

BECCA version 0.4.1

User's Guide

Brandon Rohrer

October 32, 2012



Contents

1	Get and run BECCA	5
1.1	Where do I get the code?	5
1.2	What tools do I need to run it?	6
1.3	How do I run it?	6
2	Write and run your first world	8
2.1	What is a world?	8
2.2	How do I make a <i>hello</i> world?	9
2.3	What do I need to implement in my world?	10
2.3.1	num_sensors	11
2.3.2	num_primitives	11
2.3.3	num_actions	12
2.3.4	step()	12
2.3.5	is_alive()	12
2.3.6	set_agent_parameters()	12
2.3.7	is_time_to_display()	13
2.3.8	vizualize_feature_set()	13
2.4	What is base_world.py?	14
2.5	How do I run my world?	14
3	Share your world with other BECCA users	17
3.1	Where do I put my world module so that others can find it?	17
3.2	How do I tell them about it?	18
4	Modify your agent code	20
4.1	How is the agent code structured?	20
4.1.1	State	22
4.1.2	Agent	22

4.1.3	Grouper	22
4.1.4	FeatureMap	22
4.1.5	Learner	22
4.1.6	Model	23
4.1.7	Planner	23
4.1.8	Utility modules: <code>utils</code> and <code>viz_utils</code>	23
4.1.9	TCP/IP communication implementation: <code>server.py</code>	23
4.2	Is my agent better than the core agent?	24
4.2.1	<code>grid_1D.py</code>	25
4.2.2	<code>grid_1D_ms.py</code>	25
4.2.3	<code>grid_1D_noise.py</code>	25
4.2.4	<code>grid_2D.py</code>	26
4.2.5	<code>grid_2D_dc.py</code>	27
4.2.6	<code>image_1D.py</code>	27
4.2.7	<code>image_2D.py</code>	28
4.2.8	<code>watch.py</code>	28
5	Share your agent with other BECCA users	30
5.1	How do I make my code look like the rest of the core?	31
A	How BECCA's feature creator works	33
A.1	Grouping	34
A.2	Features	36
A.2.1	Feature activity	36
A.2.2	Feature evolution	38
A.2.3	Computational requirements	39
A.3	Hierarchy, time, and domain knowledge	41
A.4	Co-activity	42
B	Related Work	44
B.1	Unsupervised Learning	44
B.2	Reinforcement Learning	46
B.3	Unsupervised Feature Learning with Reinforcement Learning	47
B.3.1	VQ-SNES	47
B.3.2	DBN-NFQ	48
B.3.3	SFA-NC	49
B.3.4	ISQL-MDQL	50
B.3.5	MLP-FQI	51

<i>CONTENTS</i>	4
B.3.6 sRAAM-SARSA	51
B.4 Feature Creation with Robots	52
B.4.1 HSSH	52
B.4.2 DTW	53
B.5 Reinforcement Learning with Robots	53
C Biological inspiration	54
D Project contributions	57
E Publications and hacks	58
F Revision history	60
F.1 Version 0.4.0, June 8, 2012	60
G The small print	61
H Citing this guide	63

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 1.6.1¹
- matplotlib 1.2.

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

1.3 How do I run it?

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

The worlds in `benchmark.py` are designed to be simple tests of BECCA's fundamental learning capabilities. BECCA is an agent, in the sense that it makes decisions in order to achieve a goal, but it was created to be used in many different settings. The worlds tested in `benchmark.py` include one and two dimensional

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

²From 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.

³Any 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. [25]

The architecture is modular. You can develop and run your own worlds without having to know very much about how the BECCA agent works. The only constraints BECCA imposes on worlds are:

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

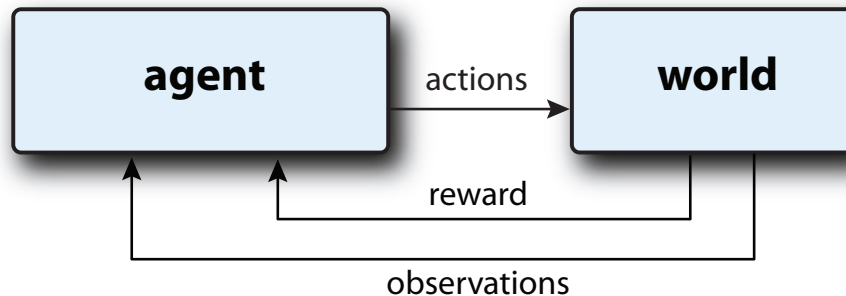


Figure 2.1: The agent-world coupling in BECCA. The agent selects actions to execute on the world, which in turn provides observations and reward feedback.

