abrain

Release 1.0rc-post5

Kevin Godin-Dubois

Apr 24, 2024

CONTENTS

This package implements the ES-HyperNEAT algorithms for the production of large-scale, regular Artificial Neural Networks. For a adequate overview of the related literature see the official homepage (http://eplex.cs.ucf.edu/ESHyperNEAT/).

CHAPTER

ONE

CONTENTS

1.1 Usage

1.1.1 Installation

End-user installation

```
(.venv)$ pip install abrain
```

Precompiled wheels are available for most linux distributions, macosx and windows on python 3.8+. See https://github.com/kgd-al/abrain/wiki/wheels for the full list.

Editable install

Due to inconsistent behavior with pip editable install, it is recommended to instead clone the repository and use the built dedicated install command:

```
$ git clone https://github.com/kgd-al/abrain.git
$ ./commands.sh install-dev
OR
$ ./commands.sh install-dev-tests
```

By default these commands will produce a debug build with symbols, suitable for coverage monitoring. If you want a performance oriented build, instead use:

```
$ ./commands.sh install-cached release
```

1.1.2 Tutorials

Basic usage

This section showcases the main components of the library by detailing the contents of *examples/basics.py*.

```
1 import sys
2 from random import Random
3
4 from abrain import Genome, Point as Pt, ANN, plotly_render
5 from common import example_path
```

We start by importing the essential components, aliased directly under the main package:

- *Genome* abstracts the evolvable part of the library
- *ANN* is the callable object representing an Artificial Neural Network of emergent topology
- *Point* describes a coordinate in the substrate ("the brain")
- *plotly_render()* is a helper function for rendering ANN to a, potentially interactive, figure
- random.Random is used as the source of random numbers

```
rng = Random(seed)
g = Genome.random(rng)
for _ in range(mutations):
    g.mutate(rng)
```

The first object we need is the Genome which can be created by providing the random number generator to its *random()* function. To simulate an evolutionary process, we subject this Genome g to a number of undirected mutations (see *Mutations*)

```
inputs = [Pt(-1, -1, -1), Pt(-1, -1, 1), Pt(1, -1, -1), Pt(1, -1, 1)]
outputs = [Pt(0, 1, -1), Pt(0, 1, 0), Pt(0, 1, 1)]
```

Before instantiating the ANN, we define the coordinates of the neural inputs (sensors) and outputs (effectors). Thanks to the ES-HyperNEAT algorithms the topology (hidden neurons & connections) will be automatically determined. As much as possible, the provided coordinates should respect the geometrical relationships (i.e. bilateral symmetry, front-back ...).

Warning: It is essential that neurons are placed at *unique* coordinates including hidden ones. Safe coordinates for inputs/outputs are of the form

$$\{(x,y,z)/\ \exists c \in \{x,y,z\}/\ \ \nexists \{i_1,...,i_n\}/\ c = \sum_{j=1}^{n} 2^{-i_j}\}$$

with $1 \leq i_j \leq$ *maxDepth*

In particular, this means that all outside planes (e.g. $y = \pm 1$) can never contain hidden neurons and are thus safe for user-defined inputs/outputs

```
ann = ANN.build(inputs, outputs, g)
print(f"empty ANN: {ann.empty()}")
print(f"maximal depth: {ann.stats().depth}")
```

Creating the ANN is then as trivial as calling the static *build()* function with the set of inputs/outputs and the evolved genome. Various statistics can be queried on the resulting object including whether the build procedure resulted in a functional network.

```
plotly_render(ann).write_html(example_path("./sample_ann.html"))
```

Optionally, one can produce a 3D rendering of the network through the utility function *plotly_render()*.

```
inputs, outputs = ann.buffers()
inputs[0] = 1
inputs[1:3] = [rng.uniform(-1, 1) for _ in range(2)]
inputs[3] = -1
```

Actually using the ANN requires defining the neural inputs at a given time step, which can be done by direct assignment (line 29) or through slices (line 30). At the same time we also retrieve the output buffer which will store the neural responses computed in the next step.

