evolearn Documentation

Release 0.1.0

Chad Carlson

Nov 12, 2018

Contents

evolearn is a python neuroevolution framework. It allows for easier interaction with the MultiNEAT library, which is itself a set of Python bindings for the original NEAT algorithm variants written in C.

CHAPTER 1

Introduction

This provides installation instructions for the **evolearn** Python package, which is used to implement variants of the Neuroevolution of Augmenting Topologies algorithm as a method of hyper-parameter optimization for artificial neural networks.

1.1 Installation

1.1.1 Stable release

To install evolearn, run this command in your terminal:

```
$ pip install evolearn
```

This is the preferred method to install evolearn, as it will always install the most recent stable release.

If you don't have pip installed, this Python installation guide can guide you through the process.

1.1.2 From sources

The sources for evolearn can be downloaded from the Github repo.

You can either clone the public repository:

```
$ git clone git://github.com/chadwcarlson/evolearn
```

Or download the tarball:

```
$ curl  -OL https://github.com/chadwcarlson/evolearn/tarball/master
```

Once you have a copy of the source, you can install it with:

```
$ python setup.py install
```

1.2 Usage

To use evolearn in a project:

```
import evolearn
```

1.3 Overview

evolearn is a neuroevolution framework for designing artificial life experiments using variants of the HyperNEAT algorithm. Using this package enables you to add a generative encoding mechanism for hyper-parameter search to your existing reinforcement learning experiments, while also providing a simplified testbed for conducting agent-based foraging research.

Addtional information about HyperNEAT and alternative implementations can be found at http://eplex.cs.ucf.edu/neat_software/.

CHAPTER 2

Getting Started

This section provides an introduction to the **evolearn** Python package, as well as the theoretical foundations for its development and implementation.

Namely, it provides an overview of *neuroevolution*, a method of hyper-parameter optimization fo artificial neural networks, where operators for both within-generation learning and between-generation inheritance widen the search of parameter space for simulated controller tasks.

2.1 Introduction

evolearn is a neuroevolution framework for designing artificial life experiments using variants of the HyperNEAT algorithm.

This manual is intended to be an introduction to the **evolearn** package, as well as its design and contribution to the fields of neuroevolution, artificial intelligence, and theoretical neuroscience according to the research goals of the author. You will find code for interacting with the library and using it to implement standard machine learning techniques and less common neuroevolutionary methods on a variety of tasks.

Once you have made your way through the *Introduction*, you can find a collection of experiments relevant to your neuroevolution research.

2.1.1 Overview

The goal of **evolearn** parellels the ideal (though often unspoken) goals of artificial intelligence - to develop solid theory for generating intelligent models for solving any type of problem a biological nervous system could encounter. That goal is tremenously ambitious, and in a way the purpose of **evolearn** is to expose the difficulty of striving towards that objective. Furthermore, **evolearn** focuses on neuroevolutionary methods of indirect encodings, taking inspiration from the mechanisms that allowed biological organisms to discovery meaningful and effective solutions for life through evolution by natural selection.

Biological organisms are intelligent in ways we hope to replicate in machines. There is a tremendous diversity of solutions of intelligence on our planet, and a theory of nervous systems should account for that diversity. A prominent way of thinking about the existence of a particular nervous system is the observation that it exists as a solution that is a

product of both learning during the lifetime of that organism, as well as the evolutionary history of previous solutions that led to it.

It may not always be necessary to include evolutionary search in training models to solve certain classes of problems. Benchmark machine learning models in image recognition do not at this point include global search methods prior to incremental error propagation. In fact, it may be possible to find sufficiently good solutions without employing evolutionary computation at all. On the other hand, there are a few reasons for thinking that evolutionary thinking could improve the performance of some of these models, as well as the discovery of better ones.

First, it seems that the problem of escaping local minima when comparing solution performance against an objective is an inescapable problem. Local gradient-based changes to a model require specific alterations to avoid this issue. Global optimization approaches like genetic algorithms, however, can often discover multiple minima in the objective function and avoid getting stuck.

Second, because local gradient-based methods often require techniques to prevent them from getting bogged down in local minima, successfully implenting a class of models on a new dataset often requires significant expertise. If instead we can initialize simple models for a task and evolve them over time, the level of expertise needed to use them may be decreased.

Third, it is an often unspoken objective in the fields of machine learning and artificial intelligence themselves to search for models that can match and/or outperform the capabilities of biological nervous systems. While it may not be necessary to undergo billions of years of evolution to reach certain solutions in the space of all possible ones, it is a fact that in biology those solutions were reached through exactly that process. Therefore, it is at least worth our attention, when attempting to connect the performance of machine learning models to that of the organisms we study in neuroscience, that global search strategies could very well be responsible for both the level and diversity of intelligent life we see around us.