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

Strictly speaking, BECCA allows for two classes of observations, *sensors* and *primitives*. The only difference between the two is that the agent has to build sensors into features before it can use them, whereas it can use primitives as features immediately. Typically, observations that are likely to need some refining before they become useful (such as sets of pixel values) are passed in as sensors, and information that is more immediately useful (such as a contact sensor) are passed in as primitives.

2.2 How do I make a *hello* world?

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

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

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

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

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

and save the changes.

3. Add the line

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

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

4. Run `tester.py`

2.3 What do I need to implement in my world?

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

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

- `num_sensors`

- `num_primitives`
- `num_actions`

these three methods,

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

and perhaps two optional methods,

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

each of which are all described in more detail below.

2.3.1 num_sensors

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

2.3.2 num_primitives

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

2.3.3 `num_actions`

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

2.3.4 `step()`

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

2.3.5 `is_alive()`

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

2.3.6 `set_agent_parameters()`

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

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

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

2.3.7 `is_time_to_display()`

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

2.3.8 `vizualize_feature_set()`

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

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

2.4 What is `base_world.py`?

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

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

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

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

2.5 How do I run my world?

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

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

```
import numpy as np
from agent.agent import Agent
```

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

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

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

```
def main():

    world = World()

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

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

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

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

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

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

```

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

```

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

```

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

```

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

```

agent.report_performance()
agent.show_reward_history()

