BECCA version 0.4.3

User's Guide

Brandon Rohrer

November 9, 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Or you can get the latest bleeding edge version (for which any of this documentation may already be outdated) from the github repository at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 collection of worlds that is 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. [22]

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.

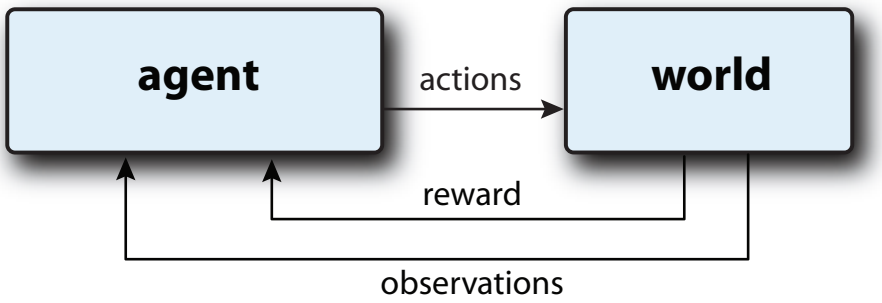

Figure 2.1: The agent-world coupling in BECCA. The agent selects actions to execute on the world, which in turn provides observations and reward feedback.

- 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 observations that are 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.

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, 14 is this version. 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`,[1] 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. 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

[1] 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, like BECCA.

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 = 1000
    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?

This section will give just an overview of the structure so that you have some idea about where to start making changes. A thorough description is included in Appendices A and B.

Figure 4.1 shows the major classes. 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.

4.1.1 `State`

- Represents the activity of primitives, features, and actions at a point in time. These are captured in the member variables `features`, which us a numpy

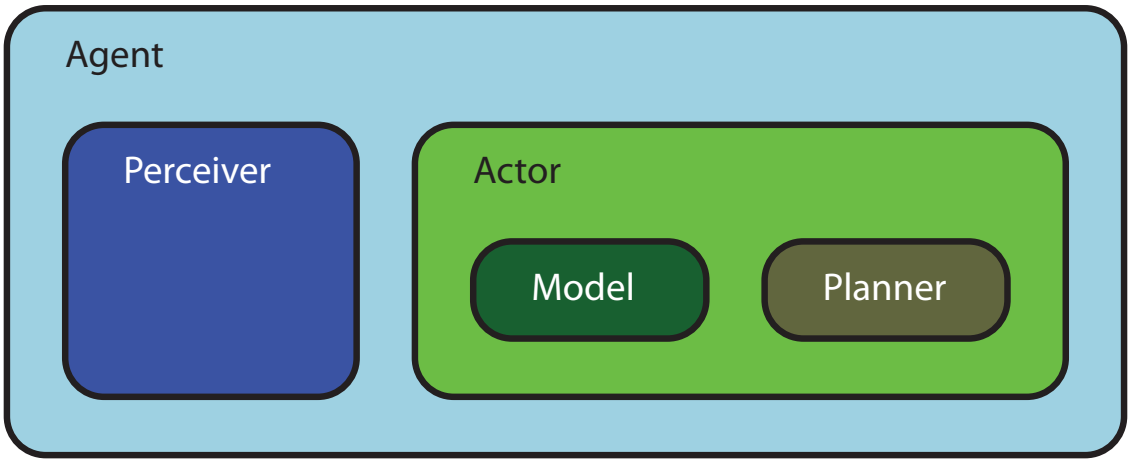

Figure 4.1: The class structure of BECCA's agent code.

array.

4.1.2 `Agent`

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

4.1.3 `Perceiver`

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

4.1.4 `Actor`

- Chooses which feature to attend to.

- Calls `Model.step()` and `Planner.step()` at each time step.

4.1.5 `Model`

- Contains a library of `transitions`, each consisting of four `States`: a `context`, a `cause`, an `effect`, and an `effect_uncertainty`. `transitions` are conceptual only, and are not actually member variables.
- Finds similar transitions in a library.
- Records new transitions in the library.
- Updates transitions in the library.

4.1.6 `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.

4.1.7 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.

4.1.8 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.

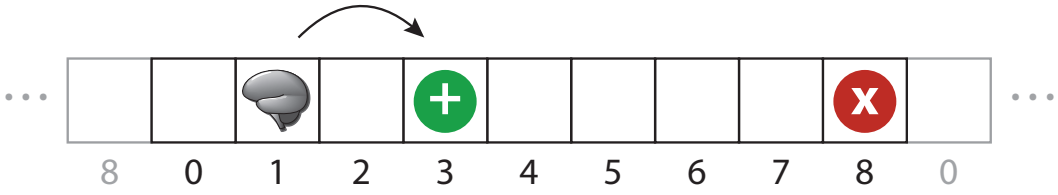

Figure 4.2: The one-dimensional grid world.

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.

[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.

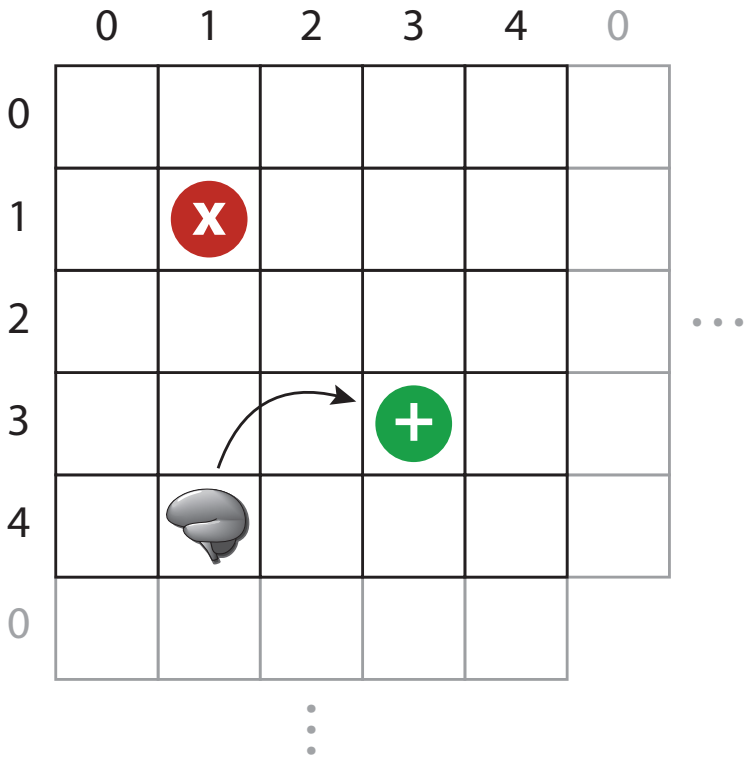

Figure 4.3: The two-dimensional grid world.

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.

It's worth noting that a naive $\mathcal{Q}$-learning approach to this problem would require creating a 2^203-element state table. The global information storage capacity is currently on the order of 2^70 bytes.

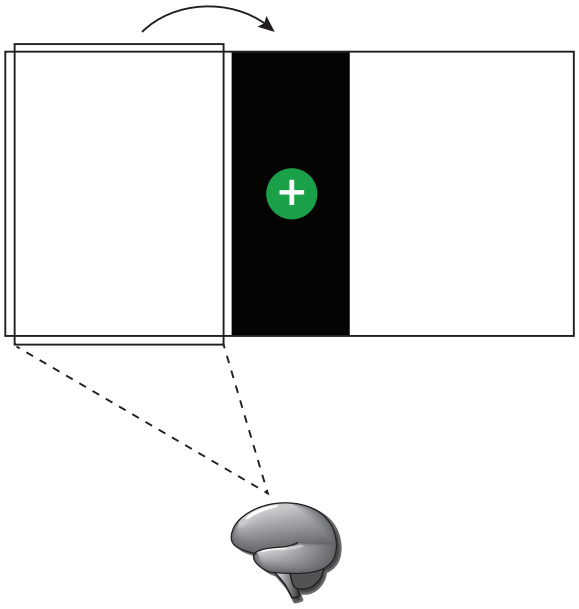

Figure 4.4: The one-dimensional image world.

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 extraction heuristic, although I hope to integrate it into a task for future versions of the benchmark.

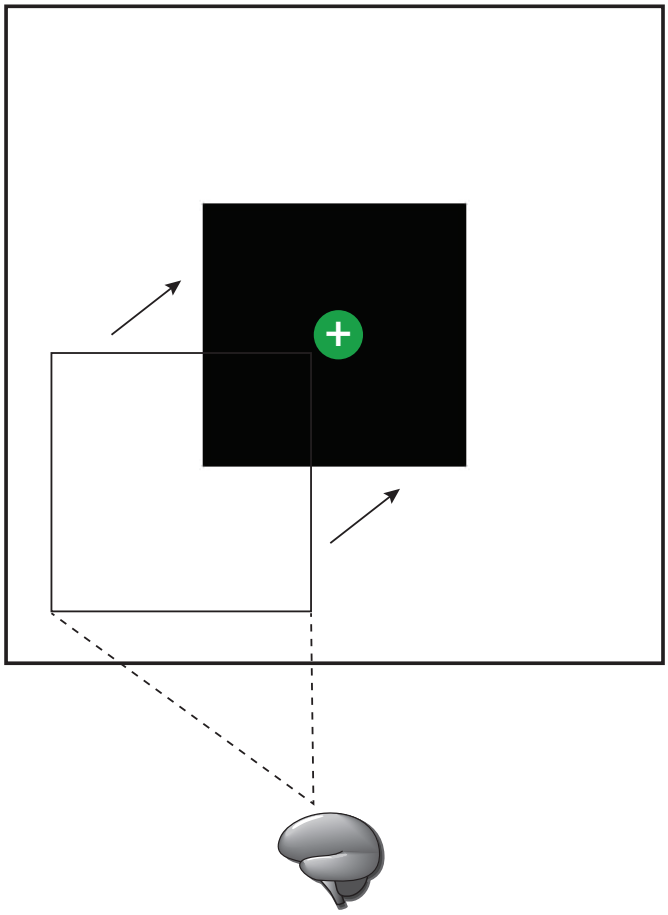

Figure 4.5: The two-dimensional image world.