2.1.2 Getting Started

Let's say for a moment that it was our goal to find an organism that can effectively move through an evironment and eat enough food to stay alive. Within that goal there are a few important components that can get us started towards modelling an abstract biological organism.

1. *The organism must move*: we would like to create a model with outputs.
2. *The organism must consume food*: this would imply there is feedback in our environment, such as a reward for nutrients.
3. *The organism must stay alive*: this organism can learn and improve against this objective.

Since this package will completely revolve around artificial neural networks, it should come as no surprise that we will end up looking for solutions (organisms) that can be represented as a neural network. An organism can observe its environment, make observations about the food it can see, make actions in response to that input, and then receive reward feedback that can be used to alter the structure of that neural network so that it can be better in the future.

Ultimately we will be able to use **evolearn** on more complex agent-environment controllers, and even simulate controllers that can then be embodied in robotic agents in the real world - but, at least our agent can act as a starting point.

2.1.3 A Simple Controller Task

We can design our agent in an environment that resembles the popular reinforcement learning benchmark game Flappy Bird. (a single frame from this task is shown below in **Figure 1**)

An agent remains stationary in a column of a grid world, while the world continues to fly by it, updating the values of columns with each time step. The agent in **Figure 1** below is black, nutrients in the world are green, and blue pixels indicate the agent's field of view (what the organism can actually see around it).

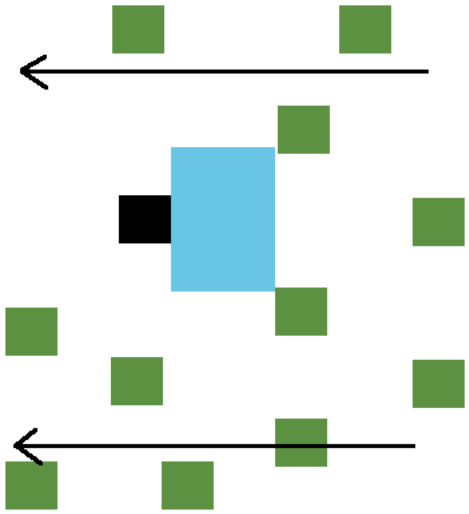

Fig. 1: Example Nutrient Environment

While the agent always remains in the same column, it can execute one of three actions at any time based on what it sees in the environment to change its row. It can stay in its current row, or it can move upwards or downwards one row.

We would like to find agents who can feed themselves - that is, organisms that can observe nutrient locations and execute the appropriate actions to pass through those pixels and ingest them. We can judge how well an agent (a solution) is performing in this environment by enforcing some kind of an objective. Some hard objectives would be to eat every piece of food that passes through its FOV, or simply eat the most food during a lifetime. A softer objective might be to just eat enough food to stay alive based on some definition of its metabolic baseline.

If we designed a neural network for this controller, it would require at least six input nodes (one for each of the pixels in its field of view) and three output nodes (one for each of the potential actions). An effective controller ANN may also include layers of hidden nodes. (**Figure 2**)

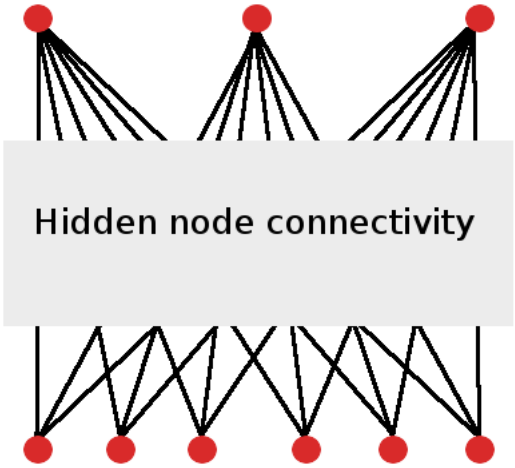

Fig. 2: Simple Controller ANN

The **evolearn** package can be used to design this basic environment for one or even multiple agents to forage for nutrients in these pixelated "Grid World" environments. From this testbed we can compare the dynamics of populations

of agents whose selection is restricted by the availability of resources at a given time, and represented by the encoding of the particular agent's neural network in its genome. The encoding we choose to implement for a genome can in turn be restricted to those of more orthodox research paradigms, mainly direct encoding schemes where each characteristic of the network must be listed in its genome, and more contemporary approaches that attempt to *compress* features of an organism into smaller genomes, known as *indirect encoding*.