return

```

8. And finally, run `main()`

```

if __name__ == '__main__':
    main()

```


Chapter 3

Share your world with other BECCA users

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

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

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

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

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

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

a battery of worlds for benchmarking, and a couple of utility modules. The plan is to keep the BECCA core small and trim with only the necessities 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 guide¹
- Write a conference paper²
- 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,

¹If 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/`.

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

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

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

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

Chapter 4

Modify your agent code

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

4.1 How is the agent code structured?

I haven't finished writing up the justification and theory behind every part of the agent. This section will give just an overview of the structure so that you have some idea about where to start making changes. A more thorough description definitely belongs in this guide, and I plan to include it in later versions. For now, the best I can do is refer you to some previously published descriptions: [22, 21]. These are no longer entirely accurate, unfortunately, but give a pretty good ideal about what is going on.

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

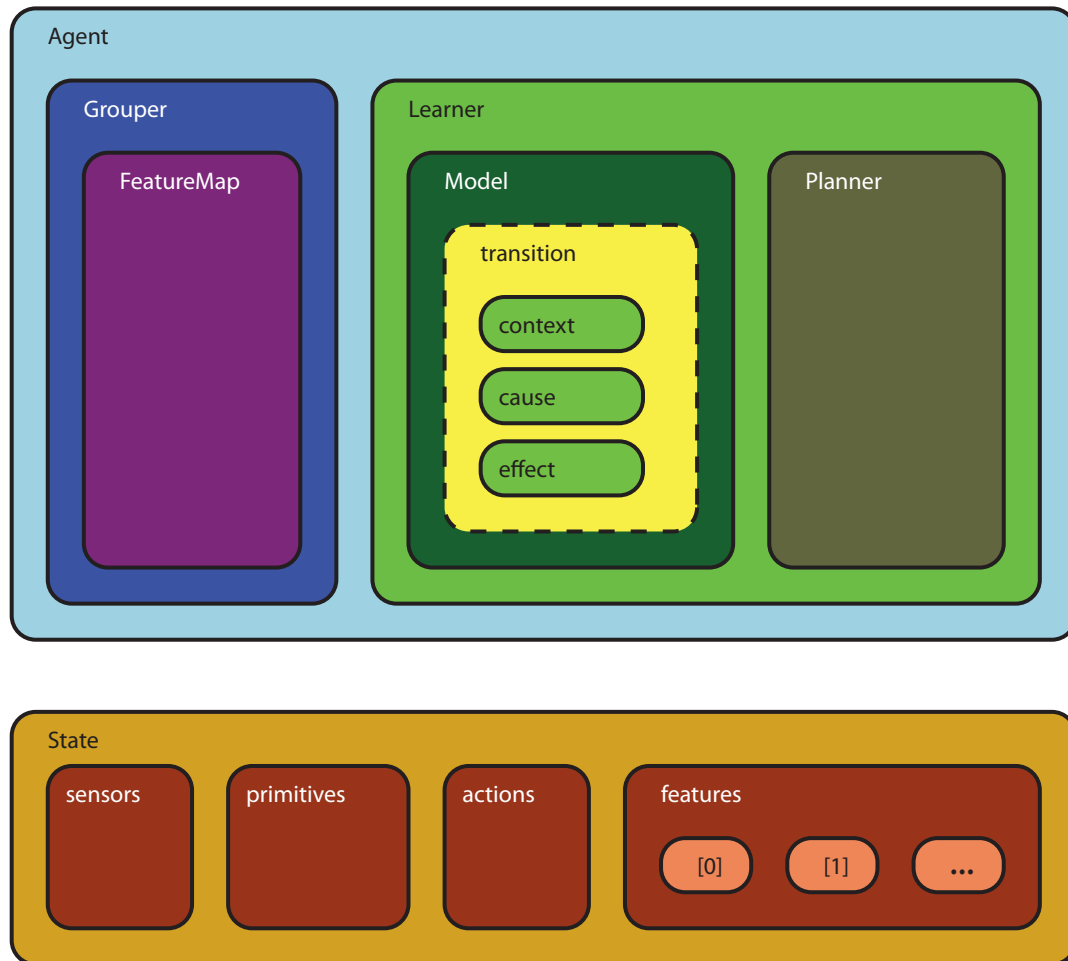


Figure 4.1: The class structure of BECCA's agent code.

4.1.1 State

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

4.1.2 Agent

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

4.1.3 Grouper

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

4.1.4 FeatureMap

- Lists the inputs that contribute to each feature.

4.1.5 Learner

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

4.1.6 Model

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

4.1.7 Planner

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

4.1.8 Utility modules: `utils` and `viz_utils`

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

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

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

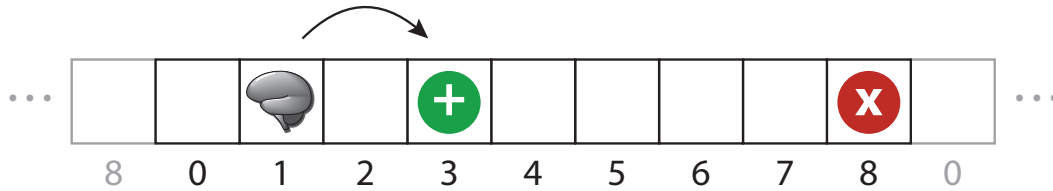


Figure 4.2: The one-dimensional grid world.

4.2 Is my agent better than the core agent?

The most natural question to ask after changing the agent is whether the new version is better than the old one.¹ The answer will of course depend on what you mean by “better”. The choice of performance measure has a great deal of influence on the performance levels of individual agents. The only necessary characteristic of a performance measure is that it produce a numerical value when applied to an agent. If you have a specific performance measure in mind, code it up and use it. If not, consider `benchmark.py`.

The worlds in `benchmark.py` are intended to be a testing battery that requires a broad learning capability to do well on. Admittedly, the battery members in this version of the benchmark are very basic. New worlds are added only when BECCA can perform well on all the old ones, each new world has brought to light more of BECCA’s bugs. This process is great for ironing out the kinks in BECCA, but a little slow. As BECCA matures, the worlds in the benchmarking battery will also grow in number and sophistication. Since the benchmark is likely to change with each release of BECCA, the version number of each benchmark will be an important identifier.