4.2.9 `world_utils.py`

There are some jobs that crop up repeatedly in multiple worlds. This module is a common toolbox where functionality can be shared. So far this is limited to image processing functions, including center-difference filtering and visualizing features in image data.

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 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

How BECCA's feature extractor works

This appendix provides enough mathematical detail to fully describe the implementation of the feature extractor. Appendix **??** contains a detailed description of the reinforcement learner.

BECCA extracts features by greedily gathering commonly co-active inputs. (See Figure A.1.) BECCA requires no prior knowledge about the environment it is in, the nature of the experiences it is likely to have, or the sensors that provide its input data. The one constraint it places on the world is that its inputs be real valued between zero and one, a constraint that is well suited to representing pixel values. BECCA interprets each input as a fuzzy binary value, indicating the degree of presence of an attribute or the degree of activation of a sensor.

A.1 Co-activity based feature extraction

The extraction of new features is driven by how often each input channel is co-active with each of the other channels. The process of feature extraction finds sets of input channels that tend to be mutually co-active. Sets of commonly co-

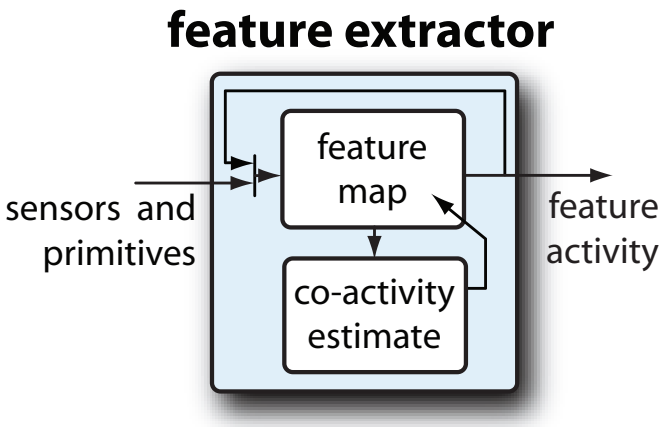

Figure A.1: Block diagram of BECCA's feature extractor.

occurring inputs represent pieces of observed structure in the world.

When two channels are co-active, both have a value close to 1. Co-activity is conceptually similar to correlation, except that correlation is also influenced by co-*in*activity. Co-activity of channel A with channel B is defined as the probability that A is active, given that B is active. For binary inputs, this can be expressed as follows:

$$\kappa(A, B) = p(b|a) \tag{A.1}$$

where $\kappa(A, B)$ is the co-activity of channel A with channel B, and $p(b|a)$ is the probability that channel B is active, given that channel A is active. For the calculation of co-activity with real-valued inputs and its interpretation, see Section A.4 at the end of this appendix.

Co-activity between input channels can be described as a fully connected directed graph, which is most conveniently represented as a fully-populated matrix. The graph must be directed, because the co-activity of channel A with B is not in general equal to the co-activity of channel B with A. For instance, if channel A is

fully active all the time and channel B is fully active only ten percent of the time, staying inactive the rest of the time, then from Equation A.1, $\kappa(A, B) = 1$ but $\kappa(B, A) = 0.1$. Co-activity is estimated statistically, and the co-activity graph is updated incrementally at each time step.

The co-activity of A with B, $\kappa(A, B)$, and the co-activity of B with A, $\kappa(B, A)$, can be combined to form the *mutual co-activity*, $\kappa_M(A, B)$ by taking the minimum of the two:

$$\kappa_M(A, B) = \min(\kappa(A, B), \kappa(B, A)) \tag{A.2}$$

While co-activity is asymmetric, the mutual co-activity will always be symmetric. Mutual co-activity between input channels can be described as a fully connected, *un*directed graph. Mutual co-activity represents the extent to which two inputs are active at the same time, but does so in a way that does not overestimate the co-activity with very active inputs.

The process of selecting a set of input channels can then be described as finding strongly interconnected subgraphs within the mutual co-activity graph. BECCA does this using a greedy subgraph construction heuristic. Subgraph construction begins when one graph edge (the mutual co-activity of some input channel with another) exceeds a threshold, C_1. Those two input channels, say A and B, are added to the new feature. Then, the channel with the highest minimum mutual co-activity with existing feature members, N_m, is added:

$$N_m = \text{argmax}_N \min(\kappa_M(A, N), \kappa_M(B, N)) \tag{A.3}$$

This agglomeration process continues until the mutual co-activity with the next candidate does not exceed another threshold, C_2. The inputs that contribute to a feature may be considered its parents (although there ay be many more than two.) An input may be collected during the extraction processes of several different features, becoming a parent several times over.

Some input pairs are not allowed to be the initial nucleus around which a feature is agglomerated. Inputs are prohibited from forming a nucleus with any

of their co-parents from another feature, and also with all of their co-parents' parents and ancestors. However, once a valid nucleus has been created, co-parents may join the new feature. Disallowing certain nucleations prevents inbreeding in feature extraction in which similar features are created over and over again.

A.1.1 Feature map

Features created by this process consist simply of a set of inputs. They are sparse in the sense that a limited number of inputs contribute to each feature. More specifically, the features are l_0 sparse, meaning that the number of contributing inputs is low, rather than the sum of their contributions (l_1 sparsity) or the square of their contributions (l_2 sparsity).

After a feature is created, it is added to the feature map. The feature map specifies which inputs are averaged to give the initial activation, a, for the feature. It acts like a mask on the input set. Because each input represents the graded output of a binary-type sensor or primitive, all inputs to a feature contribute equally.

A.2 Feature activity

Features compete for the contributions of their inputs in a process resembling lateral inhibition. (See Figure A.2.) Features with an initial activation, $\hat{a}$, that is high suppress their inputs' contributions to other features. This helps prevent ambiguity in feature activations and provides a soft winner-take-all behavior among features that share several inputs.

To be more precise, feature activities interact as follows. For an input that contributes to several different features, its contribution to feature i will be inhibited by γ_i, given by:

$$\gamma_i = \left(\frac{\hat{a}_i}{\hat{a}_{\max}} \right)^{C_3} \tag{A.4}$$

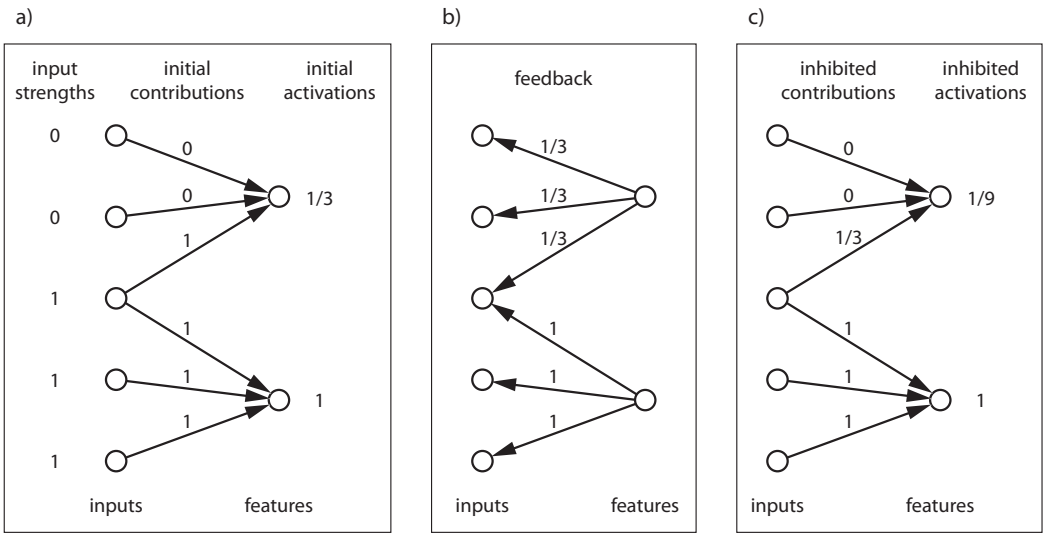

Figure A.2: An example of competitive feature activation. **a)** Inputs activate the features to their initial activity levels, $\hat{a}$. **b)** The $\hat{a}$ values are fed back to the inputs. The feature with the highest $\hat{a}$ for a given input suppresses that input's contributions to all other features. **c)** As a result of this suppression (with $C_3 = 1$ in Equation A.4), the final activities, a, of competing features are weakened.

This formulation weakens all the contributions of the input except for the one to the most active feature, the one with $\hat{a}_{\max}$.