This compression provides a interesting testbed for introducing operators for mating and mutation to explore parameter space more efficiently than direct encoding schemes. Importantly, this approach has practical value to the study of biological organisms, as we come to appreciate the highly distributed and interdependent encoding of genetic information in DNA. Life has arrived at indirect encodings through natural selection in order to more greatly cover this space of possible organisms as a solution for solving endless objectives of metabolism and environmental pressures, and it is possible that the future of optimization engineering and artificial intelligence research may rely on mimicking these naturally-derived solutions.

2.2 Neuroevolution: A Primer

If we would like to evolve a population of our example agents on the task described, it is important that we have a stable representation of an artificial neural network, which we will use to represent that agent's brain. With that representation, we can choose how to represent that network in a genome, making it possible to have two agent's reproduce and mutate to find new agent solutions.

Within the field of neuroevolution, there are generally two different types of neural network representations we can choose as our agent genome: *direct encodings* and *indirect encodings*.

2.2.1 Direct Encodings

Since we need to find a representation for an agent in our population, let's start with the simplest agent of that kind: a fully-connected neural network with 3 output nodes, 6 input nodes, and no hidden nodes. (**Figure 3**)

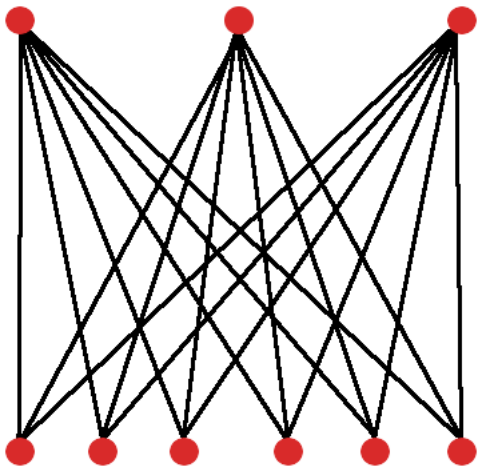

Fig. 3: Simplest Agent Neural Network

A directly encoded genome contains a one-to-one relationship between the parameters that describe the neural network and the number of genes in the associated agent genome.

For example, an agent with the network shown in **Figure 3** has 18 unique weights, and therefore a direct encoding of that agent will require at least 18 values. It may additionally require information about the number of inputs, outputs, the activation functions used in those outputs, as well parameters that influence its learning within the environment.

For the purpose of our very simple task, 18 weight values is not very much. It would not be difficult to apply standard gradient-based learning to find the appropriate parameters for this model. If we did want to represent the network with a genome, 18 genes for each of the weights is relatively small, and we don't run into many problems performing reproduction on mutation on it.

A direct encoding that will turn out to be essential to the approach of epann is known as *NeuroEvolution of Augmenting Topologies*, or NEAT. In the original paper that described the NEAT algorithm, a simple controller much like our example was described genetically as containing two separate lists to describe its neural network phenotype. (**Figure 4**)

A neural network is first described by its node genome (**Figure 4(a)**). The node genome is a list of each node in the agent's neural network, along with some characteristics about each node. Nodes have a type, which distinguishes them as being an input, output or hidden node. Individual nodes also have unique activation functions, that determine the function applied to its summed inputs.

An agent also has a connection genome (**Figure 4(b)**). The connection genome is a list of each connection in the agent's neural network, and it also has a list of attributes that describe them. They have an identification number (or, as will be called later, innovation numbers), a description of the nodes that start and end that connection (soon to be referred to as a connection's out node and in node, respectively), and a weight. Connection genes also contain a final binary attribute called the enable bit, that determines whether or not a connection described in the connection genome will actually be expressed in the final agent phenotype.

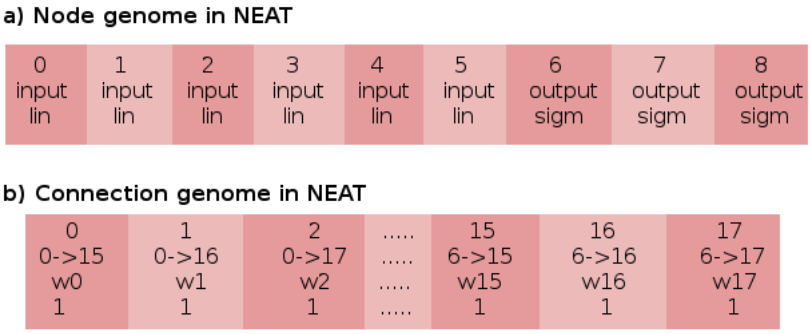

Fig. 4: Direct Encoding in the original NEAT algorithm. Node and connection numbering maintain Python's 0-based indexing left to right.