Note: The default activation function for every *hidden* and *output* neurons maps 0 to 0. By contrast input neurons expose the exact same value as that provided. This means that providing constant, small values might result in the whole network staying in a quiescent state.

```
36 n = 5
37 ann(inputs, outputs, substeps=n)
```

Following that, we can query the outbound activity by invoking the ANN with both buffers. An optional parameter *substeps* can be provided if more than a single activation step is desired, e.g. deep networks with a low update rate.

```
40 print("Outputs:", outputs[0], outputs[1:3])
```

As with the input buffer, the results can be queried individually or in bulk to set the robot's outputs (motors…).

Basic evolution

This section focuses on using the library in an evolutionary context. It showcases:

- how to include *abrain.Genome* into another class
- how to use the configuration
- how to actually produce novel solutions

First we import the relevant modules from the library (among others)

```
6 from abrain import Config, Genome, ANN, Point
7 from abrain.core.config import Strings
8 from abrain.core.genome import GIDManager
```

As with the previous example, we need *Genome*, *ANN* and *Point* to encode/decode an Artificial Neural Network. *Config* is responsible for statically stored settings and persistent configuration files. In this specific case, we also need *Strings* for one particular value.

Finally, the *GIDManager* is a tiny helper class to provide consecutive int-based id to the genome. It can be used as-is, replaced with a more powerful alternative or just ignored if you do not need to identify individual genomes.

Helper classes

To showcase real use of the genome, we define a trivial wrapper containing two fields:

```
14 class MyGenome:
15     def __init__(self, abrain_genome: Genome, nested_field: float):
16         self.abrain_genome = abrain_genome
17         self.nested_field = nested_field
18
19     @staticmethod
20     def random(rng: Random, id_m: GIDManager):
21         return MyGenome(Genome.random(rng, id_manager=id_m),
```

(continues on next page)

(continued from previous page)

```
                        rng.uniform(-1, 1))

    def mutate(self, rng: Random):
        if rng.random() < .9:
            self.abrain_genome.mutate(rng)
        else:
            self.nested_field += rng.normalvariate(0, 1)

    def mutated(self, rng: Random, id_m: GIDManager):
        copy = self.copy()
        copy.mutate(rng)
        copy.abrain_genome.update_lineage(id_m, [self.abrain_genome])
        return copy

    def copy(self):
        return MyGenome(
            self.abrain_genome.copy(),
            self.nested_field
        )
```

The presented pattern consists of the two essential functions *random* (to generate the initial population) and *mutated* (to create a mutated copy of a genome). The *mutate* function performs the bulk of the work by delegating to field-wise mutators (including *abrain.Genome.mutate()*). Note that, if using an id generator (such as *abrain.GIDManager*) you can use the *update_lineage()* function to update the *id*/*parents* fields based on the values of the parents (self in case of a mutation).

We then define an individual, in the sense of an evolutionary algorithm, as the composition of a genome and a fitness (trivially based on the ANN's depth). For completeness, we provide a serialization method which relies on *abrain.Genome.to_json()*.

```
class Individual:
    _inputs = [Point(x, -1, z) for x, z in [(0, 0), (-1, -1), (1, 1)]]
    _outputs = [Point(x, 1, z) for x, z in [(0, 0), (1, -1), (-1, 1)]]

    def __init__(self, genome: MyGenome):
        self.genome = genome
        self.fitness = None

    def evaluate(self):
        if self.fitness is None:
            ann = ANN.build(self._inputs, self._outputs,
                            self.genome.abrain_genome)
            self.fitness = self.genome.nested_field * ann.stats().depth

    def write(self, file):
        json.dump(dict(
            abrain_genome=self.genome.abrain_genome.to_json(),
            float_field=self.genome.nested_field,
            fitness=self.fitness
        ), file)
```

The main

The following sections describe the components of a trivial EA and how to use the various parts of *abrain* to smoothly implement them.

Configuration

The following lines showcase how the end-user can tweak the various fields in *abrain.Config*:

```
71    Config.functionSet = Strings(['sin', 'abs', 'id'])
72    Config.allowPerceptrons = False
73    Config.iterations = 4
74    Config.write(output_folder.joinpath("config.json"))
75    Config.show()
```

Most such fields use elementary python types (*int*, *float*, *str*, *bool*) and can thus be trivially manipulated. A few other use composite types encapsulated, for type-safety, in a C++ object. Those are exposed in the *abrain.core.config* module and can be used to generate new values. Additionally, the configuration can be written to a file (and read back with *read()*) and displayed on the screen (for the log).