A.2.1 Input energy

The input magnitudes used to calculate co-activity can be lower than the original input magnitudes. Conceptually this is analogous to inputs having limited energy. If an input excites a feature strongly, then it has less energy available to build up new features with other inputs. But if an input excites no features, then it has plenty of energy available to establish co-activity with other inputs.

The fraction of its magnitude that an input devotes to co-activity is given by γ_κ:

$$\gamma_\kappa = 2^{-C_4 \sum \gamma_i} \tag{A.5}$$

When an input excites more features, $\sum \gamma_i$ gets larger, forcing γ_κ smaller. This in effect creates a soft limit on the number of features an input can contribute to.

A.3 Hierarchy, time, and domain knowledge

Three additional details of BECCA's operation are necessary to achieve its full capabilities.

1. The first of these is the recurrent handling of its inputs and outputs. In addition to being BECCA's output, the feature activities at each time step are fed back into the grouper as the next time step's inputs. This allows features to be grouped with each other and other inputs to form new groups, in which higher-level features will be created. Due to its recurrent structure, this process may be repeated indefinitely, creating a hierarchical feature structure with an arbitrarily large number of hierarchy levels. In practice, new

groups are only formed as often as co-activity dictates, which is limited by the structure inherent in the data.

2. The second detail is the temporal decay of inputs. After an input is active, its channel retains some residual activity on subsequent time steps, even if it is not externally stimulated. This decay is of a geometric nature, resulting in an exponential decay curve:

$$s_{t+1} = C_5 s_t, \qquad 0 < C_5 < 1 \tag{A.6}$$

   The result of this decay is that temporal aspects of data can be captured in the features. For instance, if input B is always active following input A, but never at the same time, retaining a decayed ghost of A's activity allows it to be correctly correlated with that of B to form a feature with temporal structure.

3. The third detail is the direct injection of user-created features to capture domain knowledge. As presented so far, BECCA insists of deriving its own representation of the data from scratch. But in most applications, designers already have a pretty good guess about which features are likely to be informative in solving a given problem. BECCA has a mechanism for incorporating engineered features as well. It looks for a vector of user-created feature primitives at each time step, in addition to raw sensor data. It handles the primitives differently that other inputs, concatenating them with the feature activities of all the created groups. As such, the primitives are both fed back as inputs, and included with the feature extractor's output. The only constraint on primitives is that they also be real valued between zero and one. BECCA handles both raw sensor inputs and feature primitives as fuzzy binary inputs, each indicating the graded presence of a certain attribute.

A.4 Co-activity

BECCA's co-activity calculation is incremental, with the update method for real valued inputs given by the following:

$$\Delta\kappa(A, B) = C_6 a(ab - \kappa(A, B)), \qquad 0 < C_6 < 1 \tag{A.7}$$

Where κ is the co-activity, and a and b are the activity levels of channels A and B respectively. For values of a and b that remain constant over time, a convergence value for $\kappa(A, B)$ can be found from Equation A.7. For the convergence criteria to be met, $\Delta\kappa(A, B) = 0$. That implies $ab - \kappa(A, B) = 0$ (assuming $a > 0$), giving the steady state value of

$$\kappa(A, B) = ab \tag{A.8}$$

The steady state value is symmetric in a and b, even though the update rule is not. For constant-valued a and b, $\kappa(A, B)$ and $\kappa(B, A)$ converge to the same value, but at different rates.

Another instructive case is that of binary and stochastic inputs. Consider the independent input channels A and B such that

$$A \quad | \quad p(a = 1) = 0.5, \qquad p(a = 0) = 0.5 \tag{A.9}$$
$$B \quad | \quad p(b = 1) = 0.1, \qquad p(b = 0) = 0.9 \tag{A.10}$$

The structure of the update rule guarantees that $\Delta\kappa(A, B) = 0$ when $a = 0$, meaning that $\kappa(A, B)$ will not be modified in any way. It will only be updated when $a = 1$. The steady state value for $\kappa(A, B)$ can then be calculated by observing that when $a = 1$ the update rule reduces to

$$\Delta\kappa(A, B) = b - \kappa(A, B) \tag{A.11}$$

The fact that B is independent from A implies that $\kappa(A, B)$ will converge to the mean of $b = 0.1$. Following a similar set of calculations, it can be shown that $\kappa(B, A)$ converges to 0.5 . This illustrates the effect of asymmetry in the update rule on non-constant input values. The results can be verbalized as "Channel A

is co-active with Channel B ten percent of the time, but Channel B is co-active with Channel A fifty percent of the time." An intuitive metaphor is that of restless kindergarteners at nap time. The teacher has told all the children to lay still and close their eyes, but unable to help themselves, some of the children occasionally peek. $\kappa(A, B)$ is the probability that, when child A peeks, she will see child B's eyes open and vice versa.

One final observation about $\kappa(A, B)$ is that it is bounded both above and below by the product of a and b. Since $0 \leq a \leq 1$ and $0 \leq b \leq 1$, it follows that

$$0 \leq \quad ab \quad \leq 1 \tag{A.12}$$

$$0 \leq \quad \kappa(A, B) \quad \leq 1 \tag{A.13}$$

Appendix B

How BECCA's reinforcement learner works

BECCA's reinforcement learner starts with the set of feature activities passed from the feature extractor at each time step. (See Figure B.1.) It attends to one of the features, passes the attended feature together with the full feature activities and reward to the model, and chooses an action to pass to the world. Each of these steps is described in detail below

B.1 Attention

Each of the feature primitives, the actions, and the created features is eligible for attention. The most salient feature in this set is attended at each time step. In practice, this means that in the array representing attention, the attended feature value is set to one and all others are set to zero. The criteria for salience are still under heavy development, and are likely to change in future versions of BECCA, but currently they are simple. The most salient feature is the one with the largest magnitude, unless a deliberate action was taken on the previous time step, in which case that action is attended. A small amount of random noise is added to the feature activities (typically 1/20 of one percent) to act as a tie breaker be-

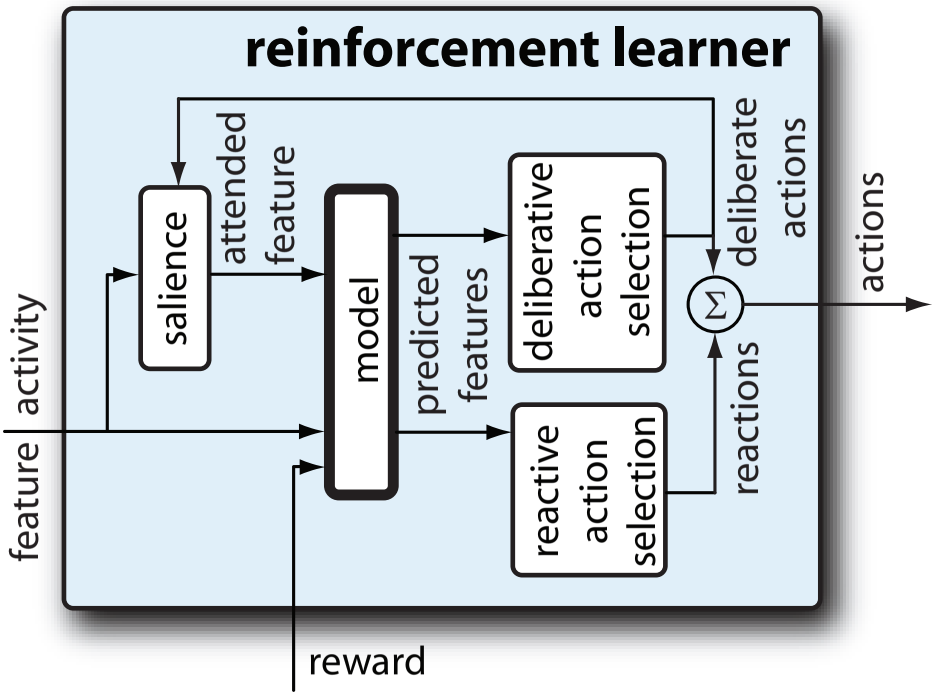

Figure B.1: Block diagram of BECCA's reinforcement learner.

tween features that may be active at exactly the same level. This condition occurs commonly when working with simulated worlds.

Future candidates for salience calculation include:

- **Recency** Recently attended features would be penalized. Rarely observed features would be more salient.
- **Surprise** Expected features would be less likely to be attended. Unexpected ones would be more salient.
- **Goal relevance** Features that have been selected as intermediate goals would be more salient.

B.2 Model

The model is the core of the reinforcement learner. It contains snippets of the agent's experience in the form of *transitions*. The collection of transitions constitutes the whole of the agent's experience. Each is a small piece of memory. Taken together, the set of transitions is a distillation of the agent's interactions with its world. On the agent's first time step, the model is empty, but as it gathers experience it gradually populates its model, increasing the fidelity of its representation of the world.

B.2.1 Transition structure

Each transition is of the form (`context`, `cause`, `reward`, `effect`) and captures a short sequence of events. (See Figure B.2.) Contexts, causes, and effects may contain state or action information or both.