However, a direct encoding present difficulty as a general purpose representation for neural networks. Direct encodings will also require as many genes as there are parameters for more complex models, such as modern benchmark convolutional neural networks, which may contain over a million parameters. Using a genetic algorithm in a space that large will be difficult, and it is likely that candidate solutions will bounce around the space without converging.

2.2.2 Indirect Encodings

We do not see an analog to direct encoding in the genomes of biological organisms. Instead, organisms have genotypes with less information than the phenotypes they generate. The number of genes in a genome is far less than it would take to encode individual connections in its nervous system, let alone features of every cell in its body.

Instead, it seems that the emergence of life has settled on genetic representations that are far more complex. Genes are associated with the development of many different systems within an organism. As a result, a mutation in a particular

gene can have widespread effects.

Indirect encodings can also change the search space we are exploring during our optimization.

In the field of neuroevolution, different indirect encoding frameworks have been proposed to accomodate potentially immense phenotype search spaces. It turns out, an effective indirect encoding can be implemented that is a variant of the NEAT direct encoding. The candidate indirect encoding is called *Compositional Pattern Producing Networks* (CPPNs) [CPPN2007], and they act as effective genome structures for a modified version of the NEAT algorithm: *Hypercube-based NEAT*. [ES2012]

2.3 Neuroevolution via Indirect Encodings

2.3.1 Compositional Pattern Producing Networks

CPPNs are a kind of indirect encoding that represents an agent's neural network as another, smaller neural network. [CPPN2007] Since neural networks act as function-approximators, an agent's genome is in effect also a function. This neural network approximation of a function will be used to construct an agent's brain, which is also a neural network.

By representing an agent this way, it is not necessary to encode every connection in the final agent phenotype when performing evolutionary computation to find good solutions. Example

Let's start with a simple example using epann. Once we construct a population using epann, we will be able to explore a few things:

- How CPPN genomes are similar to the direct encodings of the original NEAT algorithm.
- How a CPPN genome is related to the final agent neural network phenotype.
- How a population of CPPNs reproduce, mutate, and ulimately evolve.

To begin with we initialize a population of agents that each contain a phenotype (an ANN that will directly interact with our task) and a genotype (the CPPN).

```
from epann.core.population.population import Population

num_agents = 5

pop = Population(num_agents)

for agent in pop.genomes.keys():
    print 'Agent', agent, '-', pop.genomes[agent]
```

Which outputs:

```
Agent 0 - <epann.core.population.genome.cppn.CPPN instance at 0x7f4d58219950>
Agent 1 - <epann.core.population.genome.cppn.CPPN instance at 0x7f4d58219ab8>
Agent 2 - <epann.core.population.genome.cppn.CPPN instance at 0x7f4d58219bd8>
Agent 3 - <epann.core.population.genome.cppn.CPPN instance at 0x7f4d58219cf8>
Agent 4 - <epann.core.population.genome.cppn.CPPN instance at 0x7f4d58219e18>
```

As you can see, each agent is defined as an instance of a CPPN object. Within this object are attributes that define its genotype, which can then be used to construct a phenotype for the agent.

If we set aside the first agent (Agent 0), we will see something familiar. Similar to the original NEAT direct encoding, CPPNs are neural networks that can be described by node and connection genome lists. [ES2012] We can take a look at the CPPN visually first in **Figure 4** before we explore how these genomes are related to network structure in the same way we saw in Part 2.

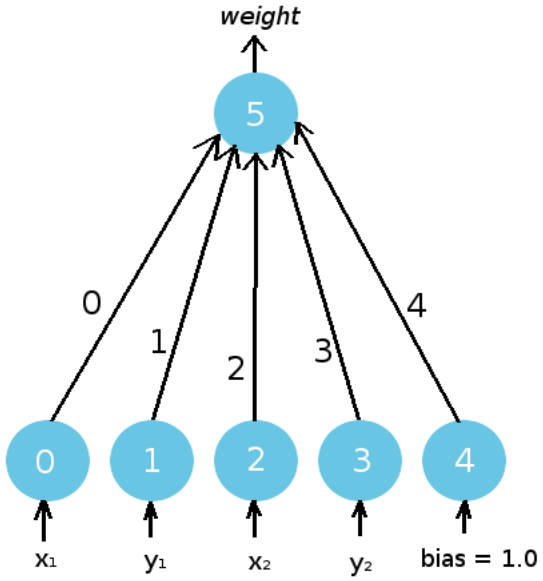

Fig. 5: An Initialized CPPN Genotype.