Variables

The initial state of this trivial EA is just as straightforward. The only thing of note is the highlighted statement where we create the genome id manager. This is purely optional and only provided for convenience.

The actual generation of the initial population simply consists of delegating the work to the dedicated function in our wrapper genome.

```
79    seed = 0
80    rng = Random(seed)
81    id_manager = GIDManager()
82    population = [Individual(MyGenome.random(rng, id_manager))
83                  for _ in range(100)]
```

CPPN

Todo: Examining the CPPN

Mutations

Todo: Discuss mutations

1.1.3 FAQ

Windows

Exporting an ANN through plotly requires UTF-8 encoding which is not the default. Setting an environment variable to *PYTHONUTF8=1* fixes the problem

Kaleido

Rendering an ANN in non-interactive format requires either kaleido or orca. While the former is unavailable on some distributions, the latter seems out

1.2 API

1.2.1 Basics

Configuration

class abrain.**Config**

Wrapper for the C++ configuration data

Allows transparent access to the values used for CPPN structure, genome mutation (*Genome.mutate()*), and ANN/ES-HyperNEAT parameters

classmethod from_json(*j: Dict*)

Restore values from a json-compliant Python dictionary

Parameters
j – the dictionary to parse values from

static known_function(*name: str*) → bool

Whether the requested function name is a built-in

classmethod read(*path: Path*)

Try to load data from provided path

Parameters
path – Filename

classmethod show()

Write the configuration on standard output

classmethod to_json() → Dict

Convert to a json-compliant Python dictionary

classmethod write(*path: Path | None*)

Write the configuration to the specified file or stdout

Parameters
path – where to write or none to print to screen

activationFunc = 'ssgn'

The activation function used by all hidden/output neurons (inputs are passthrough)

allowPerceptrons = True

Attempt to generate a perceptron if no hidden neurons were discovered

annWeightsRange = 3.0

Scaling factor s for the CPPN w output mapping $[-1, 1]o[-s, s]$

bndThr = 0.15000000596046448

Minimal divergence threshold for discovering neurons

cppnInputNames = Strings[x_0, y_0, z_0, x_1, y_1, z_1, l, b]

`const` Auto generated name of the CPPN inputs (based on dimensions and optional use of the connection length)

cppnOutputNames = Strings[w, l, b]

`const` Auto generated name of the CPPN outputs

cppnWeightBounds = Bounds(-3, -1, 1, 3, 0.01)

Initial and maximal bounds for each of the CPPN's weights

divThr = 0.30000001192092896

Division threshold for a quad-/octtree cell/cube

functionSet = Strings[abs, gaus, id, bsgm, sin, step]

List of functions accessible to nodes via creation/mutation

initialDepth = 2

Initial division depth for the underlying quad-/octtree

iterations = 10

Maximal number of discovery steps for Hidden/Hidden connections. Can stop early in case of convergence (no new neurons discovered)

maxDepth = 3

Maximal division depth for the underlying quad-/octtree

mutationRates = MutationRates{add_l: 0.0681818, add_n: 0.0454545, del_l: 0.0909091, del_n: 0.0681818, mut_f: 0.227273, mut_w: 0.5}

Probabilities for each point mutation (addition/deletion/alteration)

Glossary:

- add_l: add a random link between two nodes (feedforward only)
- add_n: replace a link by creating a node
- del_l: delete a random link (never leaves unconnected nodes)
- del_n: replace a simple node by a direct link
- mut_f: change the function of a node
- mut_w: change the connection weight of a link

outputFunctions = Strings[bsgm, step, id]

Functions used for the CPPN output (same length as *cppnOutputNames*)

varThr = 0.30000001192092896

Variance threshold for exploring a quad-/octtree cell/cube

class abrain.core.config.**Strings**(*values: list[str]*)

class abrain.core.config.**MutationRates**(*values: dict[str, float]*)

class abrain.core.config.**FBounds**(*min: float*, *rndMin: float*, *rndMax: float*, *max: float*, *stddev: float*)

A wrapper for mutation bounds. Absolute range is *[min, max]*. Values produced through random initialization are further restricted to *[rndMin, rndMax]* with

$$min \leq rndMin \leq rndMax \leq max$$

stddev is the standard deviation for every point-mutation applied to the corresponding field.