Transitions also contain a quantification of the uncertainty in both effects and rewards. `context–cause` pairs uniquely define a transition. The effect and reward that results from a given context and cause may vary a great deal each

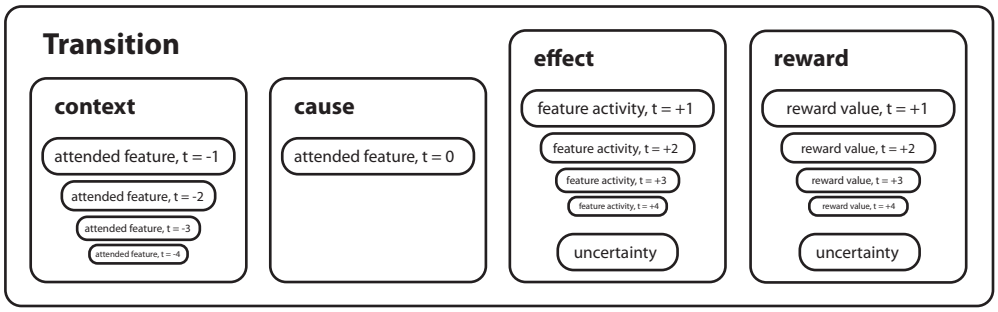

Figure B.2: The structure of a transition in the model.

time the transition is observed. Rewards and effects are the expected value of starting in a given context and executing a given cause. The reward uncertainty and effect uncertainty are the expected error on those predictions.

Context

Context is a combination of the several most recent attended features. It captures the agent's recent history of attention, summarizing the most salient parts of its world. Specifically, if the current time step is labeled t_0, the context includes the attended features from t_{-D_1} to t_{-1}, where D_1 is a user-defined integer, typically small. The attended features are decayed according to their age, as follows:

$$\tilde{\alpha}_{-i} = \alpha_{-i} D_2^{i-1}, \text{ for } i = 1, 2, ..., D_1 \tag{B.1}$$

where α_{-i} is the magnitude of the attended feature at time step $-i$ (always 1 in the current implementation), $\tilde{\alpha}_{-i}$ is the decayed magnitude of the attended feature, and D_2 is a user-selected real value between zero and one.

The context is the the combination of all the decayed attended features. In the case where the same feature has been attended more than once within the last D_1 time steps, the decayed magnitudes are added in a nonlinear way, such that the resulting magnitude is guaranteed to be less than or equal to one. This is done by mapping the two values to be summed, which are known to fall on the interval

$[0, 1]$ to the interval $[0, \infty)$ as follows:

$$a' = \frac{1}{a} - 1 \tag{B.2}$$

Then the two operands are added normally. The result, which also lies on $[0, \infty)$ is then mapped back to $[0, 1]$ using the inverse operation:

$$a = 1 - \frac{1}{a' + 1} \tag{B.3}$$

The result is a bounded sum operation that ensures the magnitude of the result will always lie between zero and one, given that its operands did.

Cause

The cause is simply the attended feature at the current time step, t_0. It may be an action, in which case the `context–cause–effect` sequence resembles a state–action–state tuple. But it can also be a primitive feature or created feature, in which case the transition represents an observed sequence of features, rather than a record of the agent's interaction with the world

Effect

The effect is the combination of the feature activities from t_1 through t_{D_1}. These are decayed and summed the same way attended features are decayed and summed to produce contexts, only in the opposite time direction, with immediate feature activity being undecayed and future feature activities increasingly decayed.

Effects are composed of feature activities, rather than attended features because they contain a more complete picture of the effects of a given context and cause. Since the transition is identified by the uniqueness of the context and cause

alone, the effect can contain a lot of information without making the transition overly specific.

Effect uncertainty

Each time a transition is observed, the effect is updated to reflect the result. It can be the case that a wide variety of effects may result from the same context and cause. The effect captures something resembling the average of these effects. The effect uncertainty provides an estimate of the expected difference between the effect and effects that will result from future encounters with the same context and cause.

The effect uncertainty is useful in making predictions. Particularly when the uncertainty is low, deviations from expected effects can be flagged as highly salient.

Reward

The reward that results from each observation of the transition is used to update an estimate of the transition's expected reward. This quantity explicitly describes the desirability of the transition, a value made use of in planning.

The reward associated with a transition combines reward signal received over the next several time steps, decayed and summed in the same way as the effect. This produces a trace which allows the transition to represent outcomes of the context and cause that are desirable but may be slightly delayed.

Since reward may be either positive or negative, the bounded summation described in equations B.2 and B.3 must be slightly extended. When both operands are negative, this is done by inverting the signs before summing them, then reversing the sign again afterward; the operation is symmetric with respect to sign. When the two operands are of opposite signs, they are simply added. Since they are both assumed to have an absolute value less than one, the resulting absolute

value can only be less than or equal to absolute values of the operands. Thus, the result will still fall on [-1, 1].

Reward uncertainty

As with the effect, the reward uncertainty is the expected value of the difference between the expected reward and future observations of reward. It is a quantification of confidence in the reward estimate.

Count

The count is a reflection of the number of times a transition has been observed or has been used in planning. It determines the update rate for transitions and whether and when they are forgotten.

B.2.2 Adding transitions

At each time step, the model compares the current context and cause to that of all of its existing transitions. A matching transition is one that has both

1. the same cause and
2. a sufficiently similar context.

Context similarity is determined by treating all the features (including basic features and actions) as dimensions of the same space. Both the current context and the transition context are vectors in that space. The similarity measure, σ, is based on the angle, θ, between two vectors. If $\sigma(\mathbf{a}, \mathbf{b})$ is the similarity between two vectors $\mathbf{a}$ and $\mathbf{b}$, then:

$$\sigma(\mathbf{a}, \mathbf{b}) = 1 - \frac{\theta}{\pi/2} \tag{B.4}$$

This similarity measure has several desirable properties:

1. $\sigma(\mathbf{a}, \mathbf{b}) = \sigma(\mathbf{b}, \mathbf{a})$ It is symmetric.
2. $\sigma(\mathbf{a}, \mathbf{b})$ is on $[0, 1]$
3. If $\mathbf{a}$ and $\mathbf{b}$ share no nonzero elements, $\sigma(\mathbf{a}, \mathbf{b}) = 0$
4. $\sigma(\mathbf{a}, \mathbf{b}) = 1$ if and only if $\mathbf{a} = c\mathbf{b}$, where c is a constant greater than 0, that is, if $\mathbf{a}$ and $\mathbf{b}$ are pointing in exactly the same direction.

Two contexts $\mathbf{a}$ and $\mathbf{b}$ are considered sufficiently similar if $\sigma(\mathbf{a}, \mathbf{b}) > D_3$, where D_3 is a user selected real value between 0 and 1, typically closer to 1 than 0.

If more than one matching transition is in the library, the one with the highest similarity wins. If no matching transitions are in the library, the new transition is added. This requires waiting until the full effect and reward has been observed, so there is a delay of D_1 time steps before new transitions are added to the model.

B.2.3 Modifying transitions

In instances where the current context and cause do match an existing transition, the effect, reward, and their uncertainties are updated based on the new observation.

When a transition is updated, the following events occur:

- The count is increased by one.
- The effect is modified to incorporate the newly observed effect.

- The effect uncertainty is modified to incorporate the difference between the effect and the newly observed effect.
- The reward is modified to incorporate the newly observed reward.
- The reward uncertainty is modified to incorporate the difference between the reward and the newly observed reward.

In order to perform the updates, an update rate, μ is first calculated:

$$\mu = \min(1.0, D_4 + \frac{1 - D_4}{\nu}) \tag{B.5}$$

where ν is the count of the transition being updated and D_4 is a user selected real value between zero and one. For large values of ν, μ approaches D_4. It can be considered a lower bound on the update rate. For values of D_4 close to zero, transitions cease to change significantly after they have been observed many times. For higher values of D_4, they remain malleable throughout their lifetimes.

Each feature in the effect is updated, shifted toward the newly observed value by a fraction, μ, of the way. For an effect feature value, f, and a new effect observation $\hat{f}$ the updated feature value, f' is given by:

$$\Delta f = \hat{f} - f \tag{B.6}$$
$$f' = f + \mu \Delta f \tag{B.7}$$

By the same mechanism, the effect uncertainty features, f_ϵ, are updated using Δf:

$$\Delta f_\epsilon = \Delta f - f_\epsilon \tag{B.8}$$
$$f'_\epsilon = f_\epsilon + \mu \Delta f_\epsilon \tag{B.9}$$

Reward, ρ, and reward uncertainty, ρ_ϵ are updated using the same mechanism as well, with the same update rate:

$$\begin{aligned} \Delta\rho &= \hat{\rho} - \rho && \text{(B.10)} \\ \rho' &= \rho + \mu\Delta\rho && \text{(B.11)} \\ \Delta\rho_\epsilon &= \Delta\rho - \rho_\epsilon && \text{(B.12)} \\ \rho'_\epsilon &= \rho_\epsilon + \mu\Delta\rho_\epsilon && \text{(B.13)} \end{aligned}$$

B.2.4 Forgetting

Every new transition encountered is stored in the model, but many transitions are observed only rarely and result in neither reward nor punishment. These are periodically removed from the model in order to keep BECCA's processing speed up and its space requirements modest. Two events can trigger forgetting: when the number of transitions in the model exceeds a user defined value, D_5, or when the number of time steps since the last forgetting episode exceeds a user defined value, D_6.