Since an agent's neural network (**Figure 3**) and its CPPN (**Figure 5**) are both networks, we have colored them differently to avoid confusion. The brain of an agent will always have nodes that are colored red or orange, while CPPNs will have blue nodes. CPPNs will have identification numbers on their nodes and connections, while an agent's brain no longer will.

The Node Genome

From our previous program, we can begin adding genomes to the environment.

```
agent_index = 0
current_agent = pop.genomes[agent_index]

print current_agent.nodes
```

Which returns:

```
{0: {'activation': 'linear', 'type': 'input'}, 1: {'activation': 'linear', 'type':
↪'input'}, 2: {'activation': 'linear', 'type': 'input'}, 3: {'activation': 'linear',
↪'type': 'input'}, 4: {'activation': 'linear', 'type': 'input'}, 5: {'activation':
↪'abs_value', 'type': 'output'}}
```

Just like we saw before, a CPPN has a node genome that describes the characteristics of its individual nodes. Each key is a node in the genome, and each nested dictionary is that particular node's attributes. Each of the nodes have an identification number specific to its node genome (written in white in (**Figure 5**).

For example, Node 5 is an output node ('type') with a unique activation function ('activation').

(Note: it might seem odd that an output node does not have a more traditional activation function, such as the sigmoid. Neurons in CPPNs can have a variety of activation functions that are selected for their ability to introduce repetition or symmetry, which gives rise to the network's pattern producing capabilities. More on this distinction later.)

For now, we can at least observe the possible activation functions output nodes can be assigned to:

```
from epann.core.tools.utils.activations import Activation

acts = Activation()
print acts.tags
```

Returning:

```
['x_cubed', 'linear', 'sigmoid', 'ramp', 'gauss', 'abs_value', 'tan_h', 'step', 'ReLU
↪', 'sine']
```

Nodes within the CPPN (except for the input nodes) can have any of these activation functions. For now, let's set the activation function to something simple.

```
# Save the old randomly generated activation function
old_output_act = current_agent.nodes[5]['activation']

# Re-assign the ouput node activation to something simple
current_agent.nodes[5]['activation'] = 'ramp'
```

We start a generation off with 5 agents that have the same number of input and output nodes. As a result, every agent will have identical node genomes when they are initialized, save the specific activation functions assigned to the output nodes.

```
# Input nodes
print '\nInput nodes are equivalent across the population when initialized...\n'
for agent in range(num_agents):
    print '\n- Agent', agent
    for node in range(5):
        print '    Node', node, ':', pop.genomes[agent].nodes[node]

# Output nodes
print '\nWhile output nodes differ in their specific activation functions...\n'
for agent in range(num_agents):
    print '\n- Agent', agent
    print '    Node', 5, ':', pop.genomes[agent].nodes[5]
```

Which returns:

```
Input nodes are equivalent across the population when initialized...


- Agent 0
    Node 0 : {'activation': 'linear', 'type': 'input'}
    Node 1 : {'activation': 'linear', 'type': 'input'}
    Node 2 : {'activation': 'linear', 'type': 'input'}
    Node 3 : {'activation': 'linear', 'type': 'input'}
    Node 4 : {'activation': 'linear', 'type': 'input'}

- Agent 1
    Node 0 : {'activation': 'linear', 'type': 'input'}
    Node 1 : {'activation': 'linear', 'type': 'input'}
    Node 2 : {'activation': 'linear', 'type': 'input'}
    Node 3 : {'activation': 'linear', 'type': 'input'}
    Node 4 : {'activation': 'linear', 'type': 'input'}

- Agent 2
```

(continues on next page)