Contains the classes and functions related to abrain's configuration

Genome

Main Object:

class abrain.**Genome**(*key=None*)

Genome class for ES-HyperNEAT.

A simple collection of Node/Link. Can only be created via random init or copy

id() → int | None

Return the genome id if one was generated

parents() → int | None

Return the genome's parent(s) if possible

mutate(*rng: Random*) → None

Mutate (in-place) this genome

Parameters
rng – The source of randomness

mutated(*rng: Random*, *id_manager:* GIDManager | *None = None*) → *Genome*

Return a mutated (copied) version of this genome

Parameters

- **rng** – the source of randomness
- **id_manager** – an optional manager providing unique identifiers

static random(*rng: Random*, *id_manager:* GIDManager | *None = None*) → *Genome*

Create a random CPPN with boolean initialization

Parameters

- **rng** – The source of randomness
- **id_manager** – an optional manager providing unique identifiers

Returns
A random CPPN genome

copy() → *Genome*

Return a perfect (deep)copy of this genome

update_lineage(*id_manager:* GIDManager, *parents: List[*Genome*]*)

Update lineage fields

Parameters

- **id_manager** – generator of unique identifiers
- **parents** – list (potentially empty) of this genome's parents

to_json() → Dict[Any, Any]

Return a json (dict) representation of this object

static from_json(*data: Dict[Any, Any]*) → *Genome*

Recreate a Genome from a string json representation

static from_dot(*path: str*, *rng: Random*) → *Genome*

Produce a Genome by parsing a simplified graph description

> **Warning:** Unimplemented

Parameters

- **path** – the file to load
- **rng** – random pick unspecified functions

Returns

The user-specified CPPN

to_dot(*path: str*, *ext: str = 'pdf'*, *title: str | None = None*, *debug=None*) → str

Produce a graphical representation of this genome

Note: Missing functions images in nodes and/or lots of warning message are likely caused by misplaced image files. In doubt perform a full reinstall

---

Parameters

- **path** – The path to write to
- **ext** – The rendering format to use
- **title** – Optional title for the graph (e.g. for generational info)
- **debug** – Print more information on the graph. Special values:
  - 'depth' will display every nodes' depth

Raises

OSError – if the *dot* program is not available (not installed, on the path and executable)

Underlying types:

class abrain.**GIDManager**

Simple integer-producing class for unique genome identifier

class abrain._cpp.genotype.**CPPNData**

C++ supporting type for genomic data

class Link

From-to relationship between two computational node

dst

ID of the destination node

id

Numerical identifier

src

ID of the source node

weight

Connection weight

class Links

Collection of Links

append(*self:* abrain._cpp.genotype.CPPNData.Links, *x:* abrain._cpp.genotype.CPPNData.Link) → None

Add an item to the end of the list

clear(*self:* abrain._cpp.genotype.CPPNData.Links) → None

Clear the contents

extend(**args*, ***kwargs*)

Overloaded function.

1. extend(self: abrain._cpp.genotype.CPPNData.Links, L: abrain._cpp.genotype.CPPNData.Links) -> None

Extend the list by appending all the items in the given list

2. extend(self: abrain._cpp.genotype.CPPNData.Links, L: Iterable) -> None

Extend the list by appending all the items in the given list

insert(*self:* abrain._cpp.genotype.CPPNData.Links, *i: int*, *x:* abrain._cpp.genotype.CPPNData.Link) → None

Insert an item at a given position.

pop(**args*, ***kwargs*)

Overloaded function.

1. pop(self: abrain._cpp.genotype.CPPNData.Links) -> abrain._cpp.genotype.CPPNData.Link

Remove and return the last item

2. pop(self: abrain._cpp.genotype.CPPNData.Links, i: int) -> abrain._cpp.genotype.CPPNData.Link

Remove and return the item at index `i`

class Node

Computational node of a CPPN

func

Function used to compute

id

Numerical identifier

class Nodes

Collection of Nodes

append(*self:* abrain._cpp.genotype.CPPNData.Nodes, *x:* abrain._cpp.genotype.CPPNData.Node) → None

Add an item to the end of the list

clear(*self:* abrain._cpp.genotype.CPPNData.Nodes) → None

Clear the contents

extend(**args*, ***kwargs*)

Overloaded function.