During forgetting, the count values of all transitions in the model are first decremented. Transition counts gradually decay over time. After every D_6 time steps the count of every transition is aged according to the relation:

$$\nu' = \nu - 1/\nu \qquad \text{(B.14)}$$

where ν' is the decayed count. The nonlinear nature of the decay is such that count values above 5 or 10 will decay very slowly, but low count values will decay more quickly. This is designed to retain repeatedly-observed transitions, even if they haven't been seen in some time.This formulation for memory degradation is intended to degrade rarely-observed transitions more rapidly than those that have been observed several times. To complete forgetting, all transitions with count values lower than zero are removed. This frees up the memory they occupied for

storing new transitions, and speeds up operations that require comparisons across the whole model.

B.3 Planner

The planner is the portion of the reinforcement learner that selects an action at each time step. There are several options available to the planner: observation, exploration, deliberation, and reaction.

B.3.1 Observation

When observing, the planner chooses no action. Observation is necessary for learning, because when a deliberate action is chosen, it is attended. If deliberate actions are made at every time step, the agent will never attend to its environment and will have no way to learn the features that result from its actions.

In its current implementation, the agent chooses each time step whether to observe or make a deliberate action by generating a random number between 0 and 1 and checking whether it is less than a user selected value, D_7. This is a naive mechanism. A more sophisticated mechanism would take into account the extent to which the agent is able to make accurately predictions, and it would perhaps include information about the agent's recent reward history. These may be incorporated in future versions of BECCA.

B.3.2 Exploration

On time steps where the agent determines that it will make a deliberate action, a certain fraction of these are exploratory. In its current implementation, exploratory actions are taken whenever a deliberate action has been called for, and a random number between 0 and 1 is less than a user selected vale D_8—the same mechanism

used to decide when to observe. It is naive in this instance as well, and it could be improved in the similar ways.

Significant attention has been given to exploration in learning research. Also termed "instrinsic motivation", "active learning", "goal exploration", and "directed exploration", exploration can be driven by the how well the agent can predict the results of its actions and the behavior of its environment. Specifically it can:

1. Search out unpredictable transitions.
2. Search out *somewhat* unpredictable transitions. Look for experiences that it can predict approximately, but not perfectly.
3. Search out transitions where new experiences most rapidly decrease their unpredictability.

These are all candidates for implementation in future versions of BECCA.

B.3.3 Deliberation

In the cases were the agent decides not to observe and not to explore, it deliberatively makes a plan that it deems likely to lead to the highest reward. Deliberation is the process of choosing one transition to pursue. In order to do this, all transitions are assigned a fitness, ϕ, given by:

$$\phi = \sigma(\mathbf{a}, \mathbf{b})\rho\phi_\nu \tag{B.15}$$

where

- $\mathbf{a}$ is the current context
- $\mathbf{b}$ is the context of the transition

- $\sigma(\mathbf{a}, \mathbf{b})$ is the similarity between the current context and the context of the transition, as defined in Equation B.4
- ρ is the expected reward associated with the transition
- ϕ_ν is the a fitness factor determined by the transition's count, and is given by:

$$\phi_\nu = \frac{2}{1 + e^{-2\frac{\log(\nu)+1}{3}}} - 1 \tag{B.16}$$

  which, despite its complicated appearance, is just a mapping of count, ν, onto $[0, 1)$, weighting more heavily transitions that have been seen more often.

The transition with the highest fitness is chosen as the deliberative plan for the time step. If the cause associated with the transition is an action, that action is selected and passed on to the world. It is also flagged to be attended on the following time step, making sure that it is acknowledged by the model and that any changes in the world that result from it are appropriately attributed to it.

Transitions selected as plans are also updated in the model. The updating mechanism is the same as for transitions that have been previously observed by the model with one exception. The update rate μ is scaled by the similarity of the selected transition, $\sigma(\mathbf{a}, \mathbf{b})$, and the count is incremented by $\sigma(\mathbf{a}, \mathbf{b})$, rather than one. This causes selected transitions with close matches to the current context to be updated more aggressively than transitions that are less similar. Updating transitions when they are used in planning has proven to be an important aspect of learning, particularly in stochastic environments. However, when the transition chosen as a plan is only a distant match to the current transition, this updating is less justified.

When the selected transition does not have an action for the cause, no action is selected. A mechanism exists within BECCA to take whatever the cause is and make it an intermediate goal. This makes it possible to use high-level abstract transitions to make abstract plans, and then expand those to low-level implementations appropriate to the specific circumstances of the agent. However, the current set of benchmark tasks haven't demanded this functionality, so in an exercise of engineering discipline, this capability has not yet been activated. As the bench-

mark expands to include more challenging tasks, it is likely that this decision will be reconsidered.

B.3.4 Reaction

In some cases it may be desirable to have the agent choose actions without having to deliberate. In some worlds, in some situations, actions may be so predictably rewarding that they should be executed every time without deliberation. Pulling fingers away from a hot stove or eating jelly beans are two such behaviors commonly observed in humans, for instance. Reactive actions provide a mechanism for this. The entire set of feature activities are passed to the model and could potentially be used to identify matching transitions without invoking attention. Highly rewarding transitions may be instigated without deliberation, unless actively suppressed.

Although not implemented in the current version of BECCA, reactive actions have been implemented in previous versions. Reworking of the model and the planner led them to be removed, as they were made unnecessary. But it is likely that in future versions as the benchmark tasks become more challenging, they will become implemented again.

Appendix C

Computational complexity and scaling

There are two main factors to consider when analyzing how well BECCA scales up to large problems. The first is the computational demands of a single time step. The second is how many time steps will be required to achieve desired performance on a given task. These are addressed separately.

C.1 Computational complexity per time step

The per-time-step complexity characterizes how much computational effort BECCA requires to update its internal state with each new set of inputs and to select an action to pass back to the world.

C.1.1 Feature extractor

In the feature extractor, computational complexity is dominated by the estimation of co-activity. This requires the creation and manipulation of several two-

dimensional arrays that can be as large as $N + M \times N + M$, where N is the number of sensor inputs and M is the maximum number of features. This results in an $\mathcal{O}((N + M)^2)$ complexity in the feature extractor. In practice the actual number of features created may be much less than M, but M provides an upper bound.

Low-order polynomial complexity is probably the best that can be hoped for in feature extraction without introducing a great deal of domain knowledge. Comparing every member in a set of inputs against every other member is an inherently polynomial operation. Polynomial complexity is typical of other feature extraction methods.

C.1.2 Reinforcement learner

In the reinforcement learner, computational complexity is dominated by comparison of the current state with the model. Specifically, it involves operations on an array that is up to $M \times L$, where M is the maximum number of features and L is the maximum number of transitions in the model. This results in an $\mathcal{O}(ML)$ complexity in the feature extractor. In practice the actual number of features created may be much less than M and the number of transitions created may be much less than L, but these provide upper bounds.

The complexity of computation in the reinforcement learner is directly related to the nature of the model it uses. By representing the value function for the task across a high-dimensional, real-valued state-action space using specific high-order transitions, the model makes a trade-off between richness of representation and the size of that representation. By making assumptions about the functional relationship between value and state variables, the model might be parameterized with just $\mathcal{O}(M)$ parameters. But this would involve the incorporation of a great deal of domain knowledge and the acceptance of assumptions that greatly limit the generality of the approach. On the other hand, explicitly expanding the high-order representation of the transitions would increase the complexity to $\mathcal{O}(M^k L)$, where k is the highest order allowed. This would greatly increase the computational requirements. The selected approach maintains a rich representation, but without creating an infeasibly large model.

C.2 Number of time steps required

The number of time steps required for BECCA to begin performing well on any task is of course highly task dependent. The two main components to this (which are conducted simultaneously) are learning a feature representation and learning a model. In order to perform a task well, the feature representation must be sufficiently sophisticated and the model must be sufficiently detailed and accurate. There are as yet no performance guarantees for any portion of BECCA, so what follows is based largely on experience gained during its development.

C.2.1 Feature extraction

The number of time steps required to create new features is determined entirely by the co-activity estimate. The estimate is calculated incrementally, as detailed in Equation A.7. There are several factors in feature extraction rate.

- The value of the constant in the update equation greatly influences how features are created. If it is too high, the co-activity estimate becomes noisy and less useful features are created. If the constant is too low, features take too long to create.
- The co-activity also depends on the nature of the inputs. If they are highly mutually predictive (e.g. redundant) then their co-activity estimate will increase rapidly. If they are related, but more loosely so, then the co-activity estimate will require more time steps to rise to the point where it will spawn feature extraction.
- The frequency with which inputs appear affects how quickly they will be used to create a feature. If two inputs are completely co-active, but appear only rarely, it will take many time steps for the estimate of their co-activity to be incremented to a significant level.

After a feature is created, it begins to establish a history of co-activity with all the inputs and other created features as well. This results in second- and later-generation features. The number of generations necessary to create features that

are sufficiently sophisticated for the agent to model and succeed at a task is completely task-dependent.

In practice, first generation visual features in the `watch` world begin to be created after a few thousand time steps, and second generation features begin to be created after about ten thousand time steps. Performance on the `image_1D` and `image_2D` worlds begins to increase with the extraction of the first features, and plateaus after the second generation features have nearly all been created, at a few tens of thousands of time steps.