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

4.2.1 `grid_1D.py`

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

4.2.2 `grid_1D_ms.py`

The `ms` stands for ‘multi-step’. This world is identical to the `grid_1D.py` world, except that the agent may only step in increments of one in either direction. Occasionally random perturbations place the agent in new positions and it must make its way back to its goal using multiple steps, rather than in a single step. This world requires multi-step planning and is a challenge for some learning agents.

4.2.3 `grid_1D_noise.py`

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

¹This 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.

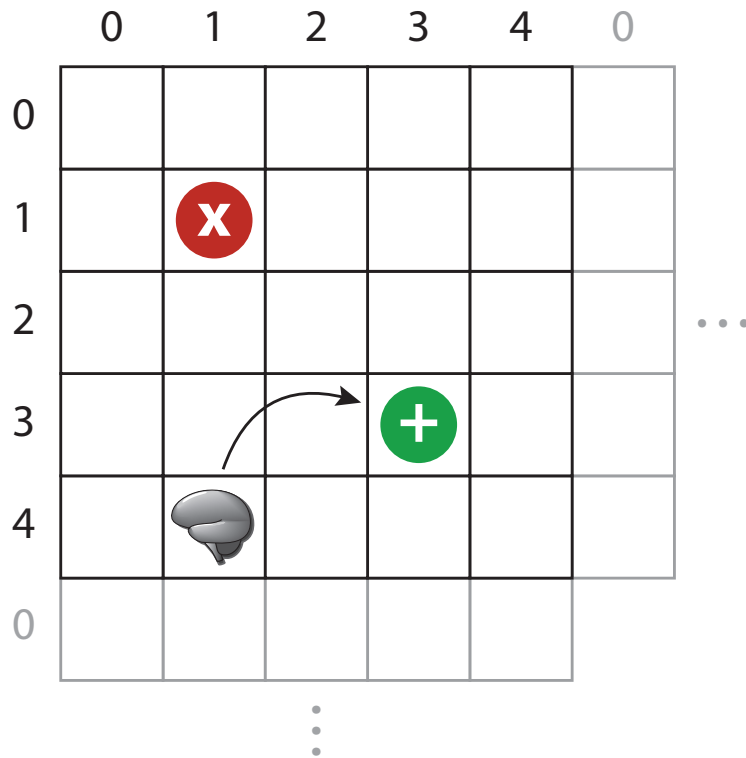


Figure 4.3: The two-dimensional grid world.

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 ($1 - \text{the values}$) are passed to the agent in a 200 element sensor array.

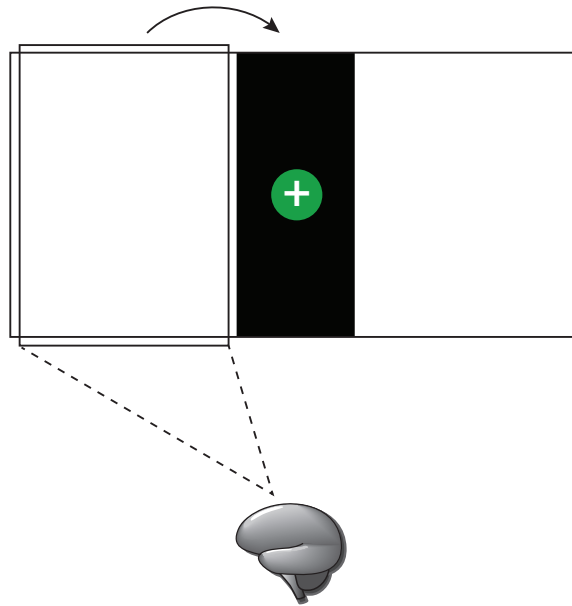


Figure 4.4: The one-dimensional image world.

4.2.7 `image_2D.py`

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

4.2.8 `watch.py`

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

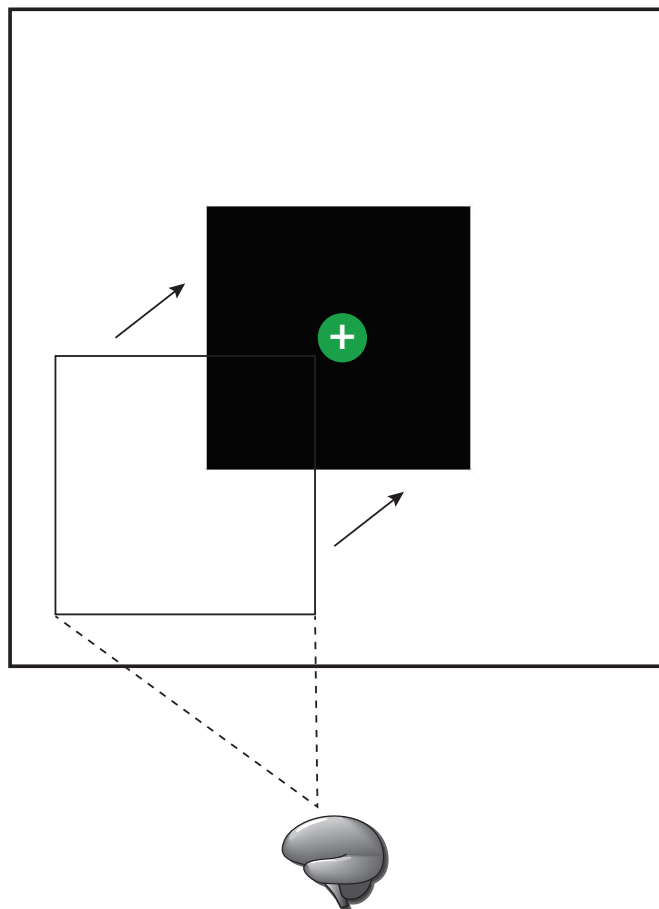


Figure 4.5: The two-dimensional image world.

Chapter 5

Share your agent with other BECCA users

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

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

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

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

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

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

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

On low level style specifics, the PEP 8 Python style guide¹ and PEP 257 Docstring style guide² 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/>

ixf

Appendix A

How BECCA's feature creator works

A paper from the 2012 AAAI Spring Symposium on intelligent robots [23] gives a full technical overview of BECCA's operation, but does not provide enough mathematical detail to fully describe its implementation. This section has those details, but only for the feature creator. A detailed mathematical description of the reinforcement learner is forthcoming

BECCA creates features by 1) forming commonly co-active inputs into groups and 2) identifying observed patterns of activity within those groups. (See Figure A.1.) For example, in prior work with image data BECCA has formed nearby pixels (which are likely to be active at the same time more often than distant pixels) into groups and has chosen features within those pixels corresponding to oriented line segments and center-surround patterns. [21] 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.

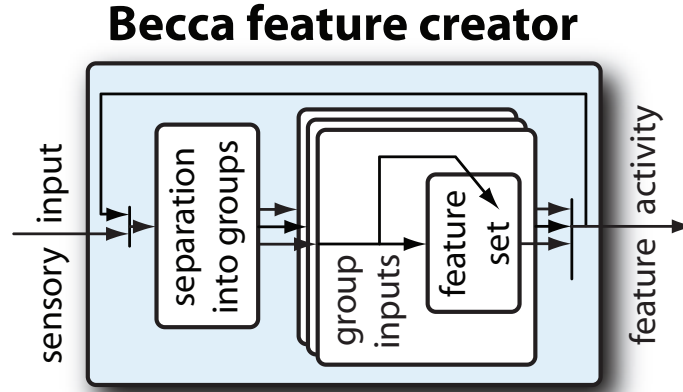


Figure A.1: Block diagram of BECCA's feature creator.

A.1 Grouping

The separation of input channels into groups is driven by how often each channel is co-active with each of the other channels. The goal of grouping is to find groups of input channels that tend to be mutually co-active. By virtue of their co-occurrence, activity in these channels is assumed to be not entirely independent.

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-inactivity. 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 multiplying them together:

$$\kappa_M(A, B) = \kappa(A, B)\kappa(B, A) \quad (\text{A.2})$$

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, *undirected* 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 grouping 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 group. Then, the channel with the highest average mutual co-activity with existing group members, N_m , is added:

$$N_m = \operatorname{argmax}_N \frac{\kappa_M(A, N) + \kappa_M(B, N)}{2} \quad (\text{A.3})$$

This agglomeration process continues until the mutual co-activity with the next candidate does not exceed another threshold, C_2 . Each time an input channel

is added to a new group, its co-activity with all other channels is penalized by a factor of e^{-C_3} from that point forward. This serves to limit the number of groups that a single channel can join.

A.2 Features

Once a group has been formed, features can be created and evolved within it. BECCA does this by randomly selecting features to start with, then incrementally modifying the features depending upon the patterns of activation observed within the group.

A.2.1 Feature activity

To be more precise, each input channel can be interpreted as a separate dimension in an n -dimensional input space, where n is the total number of input channels. Groups then form m -dimensional subspaces of the input space, where m is the number of input channels in a given group. At any time step, the input values to a group can be represented as a single point in that group’s subspace.

Features provide a way to coarsely discretize the inputs. BECCA represents features as points on a unit sphere in a group’s subspace. That is, a feature is a vector of the group’s inputs, constrained to have a magnitude of one. In a group with m inputs, empirical results suggest that $m + 2$ is a useful number of features to create. Many more than this creates an unnecessarily large feature set, increasing learning times, and many fewer than this does not provide a sufficiently rich feature set to allow high-performance learning.

Before comparing it with features, the input is scaled:

$$\hat{s} = s \frac{\max(s)}{\|s\|} \quad (\text{A.4})$$

This scaling gives the input two desirable properties:

1. Assuming no component of the input is greater than 1, the scaled output will not be greater than 1.
2. The value of \hat{s} will only be maximal (equal to 1) if one of one of its inputs is maximal. If this were not the case, inputs of many different magnitudes could produce the same scaled input.

In order to determine whether a scaled input is similar to existing features, a similarity metric is required. BECCA uses the square of the magnitude of the projection of the scaled input onto a feature to determine its similarity. Because the feature magnitude is already known to be of magnitude 1, the similarity between input s and feature f can be calculated as follows:

$$\beta(\hat{s}, f) = (s \cdot f)^2 \quad (\text{A.5})$$

The similarity, β , of an input with each feature in a group represents the *excitation* of each feature, the extent to which that input tends to excite activity within it. The similarity is treated like a vote. The feature receiving the highest vote is activated. However, the excitation of each feature is modulated by a fatigue factor, ϕ , in the following way:

$$v = \beta e^{-C_4 \phi} \quad (\text{A.6})$$

where v is the vote of the input for the feature in question and C_4 is a user-selected constant with $0 < C_4 < 1$.

The activity of each feature in the group, a , is determined by the votes:

$$a_i = \beta_i \quad \text{for } i = \text{argmax}(v) \quad (\text{A.7})$$

$$= 0 \quad \text{for all other } i \quad (\text{A.8})$$

ϕ is an incrementally updated function of each feature's recent activity given by the following:

$$\phi_{t+1} = C_5(\phi_t + a_t) \quad (\text{A.9})$$

where C_5 is a user selected constant between 0 and 1. When a feature is active, its fatigue factor increases, diminishing its tendency to be activated in subsequent time steps. When a feature is inactive, its fatigue factor decays over time, making it increasingly likely to be activated.

