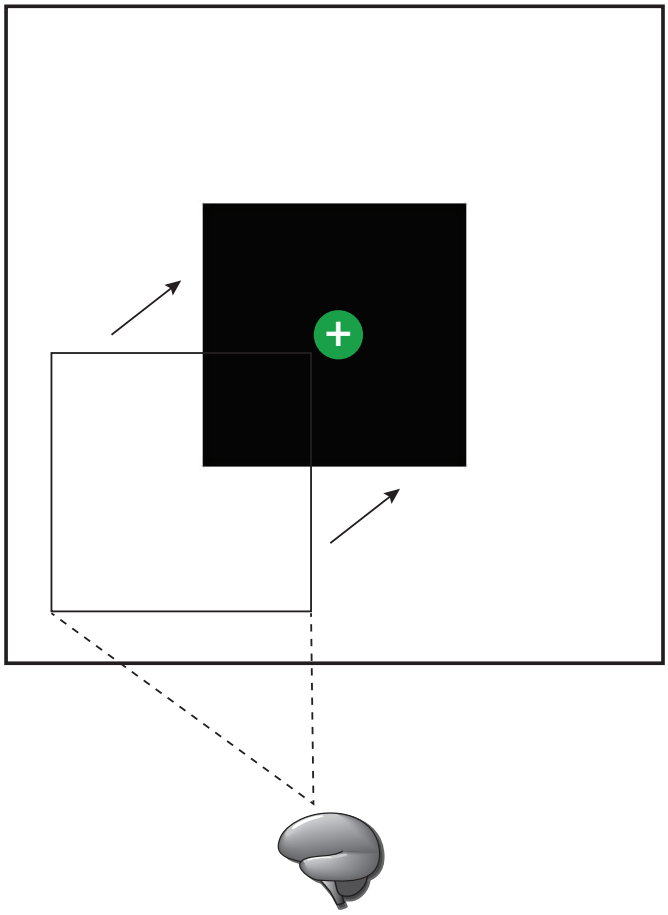

Figure 4.5: The two-dimensional image world.

Chapter 5

Share your agent with other BECCA users

So you've modified your agent, it's really cool, and now you want to share it. Everything in Chapter 3 about sharing your worlds applies to sharing agents too. Unlike other open source software projects, branching is not discouraged. The motivation behind BECCA is not to produce a slick tool that can be used as a black box. It is to make an architecture that gets used as often and as widely as possible.

To some extent every implementation will be custom. Ideally, every implementation will be driven by the well-articulated vision of one person or of a small group. BECCA 0.4.0 is intended to be a generic starting point for your work. If it were an ice cream flavor, it would be vanilla. It will be up to you to create chocolate, cappachino, and habanero. When you do, please pass them around so we can all get a taste.

If you have modifications, edits, or additions that you think may improve the core BECCA code, send a GitHub pull request to Matt Chapman, the curator of the core repository. An informal discussion will ensue in the becca_users group, and based on the outcome, your code will be incorporated into the core repository.

5.1 How do I make my code look like the rest of the core?

For any code destined for the repository, please follow these high level style goals (in rough order of priority):

1. Usability–a new user can apply it to their project with a minimum of effort and pain
2. Readability–a new developer can get oriented in the code with a minimum of effort and pain
3. Brevity–the number of packages, modules, methods, and lines of code are minimized
4. Performance–it works well and quickly

The implications of these priorities are that if performance can be increased by 0.2% by importing another package or adding another module, it's not worth it. But an increase of 50% would probably merit it. This may also mean neglecting some code development best practices because of their verbosity. Adding another layer of abstraction in places may make the core more easily extensible, but that may not be worth making it harder to navigate.

On low level style specifics, the PEP 8 Python style guide[1] and PEP 257 Docstring style guide[2] are the default word on style. However if there is ever a conflict between readability and PEP compliance, err in favor of readability.

Of course any work done to bring the existing core code into better alignment with the style goals will be greatly appreciated and applauded.

[1] `http://www.python.org/dev/peps/pep-0008/`
[2] `http://www.python.org/dev/peps/pep-0257/`

Appendix A

Project contributions

BECCA is still pretty young, but already several people have put in a lot of effort to develop it. This is an attempt to recognize some of those contributions. Please help me keep this list as complete as possible. If you don't see someone here who should be, add them or let me know. These are organized by name, with each contributor's work in reverse chronological order, and contributors listed in reverse chronological order of their most recent citation.

For time, consideration, and code given to the BECCA project:

Matt Chapman	2012 / 04 2012 / 01	For creating and releasing the openbecca.org website. For insights regarding licensing and open source community building.
Alejandro Dubrovsky	2012 / 02	For heroic efforts in making a first-pass port of BECCA from MATLAB to Python in its entirety in about one weekend.
SeH	2012 / 01	For insights regarding the code structure, version control, and cooperative workflow.

Appendix B

Publications and hacks

The number one goal in creating BECCA is to help machines do cool stuff. This section is a listing of some of those achievements. Please help me keep this list as complete as possible. If you don't see someone here who should be, add them or let me know. These are organized by name, with each contributor's work in reverse chronological order, and contributors listed in reverse chronological order of their most recent citation.

For using BECCA to do something cool and possibly publishing it:

Nick Malone	2012	For integrating BECCA 0.3.10 with a Barrett WAM arm, driving it to reach goal positions, and for publishing the results in a paper at ICRA.
Aleksandra Faust	2012	For searching for a target location in a simulated world using both visual and auditory data.
SeH	2012	For driving a Critterdroid simulation in Java through a TCP/IP socket and for publishing videos of the results.

Matt Chapman	2012 2012	For integrating BECCA (0.3.11 under Octave, slightly modified) with a Lego NXT robot via ROS to run the 1D grid task. For presenting BECCA/ROS/Lego NXT integration work to the LA Robotics Club.
Brandon Rohrer	2012 and earlier	For publishing a series of conference papers describing BECCA's development and use in a number of simulations and phyiscal robot systems.

Appendix C

Revision history

C.1 Version 0.4.0, June 8, 2012

- Ported to Python 2.7 from MATLAB. Props to to Alejandro Dubrovsky.
- Agent and World objects disentangled.
- Grouper object expanded to be responsible for all aspects of feature creation.
- Learner object created, responsible for all model building and learning.
- State object created, containing sensors, primitives, actions, and features.
- Model structure revised to reflect new state structure.
- Deliberate actions are now attended immediately.
- Benchmark module added for measuring performance on all worlds in the battery.
- Becca now works on all battery worlds, but with much room for improvement.
- Users Guide added.

Appendix D

The small print

[1] `http://www.opensource.org/licenses/mit-license.php`

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Appendix E

Citing this guide

Feel free to cite this guide in any BECCA related publications or reports you write. If you're using LaTeX, a good BibTeX entry looks like this:

```
@MISC{rohrer12a,
  AUTHOR =       {B. Rohrer},
  title =        {\textsc{Becca} 0.4.0 User's Guide},
  year =         {2012},
}
```

If you're writing in a WYSIWYG text editor, you can use this:

B. Rohrer. BECCA 0.4.0 user's guide, 2012.

or any other favorite format of yours.

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.