C.2.2 Model learning

The model can only be as good as the features upon which it is based. So while the model is being created with every time step, if the existence of a certain feature set is critical to learning a task, it cannot begin to be useful until after that feature set is created. Following this reasoning, the number of time steps required to learn to perform well on a task is most closely related to the *sum* of the time step requirements for the feature extractor and the model, even though they are both learning at the same time.

The learning time for just the model is best understood by looking at the case where the feature extractor is inactive. The model learns based on the primitive feature inputs only. The learning time is highly dependent on the particular set of features used and on the inherent difficulty of the task. In the `grid_2D` task with 50 discrete states and 9 discrete actions, this time has been shown to be roughly 2000 time steps, with about 10,000 needed to plateau at its best behavior. This is consistent with a few visits, on average, to each position in the state-action space.

C.3 How well will it scale?

The computational complexity per time step is not negligible, but is also not prohibitively high. Computation costs being what they currently are, the $\mathcal{O}((N + M)^2)$ and $\mathcal{O}(ML)$ calculations are quite feasible to do, for even moderately large

values of L, M, and N. And truly large values could be handled at using more exotic architectures with commensurate additional expense.

In fact, if using BECCA with a physical robot, the length of a time step is much more likely to be limited by whatever physical action the robot takes. This leaves the number of time steps to consider.

The time required to create features does not show a dependence on the number of inputs. This is a reflection of the fact that the the feature extraction process is parallel in its structure. The co-activity between any two inputs is calculated in the same way, regardless of how many other inputs are simultaneously present. This supports cautious extrapolation that the feature extraction process is not a function of the number of inputs. Increasing the number and types of sensors in unlikely to change the rate at which features are created. How the number of time steps required to create features will extrapolate to more challenging tasks is still a matter of some speculation and is likely to be highly task dependent.

The reinforcement learner's rate of learning puts BECCA on par with some of the faster reinforcement learning algorithms, including those that use experience playback to help their value functions converge more quickly. Yet these methods still suffer from the criticism that they take too long to learn to apply to many practical problems. There are several reasons to suspect that BECCA may be able to overcome these limitations.

1. **Attention** Focusing attention on a few features at a time allows an agent to simplify its environment. When performed effectively, filtering experience through attention essentially reduces the dimensionality of the agent's state down to the minimum required to execute a task. BECCA's attention mechanism was intended to do exactly this.

2. **State generalization** BECCA's action selection heuristic makes use a type of generalization when it looks for contexts that are similar to the current context, rather than restricting itself to exact matches. This allows it to make use of experience that may be loosely relevant or somewhat erroneous, but still a big improvement over knowing nothing at all. Then, as BECCA gathers more experience and populates its model more densely, it can rely on increasingly relevant experience as it chooses actions.

3. **Temporal generalization** Transitions' representation of contexts and effects, using temporally collapsed state sequences allows for some temporal generalization. Features that are observed in opposite order or with a slight time delay are still recognized as being similar, if not exact matches. This helps to generalize the complex temporal information contained in the transitions.

4. **Multiple resolution representation** Sensors that represent information, such as pixel values or a joint's position, at different resolutions can speed up the learning process as well. Using the coarse information, it may be possible to learn to perform a task quickly and robustly, but poorly. However, the ability to perform poorly provides a great head start to learning using the finer-resolution sensors, with the result that the agent learns to perform the task well more quickly than it would without the coarse sensors. Having multiple sets of sensors at different resolutions has a strong biological motivation.

5. **Feature learning** For many learning tasks, once an appropriate feature representation is in place, the task becomes trivial. BECCA has the advantage over traditional reinforcement learning techniques that it is paired with a feature extractor that is always trying to represent the structure of its environment in more sophisticated ways. As the reinforcement learner spends time learning a task, it is being supplied with ever higher-level representations of its world.

Based only on BECCA's demonstrated performance and these considerations, it is too early to be certain how many time steps will be required to learn more complex tasks, but there is reason for cautious optimism.

Appendix D

Related Work

NOTE: This chapter is incomplete as it stands. Consider its deficiencies and inaccuracies as an invitation to contribute to it.

Most machine learning techniques can be broadly classified as either supervised, unsupervised, or reinforcement learning. Supervised learning techniques use data, labeled with either a category or a value, to infer the category or value of unlabeled data types. Typically they learn on a human-labeled training data set, then process an unlabeled data set that is much larger. Facebook's face labeling system is a popular example of a supervised learner.

D.1 Unsupervised Learning

Unsupervised learning is primarily concerned with clustering data. Unlabeled data points are grouped based on a given distance metric or similarity measure. Centroids of the resulting clusters can be used to represent the various classes of observations. Some unsupervised learning algorithms learn these centroids directly. The centroids can be used as features by which to classify future observations. Common methods for unsupervised learning include principal components analysis (PCA), k-means, vector quantization, and perceptrons and other

neural networks. (Latent elements in a multilayer perceptron represent cluster centroids.) Besides clustering, other terms for unsupervised learning can include compression, basis discovery, and dimensionality reduction.

Unsupervised learners may be create either a fixed number of clusters or a hierarchical tree in which the number of clusters is determined by selecting a cut point. There is a second sense in which an unsupervised learning method may be hierarchical. Some methods not only perform clustering on observations, but also on the clusters themselves. By iteratively creating clusters in this way, a hierarchy of clusters results. BECCA's feature extractor is hierarchical in this second sense.

BECCA's feature extractor is unique in that it is the only hierarchical unsupervised learning method that does not specify the number of clusters or the number of levels in the hierarchy. In deep neural networks, the number of layers and number of elements per layer can effect the performance of the feature extractor significantly. [19] The ability to generate feature hierarchies automatically removes once source of human engineering when applying a feature extractor to novel problems, and increases BECCA's generality.

The problem of hierachical feature extraction is closely related to **deep learning**. [4] Deep learning approaches seek to discover and exploit the underlying structure of a world by creating higher level, lower-dimensional representations of the system's input space. Deep learning algorithms include Convolutional Neural Networks (CNN) [14], Deep Belief Networks (DBN) [12, 8], Hierarchical Temporal Memory (HTM) [11], and the Deep SpatioTemporal Inference Network (DeSTIN) [3]. Deep learning algorithms such as these are alternative approaches, worthy of consideration for automatic concept acquisition, although they differ somewhat from BECCA's feature extractor. CNNs are designed to work with two-dimensional data, such as images, and they do not apply to arbitrarily structured data, as BECCA does. By using several layers of Restricted Boltzmann Machines, DBNs are capable of generating sophisticated features that allow it to interpret novel inputs. However, they are typically applied to the supervised learning problem of discrimination, and require a substantial amount of labeled data in order to be adequately trained. Whether DBNs can be applied to the unsupervised learning problem of feature extraction is unclear. HTM has been described conceptually and in pseudocode, but no results have been published and the method has not been subjected to peer review. However, it is of interest in that it promises to create hierarchical feature sets that incorporate the ability to store and predict

temporal sequences of activity, combining BECCA's feature extraction and model extraction functions into one element. DeSTIN incorporates both unsupervised and supervised learning methods and appears to be fully capable of hierarchical feature extraction. It has been published only recently; future papers describing its operation and performance will allow a more detailed comparison with BECCA's feature extractor.

D.2 Reinforcement Learning

Reinforcement learning (RL) is focused on the problem of choosing actions in order to maximize a reward. The canonical formulation of RL is given in [22]. Because it is focused on action selection and because it requires no training information other than reward, RL is well suited to autonomous agents.

A reinforcement learner may be model-based or model-free. In the model-free case, the learner chooses actions based only on the value of its current state. In the model-based learners, some explicit representation of the agent's history is retained, allowing the learner to plan. In this way, a model is bootstrapped by the learner during its lifetime. BECCA's reinforcement learner is an example of model-based RL. If implemented appropriately, model-based RLs can adapt to changes in the reward function more quickly than model-free RLs, which must learn not only a new reward function, but values it has already associated with other states and actions. BECCA's RL retains its model information and learns only the changing reward function.

There are several well known methods for solving the reinforcement learning problem, that is, given state inputs and a reward at each timestep, choose actions that maximize the future reward. Some examples that have been applied in agents include Q-learning [24], the Dyna architecture [21], Associative Memory [16], and neural-network-based techniques including Brain-Based Devices [18] and CMAC [2]. BECCA's reinforcement learner is another such algorithm. It is *online* and *model-based*, meaning that as it accumulates experience it creates and refines an internal model of itself and its environment. It differs from most previous methods in two ways. First, its internal model is not a first order Markov model. Instead, by using cause-effect transition pairs in which the cause is a com-

pressed version of the agent's recent state history, it creates a compressed higher order Markov model. This potentially allows BECCA to learn more sophisticated state dynamics and to record distinct sequences more naturally. Second, BECCA's reinforcement learning algorithm can handle a growing state space. This is necessary because it must work in tandem with BECCA's feature extractor, which continues to identify new features throughout the life of the agent.

BECCA's feature extractor and reinforcement learner are *incremental*, meaning that they can be efficiently updated with single observations, and *on-line*, meaning that this update can take place quickly enough to happen in real time during the agent's operation.

One challenging RL problem is the general Partially Observable Markov Decision Process (POMDP). Exact solutions to general POMDPs are computationally intractable, but some approximate solutions have been put forward. As challenging as they are, POMDPs are a subset of the Natural World Interaction problem. POMDPs are assumed to contain a stationary Markov Decision Process, whereas NWI problems may be functions of time.