By holding most feature activities to zero, the group's features provide a sparse representation of its input subspace. The feature activities from all groups are combined to form the feature creator's output.

A.2.2 Feature evolution

At each time step features are also incrementally updated. The nature of the the updates is such that features evolve over time to represent the most commonly observed patterns in a group's inputs.

The active feature in each group is updated to resemble the set of inputs that resulted in its selection. The size of the update is determined by the feature activity and inhibition, γ , that the feature receives. Inhibition in turn is determined by the excitation of nearby features, and their similarity to the winning feature:

$$\gamma = e^{\sum_{j \neq i} \beta(f_i, f_j) \beta(\hat{s}, f_j) C_6} \quad (\text{A.10})$$

where f_i is the winning (active) feature, $\beta(f_i, f_j)$ is the similarity between the winning feature and each of the others, $\beta(\hat{s}, f_j)$ is the excitation of each of the other features, and C_6 is a user-selected constant with $0 < C_6 < 1$.

The inhibition scales the update magnitude:

$$f_{i \text{ new}} = f_i + (f_i - \hat{s})a_i\gamma C_7 \quad (\text{A.11})$$

where a_i is the feature activity of the winning feature, calculated above, and C_7 is a user-selected constant with $0 < C_7 < 1$. The newly adapted feature is then renormalized to ensure that it maintains a magnitude of 1:

$$f_{i \text{ new}} = \frac{f_{i \text{ new}}}{|f_{i \text{ new}}|} \quad (\text{A.12})$$

Updating features in this way favors large updates when the winning feature is:

1. far from other features
2. far from the input that excited it
3. strongly active

Figure A.2 summarizes the life of a feature set in pictorial form.

A.2.3 Computational requirements

An important difference between BECCA and other feature creation methods is that BECCA's feature creation and feature calculation operations occur on small subspaces of the input, rather than operating on the entire input space at once, as for instance principal components analysis does. This gives BECCA the advantage of lower-order computational complexity for these operations: it is linear in the size of the input space rather than polynomial or exponential. This allows BECCA to circumvent the curse of dimensionality to some extent and bodes well for its scalability to large problems.

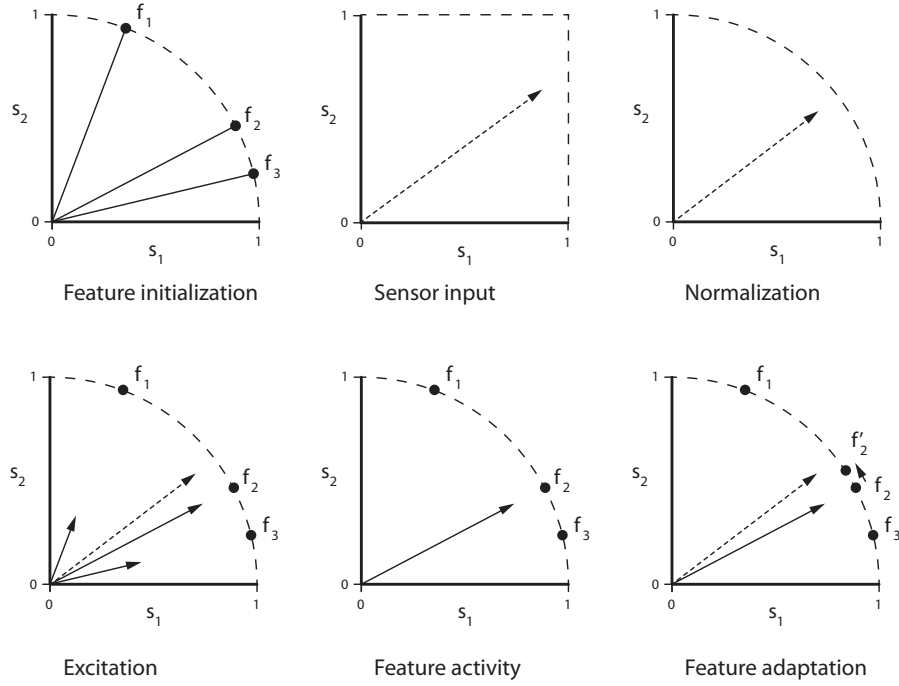


Figure A.2: The creation, operation, and evolution of features. In this example, a new group is created out of just two inputs, s_1 and s_2 . Three initial features are chosen randomly from the set of positive unit vectors in the group space. On each subsequent time step, the values of s_1 and s_2 make a sensor input vector. Each time it is normalized, to the maximum possible magnitude of an input vector in the same direction. Then, it excites each of the features, according to its magnitude and its distance from them. The most excited feature is considered active for that time step, and the others are not. Finally, the winning feature migrates slightly toward the input that excited it.

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:

$$s_{t+1} = C_8 s_t, \quad 0 < C_8 < 1 \quad (\text{A.13})$$

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 than 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 creator'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:

$$\Delta\kappa(A, B) = C_9 a(ab - \kappa(A, B)), \quad 0 < C_9 < 1 \quad (\text{A.14})$$