(continued from previous page)

```
    Node 0 : {'activation': 'linear', 'type': 'input'}
    Node 1 : {'activation': 'linear', 'type': 'input'}
    Node 2 : {'activation': 'linear', 'type': 'input'}
    Node 3 : {'activation': 'linear', 'type': 'input'}
    Node 4 : {'activation': 'linear', 'type': 'input'}

- Agent 3
    Node 0 : {'activation': 'linear', 'type': 'input'}
    Node 1 : {'activation': 'linear', 'type': 'input'}
    Node 2 : {'activation': 'linear', 'type': 'input'}
    Node 3 : {'activation': 'linear', 'type': 'input'}
    Node 4 : {'activation': 'linear', 'type': 'input'}

- Agent 4
    Node 0 : {'activation': 'linear', 'type': 'input'}
    Node 1 : {'activation': 'linear', 'type': 'input'}
    Node 2 : {'activation': 'linear', 'type': 'input'}
    Node 3 : {'activation': 'linear', 'type': 'input'}
    Node 4 : {'activation': 'linear', 'type': 'input'}

While output nodes differ in their specific activation functions...


- Agent 0
    Node 5 : {'activation': 'ramp', 'type': 'output'}

- Agent 1
    Node 5 : {'activation': 'ReLU', 'type': 'output'}

- Agent 2
    Node 5 : {'activation': 'step', 'type': 'output'}

- Agent 3
    Node 5 : {'activation': 'x_cubed', 'type': 'output'}

- Agent 4
    Node 5 : {'activation': 'x_cubed', 'type': 'output'}
```

The Connection Genome

The CPPN also has a connection genome that keeps track of the connections between these nodes:

```
print current_agent.connections
```

```
{0: {'enable_bit': 1, 'in_node': 5, 'weight': 1.3157653667504219, 'out_node': 0}, 1: {
↪'enable_bit': 1, 'in_node': 5, 'weight': -0.1669125280086818, 'out_node': 1}, 2: {
↪'enable_bit': 1, 'in_node': 5, 'weight': -0.3487183711861378, 'out_node': 2}, 3: {
↪'enable_bit': 1, 'in_node': 5, 'weight': -1.5476773857272677, 'out_node': 3}, 4: {
↪'enable_bit': 1, 'in_node': 5, 'weight': 0.31068179560931486, 'out_node': 4}}
```

Just like any neural network you are accustomed to seeing, a CPPN is composed of an input layer (Nodes 0-4) and an output layer (with a single output node, Node 5).

Each agent begins with 6 total nodes which are fully connected, making 5 initial weights. Although agents in the population are structurally identical (they have the same number of initial nodes in their CPPN), the weights of these

connections will not be the same for each agent. Similar to before, each connection has an innovation number that identifies it.

Let's compare a single connection across the population - Connection 0, between Node 0 and Node 5 to show this fact:

```
print 'Connection weights are randomly initialized across the population for the same␣
↪connection...\n'
for agent in range(num_agents):
    print '\n- Agent', agent
    print '    Connection', 0, ':', pop.genomes[agent].connections[0]
```

Gives us:

```
Connection weights are randomly initialized across the population for the same␣
↪connection...


- Agent 0
    Connection 0 : {'enable_bit': 1, 'in_node': 5, 'weight': 1.3157653667504219, 'out_
↪node': 0}

- Agent 1
    Connection 0 : {'enable_bit': 1, 'in_node': 5, 'weight': -1.460571493663236, 'out_
↪node': 0}

- Agent 2
    Connection 0 : {'enable_bit': 1, 'in_node': 5, 'weight': 0.4822892299948137, 'out_
↪node': 0}

- Agent 3
    Connection 0 : {'enable_bit': 1, 'in_node': 5, 'weight': 2.042117498128114, 'out_
↪node': 0}

- Agent 4
    Connection 0 : {'enable_bit': 1, 'in_node': 5, 'weight': -1.2561727328710062,
↪'out_node': 0}
```

We can verify that Connection 0, which has the innovation number 0 in both the connection genome and in our visualization in **Figure 5**, describes a connection between Node 0 and Node 5. These landing points are intuitively labeled within the gene: Connection 0 goes out from Node 0 ('out_node'), and terminates into Node 5 ('in_node').

You can go over the connection genome for Agent 0 and compare it to **Figure 5** to verify that the rest of the connections share this convention.

```
for connection in current_agent.connections.keys():
    print 'Innovation #:', connection, '-', current_agent.connections[connection]
```

Gives us:

```
Innovation #: 0 - {'enable_bit': 1, 'in_node': 5, 'weight': 1.3157653667504219, 'out_
↪node': 0}
Innovation #: 1 - {'enable_bit': 1, 'in_node': 5, 'weight': -0.1669125280086818, 'out_
↪node': 1}
Innovation #: 2 - {'enable_bit': 1, 'in_node': 5, 'weight': -0.3487183711861378, 'out_
↪node': 2}
Innovation #: 3 - {'enable_bit': 1, 'in_node': 5, 'weight': -1.5476773857272677, 'out_
↪node': 3}
Innovation #: 4 - {'enable_bit': 1, 'in_node': 5, 'weight': 0.31068179560931486, 'out_
↪node': 4}
```

Just like before, the connection genome has attributes that define the 'weight' and 'enable_bit' for each connection.