D.3 Unsupervised Feature Learning with Reinforcement Learning

There are several recent examples of RL combined with unsupervised feature learning. A brief description of each describes the feature learning algorithm and the RL algorithm used and a list of the tasks to which it has been applied. A list of key differences between each approach and BECCA is included as well.

D.3.1 VQ-SNES

A team based at IDSIA use online vector quantization (VQ) as an unsupervised learner and Separable Natural Evolution Strategies (SNES), an evolutionary recurrent neural network RL, to learn a vision-based variation of the Mountain Car

task. [7] The RL was rewarded not only for good performance on the task, but also for high reconstruction error. This drove it to search for novel visual inputs, as well as to complete the task quickly.

Some differences between BECCA and VQ-SNES:

- VQ-SNES training occurs in batches, corresponding to evolutionary generations, rather than on-line.
- VQ fails in the presence of background noise. BECCA's feature extractor is designed to be robust to background noise.
- VQ does not create a hierarchical feature set.
- SNES is model-free.

D.3.2 DBN-NFQ

A University of Arizona team is using Deep Belief Networks (DBN) to initialize Neural-Fitted Q-learning (NFQ). [1] They demonstrate their work on Mountain Car and on a two-dimensional grid world with obstacles of their creation, Puddle World. NFQ uses a multilayer perceptron to perform function approximation when learning the value function, Q. DBNs create a multilayer Restricted Boltzmann Machine directly from observing data. The U of A team initialized a DBN in an unsupervised manner, and then used that as the initial multilayer perceptron for learning the value function in NFQ.

Some differences between BECCA and DBN-NFQ:

- In order to produce good results, the data used to train DBNs is often selected by the programmer, incorporating significant knowledge of the problem domain. Care must be taken to avoid over- under-sampling.
- Training takes place in a distinct epoch, separately from behavior learning and performance. As a result, feature learning ceases and the number of

features used to represent the world becomes fixed. This is not the case in BECCA.

- NFQ learns a value function. If the underlying reward function changes, the entire value function must be re-learned. BECCA avoids this by learning a state transition model separately from a reward function.

D.3.3 SFA-NC

An inter-institute team from Graz University and Humbolt University combined hierarchical Slow Feature Analysis (SFA) and two neural circuit (NC) implementations of RL methods. [15] The most interesting aspect of the work was the application of hierarchical SFA as an unsupervised method. SFA searches for slowly varying aspects of the input data and extracts those as features. In hierarchical SFA, a network of nodes process data in multiple levels, with the features extracted in one level serving as the inputs to the next. On the RL side, neural circuit implementations of both Q-learning and a policy gradient method were used. The authors demonstrated the method on a Morris water maze-type task and a vision-based two-dimensional navigation task involving a ball, a cross, and a fish.

Some differences between BECCA and SFA-NC:

- SFA, as implemented by the authors, has a fixed number of nodes and a fixed number of hierarchy layers. The network structure is fixed at the time of creation. It's design reflects a great deal of knowledge about the nature of the sensory information being processed. This is in contrast to BECCA's feature extractor, whose grouping of inputs and hierarchical structure is driven by the data observed.
- SFA undergoes a training period separate from the performance period. Training inputs are user-crafted and incorporate knowledge of the nature of the sensors.
- The NC methods employed cannot handle a changing number of inputs.

D.3.4 ISQL-MDQL

Discretization is the process of converting continuous state spaces into discrete ones, or of converting discrete state spaces in to more coarsely discretized spaces. It may be considered a degenerate case of feature extraction, although it is weaker than other feature extraction methods in that it cannot reduce the dimensionality of the learning problem. For certain problems, appropriate discretization can greatly decrease computational requirements. However, the problems on which this is most commonly demonstrated involve planar navigation, an task set that can easily be expressed using a low-dimensional state space.

A team at Universidad Carlos III de Madrid presented a methods of automatic state space discretization, iterative smooth Q-learning (ISQL), paired with a reinforcement learning algorithm called multiple discretization Q-learning (MDQL). ISQL is interesting as a discretization method in that it learns a discretization based on the performance of a preliminary RL algorithm. It can sole RL problems by itself, but when its discretization is output to for use by MDQL, the learning rate increases. Together, ISQL and MDQL are called Two Steps RL. [9] Two Steps RL has been demonstrated on a continuous grid world problem resembling an office building, on Mountain Car, and on Acrobot.

Some differences between BECCA and ISQL-MDQL:

- Both ISQL and MDQL rely on variants of Q-learning, a model-free learning method that relies on value approximation to learn large state spaces.
- ISQL requires a separate training period before its state space discretization can be handed over to MDQL.
- Due to the face that ISQL performs discretization only (a non-hierarchical type feature extraction) BECCA has potentially a much richer representational capacity.
- It is unclear how well ISQL-MDQL generalizes to higher-dimensional data. It has only been demonstrated on low-dimensional test problems.

D.3.5 MLP-FQI

A University of Freiberg team paired a multi-layer perceptron (MLP) with Fitted Q-Iteration (FQI). [13] MLPs are classic neural networks that have been shown to perform well in generating low-dimensional representations (commpressions) of high-dimensional data. FQI is a batch RL method that calculates a Q-function (state-action value function) based on a collection of state-action-reward-state transitions. In their implementation, the authors maintain a history of all such transitions so that MLP can periodically re-compute its high-level features and FQI can generate an ever-better-informed Q-function. MLP-FQI was demonstrated in a discrete gridworld task using real digital images.

Some differences between BECCA and MLP-FQI:

- Training of the MLP and FQI both are in batch mode. They don't incrementally update in an inexpensive manner.
- MLP have a fixed number of layers and elements that is set at their creation.

D.3.6 sRAAM-SARSA

The IDSIA team presents another unsupervised-reinforcement learning algorithm pair, [10] this one consisting of Sequential Recurrent Auto-Associative Memory (sRAAM) and SARSA(λ), described in [22]. sRAAM is a recurrent neural network that functions as an autoencoder capable of forming features that have not only spatial, but also temporal extent. It not only represents patterns in its inputs, but also patterns in how they change over time. It has been demonstrated solving high-dimensional, point-of-view vision-based maze navigation problems and problems that require learning long time delays. The algorithm incorporates a simple discretization (vector quantization) step to convert the output from sRAAM from a continuous to a discrete state space, so that SARSA, which is a tabular (table-based) RL algorithm could be used with it. The authors mention a lack of stability in the algorithm in the last paragraph of the paper, but do not elaborate.

Some differences between BECCA and sRAAM-SARSA:

- sRAAM is described as a Sequential Constant-Size Compressor. It has a fixed number of features, and as implemented, the vector quantization step creates a fixed number of feature space regions.
- sRAAM and SARSA(λ) are batch algorithms. They do not gracefully update after each observation. They are bath-updated in alternating chunks. This precludes one-shot learning.

D.4 Feature Extraction with Robots

Although not implemented with RL algorithms, there are several feature extraction algorithms that have been implemented on robots, physically embodied systems with sensors that respond to changes in the physical world.

D.4.1 HSSH

Automatic mapping of an environment can support model-based RL. It is a form of feature extraction. The Hybrid Spatial Semantic Hierarchy [20, 5] work, performed at the University of Texas at Austin, is one such mapping methodology. It stitches local sensor snapshots into a global map topology. It is particulary notable for the fact that it makes few assumptions about its sensors and environment. This allows it to build a model of its own sensor arrangement from scratch, then use that model to take low-level sensor snapshots of its environment. It also incorporates the learned interaction between its motor behaviors changes in sensory information to make local maps of its environments. As published, however, it is not a complete FC-RL solution in that it has no reward signal it is trying to maximize. However, such a signal could very conceivably be implemented on top of it.

D.4.2 DTW

A team at the University of Massachusetts at Amherst used Dynamic Time Warping (DTW) to aid in clustering temporal snippets of data from a mobile robot to create features that proved useful in navigation. The prototypes calculated from these clusters were surprisingly similar to features identified by human annotators. Clustering time series is not new, but the distance measure used was novel. By warping the sensor returns in time to align them, the most important aspects of the sensor information was preserved by making it invariant to variations in the speed with which a maneuver was executed.

D.5 Reinforcement Learning with Robots

Reinforcement learning is appropriate for learning in robots because it has an active component. An RL agent chooses actions, and so can direct its experience and indirectly influence its learning. This can, for instance, help solve the problem of grounding (providing a semantic interpretation for) ungrounded sensor inputs. [6]

A team centered at the *University of Alberta* has implemented *Horde* [23], an RL algorithm, on the Critterbot, a comma-shaped three degree of freedom mobile robot with a rich sensor suite. Horde is actually a collection of many RL demons, each performing GQ(λ) [17], an off-policy gradient-descent temporal difference algorithm. They create features using tile coding. Each demon in Horde captures some aspect of semantic information about the agent's interaction with its environment. As a result, Horde is capable of representing very general classes of sensorimotor information. One apparent shortcoming of Horde is that the parameters of the demons must still be set in some way. It is unclear if this can be done in an automated manner or whether significant engineering and domain knowledge is required. A second shortcoming is that, rich though it may be, the knowledge that Horde can capture is limited by its demons. If no demons are created after the initiation of the system, its knowledge will be limited.