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.14. 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

$$\kappa(A, B) = ab \quad (\text{A.15})$$

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

$$A \mid p(a = 1) = 0.5, \quad p(a = 0) = 0.5 \quad (\text{A.16})$$

$$B \mid p(b = 1) = 0.1, \quad p(b = 0) = 0.9 \quad (\text{A.17})$$

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

$$\Delta\kappa(A, B) = b - \kappa(A, B) \quad (\text{A.18})$$

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 if 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

$$0 \leq ab \leq 1 \quad (\text{A.19})$$

$$0 \leq \kappa(A, B) \leq 1 \quad (\text{A.20})$$

Appendix B

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.

B.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 creator is hierarchical in this second sense.

BECCA’s feature creator 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 creator significantly. [19] The ability to generate feature hierarchies automatically removes once source of human engineering when applying a feature creator to novel problems, and increases BECCA’s generality.

The problem of hierachical feature creation 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 creator. 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 creation 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 creation and model creation functions into one element. DeSTIN incorporates both unsupervised and supervised learning methods and appears to be fully capable of hierarchical feature creation. It has been published only recently; future papers describing its operation and performance will allow a more detailed comparison with BECCA’s feature creator.

B.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 [25]. 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 [27], the Dyna architecture [24], 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 *on-line* 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 compressed

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 creator, which continues to identify new features throughout the life of the agent.

BECCA’s feature creator 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.

B.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.

B.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 creator is designed to be robust to background noise.
- VQ does not create a hierarchical feature set.
- SNES is model-free.

B.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.

B.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 creator, 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.

B.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 creation, although it is weaker than other feature creation 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 solve 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 fact that ISQL performs discretization only (a non-hierarchical type feature creation) 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.

B.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 (compressions) 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.

B.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 [25]. 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.

B.4 Feature Creation with Robots

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

B.4.1 HSSH

Automatic mapping of an environment can support model-based RL. It is a form of feature creation. 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 particularly 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.

B.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.

B.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* [26], 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 $GQ(\lambda)$ [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 C