1. extend(self: abrain._cpp.genotype.CPPNData.Nodes, L: abrain._cpp.genotype.CPPNData.Nodes) -> None

Extend the list by appending all the items in the given list

2. extend(self: abrain._cpp.genotype.CPPNData.Nodes, L: Iterable) -> None

Extend the list by appending all the items in the given list

insert(*self:* abrain._cpp.genotype.CPPNData.Nodes, *i:* *int*, *x:* abrain._cpp.genotype.CPPNData.Node) → None

Insert an item at a given position.

pop(**args*, ***kwargs*)

Overloaded function.

1. pop(self: abrain._cpp.genotype.CPPNData.Nodes) -> abrain._cpp.genotype.CPPNData.Node

Remove and return the last item

2. pop(self: abrain._cpp.genotype.CPPNData.Nodes, i: int) -> abrain._cpp.genotype.CPPNData.Node

Remove and return the item at index `i`

static from_json(*j:* *dict*) → *abrain._cpp.genotype.CPPNData*

Convert from the json-compliant Python dictionary *j*

to_json(*self:* abrain._cpp.genotype.CPPNData) → dict

Convert to a json-compliant Python dictionary

INPUTS = 8

Number of inputs for the CPPN

OUTPUTS = 3

Number of outputs for the CPPN

links

The collection of inter-node relationships

nextLinkID

ID for the next random link (monotonic

nextNodeID

ID for the next random node (monotonic)

nodes

The collection of computing nodes

Contains the classes and functions related to abrain's genotype

Artificial Neural Network

Todo: Missing reference to plotly (hacked something in but fails with 404)

Todo: Implement read-/write-through I/O Buffers

Main object:

class abrain.**ANN**

3D Artificial Neural Network produced through Evolvable Substrate Hyper-NEAT

__call__(*self:* abrain.ANN, *inputs:* abrain.ANN.IBuffer, *outputs:* abrain.ANN.OBuffer, *substeps:* *int* = *1*) → None

Execute a computational step

Assigns provided input values to corresponding input neurons in the same order as when created (see build). Returns output values as computed. If not otherwise specified, a single computational substep is executed. If need be (e.g. large network, fast response required) you can requested for multiple sequential execution in one call

Parameters

- **inputs** – provided analog values for the input neurons
- **outputs** – computed analog values for the output neurons
- **substeps** – number of sequential executions

See also:

Basic usage

buffers(*self:* abrain.ANN) → tuple[*abrain.ANN.IBuffer*, *abrain.ANN.OBuffer*]

Return the ann's I/O buffers as a tuple

static build(*inputs:* *list[*abrain.Point*]*, *outputs:* *list[*abrain.Point*]*, *genome:* abrain._cpp.genotype.CPPNData) → *abrain.ANN*

Create an ANN via ES-HyperNEAT

The ANN has inputs/outputs at specified coordinates. A CPPN is instantiated from the provided genome and used to query connections weight, existence and to discover hidden neurons locations

Parameters

- **inputs** – coordinates of the input neurons on the substrate
- **outputs** – coordinates of the output neurons on the substrate
- **genome** – genome describing a cppn (see *abrain.Genome*, *CPPN*)

See also:

Basic usage

empty(*self:* abrain.ANN, *strict: bool = False*) → bool

Whether the ANN contains neurons/connections

Parameters

strict – whether perceptrons count as empty (true) or not (false)

See also:

Config::allowPerceptrons

ibuffer(*self:* abrain.ANN) → *abrain.ANN.IBuffer*

Return a reference to the neural inputs buffer

neuronAt(*self:* abrain.ANN, *pos:* abrain.Point) → *abrain.ANN.Neuron*

Query an individual neuron

neurons(*self:* abrain.ANN) → *abrain.ANN.Neurons*

Provide read-only access to the underlying neurons

obuffer(*self:* abrain.ANN) → *abrain.ANN.OBuffer*

Return a reference to the neural outputs buffer

perceptron(*self:* abrain.ANN) → bool

Whether this ANN is a perceptron

reset(*self:* abrain.ANN) → None

Resets internal state to null (0)

stats(*self:* abrain.ANN) → *abrain.ANN.Stats*

Return associated stats (connections, depth...)