Appendix E

Biological inspiration

The problem of processing and responding to a continuous high-bandwidth stream of sensor data in a complex environment has been solved by biology. The prioritizing and searching of large amounts of tactile, auditory, and visual data are performed by humans and animals all the time. This has motivated the mining of neurological structures and psychological processes (to the extent that they are understood) for hints on how to address these problems. BECCA is not intended to be a model of how a biological brain operates, but it does rely heavily on plausibly brain-like processing to achieve its goals.

1. Co-activity based grouping is an expression of the Hebbian rule of thumb: "Neurons that fire together wire together." Neurons that are active simultaneously or in quick succession tend to form interconnected structures. The nature of these structures can depend critically on the relative timing of their firings, as well as how often they occur. Studies of prenatal chickens suggest that spontaneous waves of retinal activity are responsible for organizing axons in the optical pathway. As nerves grow from the retina toward the cortex, patterns of co-activity may draw the axons of nearby receptors to grow together and enable them to self-organize retinotopically.

2. *Hierarchy of features* is an often-hypothesized organization principle for mental representations. For example, the human visual system appears to contain an extensive hierarchy of visual features. To the extent that research

has been able to determine, cortical cells in area V1 represent oriented line segments, those in V2 represent combinations of oriented lines in more complex features, and other areas (V3, V4, the mediotemporal area, etc.) represent yet more complex features.

3. *Lateral inhibition* is an observed behavior of neurological circuits in the cerebral cortex, resulting in winner-take-all-like function. The competition between features that share inputs is an example of behavior that might result from lateral inhibition.

4. *Source blindness* refers to the fact that whether a neural signal originates at a cone in the eye or a pain receptor in the toe, it has the same form by the time it reaches the brain: a pattern of spiking activity at a synapse. BECCA's sensor blindness was inspired by this. It has far-reaching implications. In other feature extraction methods, knowledge of the sensor type allows a certain amount of hand-crafting of first level features, such as oriented Gabor filters in the case of vision. By not making use of this information, BECCA is forced to derive even those first level features directly from data. However, by not relying on knowledge of the sensor, BECCA is not limited by developers' insights or the current state of neurophysiological knowledge.

5. *Cerebro-cortical structure* is surprisingly uniform. This has led some to speculate about a "common cortical algorithm", some recurring function that all areas of the cortex perform, despite the fact that fMRI studies associate them with a vast spectrum of perceptual, behavioral, and cognitive functions. One intriguing possibility is that the cortex serves to represent features, from the lowest level up to the most abstract, that humans use to process and respond to their worlds (leaving planning and reward assignment to subcortical structures, including the basal ganglia and amygdala). If this were shown to be the case, BECCA's feature extractor would be one candidate for describing the operation of the cortex. It performs the same set of operations on all data, regardless of its source, and builds an arbitrarily rich hierarchy of features. This line of speculation prompted the feature extractor's development.

Appendix F

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 G

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 papers at ICRA and IROS.
Aleksandra Faust	2012	For searching for a target location in a simulated world using both visual and auditory data and for publishing the results in a paper at IROS.
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 H

Revision history

Version 0.4.3, November 9, 2012

- **Major change** Groups are eliminated. Features are created directly now using the old group-creation algorithm. This in turn spawned a large number of minor changes to support it. The end result is that features in the higher-levels of the hierarchy tend to be created more gracefully, make more sense to a human user, and naturally terminate. The number of user-selected parameters in BECCA also went down a bit.
- The maximum possible size for large variables (the co-activity estimate and the model) is allocated on startup. This helps prevent running out of memory mid-run.
- The `learner` was renamed `actor` to more accurately reflect its function and to provide symmetry with `perceiver` in the sense of “perception-action loops".

Version 0.4.2, September 28, 2012

- A chapter describing the operation of the reinforcement learner has been added to the Users Guide.
- The image-based tasks all implement center-surround filtering as a preprocessing step.
- Perceiver parameters have been adjusted to produce more intuitive features in the image-based tasks.
- Features in the image-based tasks are displayed as receptive fields, resulting in a higher-contrast format.
- Mutual co-activity has been modified to be the minimum, rather than the product, of two co-activities.

Version 0.4.1, August 17, 2012

- Nearly all numpy 1D arrays were converted to 2D arrays for consistency
- Co-activity has been modified to be symmetric.
- Model transitions' effects are updated with each observation or use. That is, they are updated both when they are observed and when they are used by planner as a basis for selecting an action.
- Model transitions now include an effect uncertainty and a reward uncertainty.
- All the features in a group are now created when the group is created. They evolve over time based on the inputs observed.
- Feature activity is driven both by excitation and fatigue.
- Feature evolution is driven both by activity and inhibition.

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 I

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 J

Citing this guide

BECCA is an acronym. When you refer to it, you can write it in all caps like this: BECCA. But that always feels like I'm shouting at someone, so I write it with small caps. In LaTeX, that looks like `\textsc{Becca}`

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] F. Abtahi and I. Fasel. Deep belief nets as function approximators for reinforcement learning. In *IEEE International Conference on Development and Learning and Epigenetic Robotics*, Frankfurt, Germany, 2011.

[2] J. Albus. A new approach to manipulator control: Cerebellar model articulation controller (CMAC). *Journal of Dynamic Systems, Measurement and Control*, 97:220–227, 1975.

[3] I. Arel, D. Rose, and B. Coop. DeSTIN: A deep learning architecture with application to high-dimensional robust pattern recognition. In *Proc. AAAI Workshop Biologically Inspired Cognitive Architectures (BICA)*, 2009.

[4] I. Arel, D. C. Rose, and T. P. Karnowski. Deep machine learning—a new frontier in artificial intelligence research. *IEEE Computational Intelligence Magazine*, November 2010.

[5] P. Beeson, J. Modayil, and B. Kuipers. Factoring the mapping problem: Mobile robot map-building in the hybrid spatial semantic hierarchy. *The International Journal of Robotics Research*, 29(4):428–459, 2010.

[6] Y. Choe, H.-F. Yang, and D. C. Eng. Autonomous learning of the semantics of internal sensory states based on motor exploration. *International Journal of Humanoid Robotics*, 4:211–243, 2007.

[7] G. Cuccu, M. Luciw, J. Schmidhuber, and F. Gomez. Intrinsically motivated neuroevolution for vision-based reinforcement learning. In *International Conference on Development and Learning and Epigenetic Robotics*, 2011.

[8] I. Fasel and J. Berry. Deep belief networks for real-time extraction of tongue contours from ultrasound during speech. In *Proceedings of the International Conference on Pattern Recognition (ICPR)*, 2010.

[9] F. Fernández and D. Borrajo. Two steps reinforcement learning. *International Journal of Intelligent Systems*, 23(2):213–245, 2008.

[10] L. Gisslen, M. Luciw, V. Graziano, and J. Schmidhuber. Sequential constant size compressors for reinforcement learning. In *Proceedings of the Fourth Conference on Artificial General Intelligence*, 2011.

[11] J. Hawkins, S. Ahmad, and D. Dubinsky. Hierarchical temporal memory. Technical report, Numenta, Sep 12 2011. version 0.2.1.

[12] G. E. Hinton, S. Osindero, and Y. Teh. A fast learning algorithm for deep belief nets. *Neural Computation*, 18:1527–1554, 2006.

[13] S. Lange and M. Reidmiller. Deep auto-encoder neural networks in reinforcement learning. In *Proc International Joint Conference on Neural Networks*, 2010.

[14] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proc. IEEE*, 86(11):2278–2324, 1998.

[15] R. Legenstein and N. Wilbert amd L. Wiskott. Reinforcement learning on slow features of high-dimensional input streams. *PLoS Computational Biology*, 6(8):e1000894, 2010.

[16] S. E. Levinson. *Mathematical Models for Speech Technology*. John Wiley and Sons, Chichester, England, 2005. pp. 238-239.

[17] H. R. Maei and R. S. Sutton. Gq(lamda): A general gardient algorithm for temporal-difference prediction learning with eligibility traces. In *Proceedings of the International Conference on Artificial General Intelligence*, Lugano, Switzerland, 2010.

[18] J. L. McKinstry, G. M. Edelman, and J. L. Krichmar. A cerebellar model for predicitive motor control tested in a brain-based device. *Proceedings of the National Academy of Sciences*, 103(9):3387–3392, 2006.

[19] G. Montavon, M. L. Braun, and K.-R. Müller. Kernel analysis of deep networks. *Journal of Machine Learning Research*, 12:2563–2581, 2011.

[20] D. Pierce and B. J. Kuipers. Map learning with uninterpretet sensors and effectors. *Artificial Intelligence*, 92:169–227, 1997.

[21] R. S. Sutton. Planning by incremental dynamic programming. In *Proceedings of the Eighth International Workshop on Machine Learning*, pages 353–357. Morgan Kaufmann, 1991.

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

[23] R. S. Sutton, J. Modayil, M. Delp, T. Degris, P. M. Pilarski, A. White, and D. Precup. Horde: A scalable real-time architecture for learning knowledge from unsupervised sensorimotor interaction. In *Proceedings of the 10th International Conference on Autonomous Agents and Multiagent Systems*, Taipei, Taiwan, 2011.

[24] C. J. C. H. Watkins and P. Dayan. Technical note: Q-learning. *Machine Learning*, 8(3-4):279–292, May 1992.