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 winner-take-all nature of feature activation within a single group was based on this phenomenon.
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 creation 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 creator 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 creator's development.
6. *Grouping* is consistent with the localized information processing in the cortex. Since Mountcastle and Hubel and Wiesel's first observations of column-like structures, the notion that small discrete pieces of the cortex operate nearly independently has been alluring, but problematic to those

who have tried to explain in detail how big those pieces are and what they do. Regardless, it seems clear that the majority of connections in the cortex are very local in scope. BECCA's focus of operating on groups was inspired by this observation.

Appendix D

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	For creating and releasing the openbecca.org website.
	2012 / 01	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 E

Publications and hacks

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

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

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

Matt Chapman	2012	For integrating BECCA (0.3.11 under Octave, slightly modified) with a Lego NXT robot via ROS to run the 1D grid task.
	2012	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 physical robot systems.

Appendix F

Revision history

F.1 Version 0.4.0, June 8, 2012

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

Appendix G

The small print

This guide Copyright (c) 2012 Brandon Rohrer

BECCA 0.4.0 code Copyright (c) 2011 Sandia Corporation. Copyright (c) 2012 Brandon Rohrer, Alejandro Dubrovsky, SeH

Version 0.3.11 of this software was released under the MIT Open Source license by Sandia Corporation. Under the terms of the license, the software was ported to Python and assigned version number 0.4.1. It and subsequent versions are released under the MIT Open Source License,¹ copyright the contributing authors.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

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 H

Citing this guide

Feel free to cite this guide in any BECCA related publications or reports you write. If you're using L^AT_EX, 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 and 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(λ): A general gradient 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 predictive 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 uninterpreted sensors and effectors. *Artificial Intelligence*, 92:169–227, 1997.
- [21] B. Rohrer. Biologically inspired feature creation for multi-sensory perception. In *Second International Conference on Biologically Inspired Cognitive Architectures*, 2011.
- [22] B. Rohrer. A developmental agent for learning features, environment models, and general tasks. In *First Joint International Conference on Development and Learning and Epigenetic Robotics*, 2011.
- [23] B. Rohrer. BECCA: Reintegrating ai for natural world interaction. In *Proceedings of the AAAI Spring Symposium on Designing Intelligent Robots: Reintegrating AI*, 2012.
- [24] R. S. Sutton. Planning by incremental dynamic programming. In *Proceedings of the Eighth International Workshop on Machine Learning*, pages 353–357. Morgan Kaufmann, 1991.
- [25] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, Massachusetts, 1998.
- [26] 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.
- [27] C. J. C. H. Watkins and P. Dayan. Technical note: Q-learning. *Machine Learning*, 8(3-4):279–292, May 1992.