The relationship between an agent's neural network and a CPPN should still be relatively unclear at this point, and that's because we're missing an important component that links the two together.

What are the inputs to a CPPN? The outputs?

A CPPN can act as a mapping by evaluating potential connections in the final phenotype, and the component we are missing in order to accomplish this is something called the substrate: a two-dimensional coordinate space that defines the locations of neurons in the agent's brain.

The Substrate: Mapping Genotype to Phenotype

An important starting point when thinking about our CPPN abstraction is that we are going to treat our final agent neural network, the brain that will actually be interacting with the task we are interested in, as something that exists in a two-dimensional coordinate space. As a consequence of constructing this plane, individual nodes within a CPPN can be defined with coordinates on that plane.

To begin, let's place the nodes from our minimum agent controller network onto the plane so that we can investigate these properties.

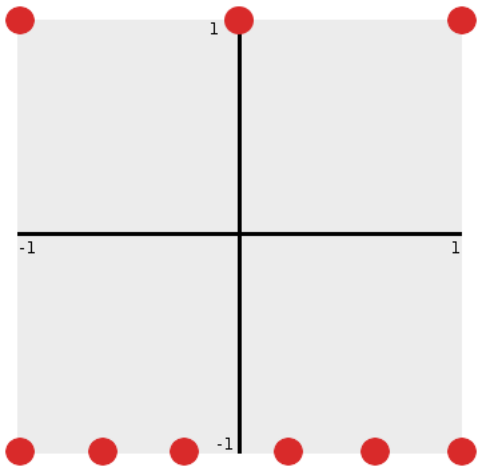

Fig. 6: Agent Controller Nodes on a Substrate.

From **Figure 6** we can immediately notice a few things:

- The substrate coordinate space is continuous between *-1* and *+1* in both *x* and *y*. (That is, *x*, *y* are in [*-1*, *1*].)
- Input nodes can be defined with coordinates that describe their location along the bottom edge of this space: (*x_i*, *-1*).
- Output nodes can be defined with coordinates that describe ther location along the top edge of this space: (*x_i*, *+1*).

If we assume that within a layer nodes are evenly spaced along the width of the plane, we can initialize their locations as:

```
import numpy as np

input_locations = [ [ round(xi, 2), -1.0 ] for xi in np.linspace(-1, 1, 6) ]
```

(continues on next page)

(continued from previous page)

```
output_locations = [ [ round(xo, 2), 1.0 ] for xo in np.linspace(-1, 1, 3) ]

print '\nInput locations:\n', input_locations
print '\nOutput locations:\n', output_locations
```

Where we get:

```
Input locations:
[[-1.0, -1.0], [-0.6, -1.0], [-0.2, -1.0], [0.2, -1.0], [0.6, -1.0], [1.0, -1.0]]

Output locations:
[[-1.0, 1.0], [0.0, 1.0], [1.0, 1.0]]
```

The purpose of our CPPN genome is to allow us to more compactly represent an agent's phenotype with an indirect encoding. The way were are able to do this is by treating a genome as a function of a possible connection in the phenotype. More specifically, A CPPN genome samples a possible connection between two nodes, which are represented within a substrate as a pair of coordinates, and uses them as inputs to the CPPN network. In return, we can receive a property of that connection in the phenotype - for example, the weight of that connection.

CHAPTER 3

Modules

3.1 evolearn.algorithms module

class evolearn.algorithms.neat.**NEAT**(*population_size*, *num_inputs*, *num_outputs*)
NeuroEvolution of Augmenting Topologies (NEAT) Population Class.

Parameters

- **population_size** (*int*) – number of agents in the current simulation's population.
- **num_inputs** (*int*) – environment observation space.
- **num_outputs** (*int*) – environment action space.

single_evaluation(*net*, *current_input*)
Evaluate agent phenotype network on current environment input.

Parameters

- **net** – agent phenotype network.
- **current_input** – current environment observation.

Returns network output

3.2 evolearn.analysis module

class evolearn.analysis.neighbor_join.**NeighborJoin**
Neighbor joining analysis class.

3.3 evolearn.controllers module

class evolearn.controllers.controller_simple.**SimpleAgent**(*world_size*)
Simple agent object for interacting with Simple Wrapping Environments.

cyclical_heading(*heading*)
Heading conversion. Prevents requests for heading indices that do not exist (range(4) possible).

Parameters **heading** – original agent.heading

Returns converted (cyclical) agent.heading

define_fov(*world*)
Defines the agent's field-of-vision.

Parameters **world** – current environment

Returns observation

define_fov_heading()
Defines the structure of an agent's field-of-vision

Returns FOV_heading

enforce_wrapping(*position*)
Location conversion. Prevents requests for locations that are not accessible in the current environment.

Parameters **position** – world positionX or world positionY

Returns new world positionX or world positionY

heading_row_calc(*r*, *th*, *t*, *l*)
Calculates location indices for individual cells in a row of an agent's field-of-view.

Todo:

- Flush out input parameters definitions.
- Rename input parameters to more readable ones.

Return output

reset()
Reset agent location and heading in environment to initial values.

3.4 evolearn.environments module

class evolearn.environments.environment_gym.**BalancingPole**
FlappyBird test class

Parameters **genome** (*NaN*) – population of cppn networks.

class evolearn.environments.environment_gym.**FlappyBird**
FlappyBird test class

Parameters **genome** (*NaN*) – population of cppn networks.

class evolearn.environments.environment_simple.**Recognition**
General image recognition object.

class evolearn.environments.environment_simple.**SimpleEnvironment**
Simple wrapped callable nutrient environment.

Todo:

- Allow for the import of txt file for defining maze/track boundaries.
- Connect imported boundaries to evaluation loop break collision flag.

build_actions()
Builds an accessible dictionary of possible actions to be called with each agent action to provide adjustments for location and heading adjustments.

Returns environment action dict. Indices define position and heading adjustments for a selected action.

collision_check()
Collision check to potentially break current agent's evaluation.

Returns collide Boolean

initialize_environment()
Initialize environment.

Returns initialized world

make_observation()
Making an observation for a single step through environment.

move_agent(*action*)
Update agent location based on selected action.

reformat_action(*agent_output*)
Reformat raw network output into environment-specific (or experiment specified) action/class choice.

Returns reformatted action/class index

reset()
Complete environment reset.

Returns intial environment observation

return_reward()
Returns reward for agent's current location.

Returns reward/state at agent.location

step(*action*)
Making a single step through the environment.

Returns next observation, current reward, collision Boolean.

update(*action*)
Update environment.world with respect possibly consumed nutrients at agent's current location.

3.5 evolearn.experiments module

3.6 evolearn.utils module

CHAPTER 4

Development Information

4.1 Credits

4.1.1 Development Lead

- Chad Carlson <ccarlson2013@fau.edu>

4.1.2 Contributors

None yet. Why not be the first?

4.2 Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

4.2.1 Types of Contributions

Report Bugs

Report bugs at https://github.com/chadwcarlson/evolearn/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" and "help wanted" is open to whoever wants to implement it.

Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" and "help wanted" is open to whoever wants to implement it.

Write Documentation

evolearn could always use more documentation, whether as part of the official evolearn docs, in docstrings, or even on the web in blog posts, articles, and such.

Submit Feedback

The best way to send feedback is to file an issue at https://github.com/chadwcarlson/evolearn/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

4.2.2 Get Started!

Ready to contribute? Here's how to set up *evolearn* for local development.

1. Fork the *evolearn* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/evolearn.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv evolearn
$ cd evolearn/
$ python setup.py develop
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ flake8 evolearn tests
$ python setup.py test or py.test
$ tox
```

To get flake8 and tox, just pip install them into your virtualenv.

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

4.2.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 2.6, 2.7, 3.3, 3.4 and 3.5, and for PyPy. Check https://travis-ci.org/chadwcarlson/evolearn/pull_requests and make sure that the tests pass for all supported Python versions.

4.2.4 Tips

To run a subset of tests:

```
$ python -m unittest tests.test_evolearn
```

4.3 History

4.3.1 0.1.0 (2017-05-03)

- First release on PyPI.

genindex modindex

Bibliography

[CPPN2007] Stanley, Kenneth O. “Compositional pattern producing networks: A novel abstraction of development.” *Genetic programming and evolvable machines* 8.2 (2007): 131-162.)

[ES2012] Risi, Sebastian, and Kenneth O. Stanley. “An enhanced hypercube-based encoding for evolving the placement, density, and connectivity of neurons.” *Artificial life* 18.4 (2012): 331-363.)

[CPPN2007] Stanley, Kenneth O. “Compositional pattern producing networks: A novel abstraction of development.” *Genetic programming and evolvable machines* 8.2 (2007): 131-162.)

[ES2012] Risi, Sebastian, and Kenneth O. Stanley. “An enhanced hypercube-based encoding for evolving the placement, density, and connectivity of neurons.” *Artificial life* 18.4 (2012): 331-363.)

Python Module Index

e

Index

U