Rendering tool(s):

abrain.**plotly_render**(*ann:* ANN, *labels: Dict*[Point, *str*] | *None = None*) → Figure

Produce a 3D figure from an artificial neural network

The returned figure can be used to save an interactive html session or a (probably poorly) rendering to e.g. a png file

Supporting types:

class abrain.**Point**

3D coordinate using fixed point notation with 3 decimals

__init__(*self:* abrain.Point, *x: float*, *y: float*, *z: float*)

Create a point with the specified coordinates

Parameters

- **x** (*float*) – x, y, z coordinate
- **y** (*float*) – x, y, z coordinate
- **z** (*float*) – x, y, z coordinate

tuple(*self:* abrain.Point) → tuple[float, float, float]

Return a tuple for easy unpacking in python

class abrain.ANN.**IBuffer**

Specialized, fixed-size buffer for the neural inputs (write-only)

class abrain.ANN.**OBuffer**

Specialized, fixed-size buffer for the neural outputs (read-only)

Underlying types:

class abrain.ANN.**Neuron**

Atomic computational unit of an ANN

class Type

Members:

I : Input (receiving data)

H : Hidden (processing data)

O : Output (producing data)

class Link

An incoming neural connection

src(*self:* abrain.ANN.Neuron.Link) → *abrain.ANN.Neuron*

Return a reference to the source neuron

weight

Connection weight (see attr:*Config.annWeightScale*)

bias

Neural bias

depth

Depth in the neural network

flags

Stimuli-dependent flags (for modularization)

links(*self:* abrain.ANN.Neuron) → list[*abrain.ANN.Neuron.Link*]

Return the list of inputs connections

pos

Position in the 3D substrate

type

Neuron role (see Type)

value

Current activation value

class abrain.ANN.**Neurons**

Wrapper for the C++ neurons container

class abrain.ANN.**Stats**

Contains various statistics about an ANN

axons

Total length of the connections

density

Ratio of expressed connections

depth

Maximal depth of the neural network

dict(*self:* abrain.ANN.Stats) → dict

Return the stats as Python dictionary

edges

Number of connections

hidden

Number of hidden neurons

iterations

H -> H iterations before convergence

1.2.2 Advanced

Composite Pattern-Producing Network

class abrain.**CPPN**

Middle-man between the descriptive *Genome* and the callable *ANN*

class Output

Members:

Weight

LEO

Bias

DIMENSIONS = 3

for the I/O coordinates

INPUTS = 8

Number of inputs

OUTPUTS = 3

Number of outputs

OUTPUTS_LIST = [<Output.Weight: 0>, <Output.LEO: 1>, <Output.Bias: 2>]

The list of output types the CPPN can produce

class Outputs

Output communication buffer for the CPPN

__call__(**args*, ***kwargs*)

Overloaded function.

1. __call__(self: abrain.CPPN, src: abrain.Point, dst: abrain.Point, buffer: abrain.CPPN.Outputs) -> None

Evaluates on provided coordinates and retrieve all outputs

2. __call__(self: abrain.CPPN, src: abrain.Point, dst: abrain.Point, type: abrain.CPPN.Output) -> float

Evaluates on provided coordinates for the requested output

3. __call__(self: abrain.CPPN, src: abrain.Point, dst: abrain.Point, buffer: abrain.CPPN.Outputs, subset: set[abrain.CPPN.Output]) -> None

Evaluates on provided coordinates for the requested outputs

Note: Simplified namespace

---

static functions() → dict[str, Callable[[float], float]]

Return a copy of the C++ built-in function set

static outputs() → *abrain.CPPN.Outputs*

Return a buffer in which the CPPN can store output data

1.3 Function set

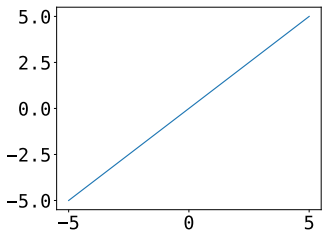
Note: Automatically extracted from sources on Tue Jan 3 17:03:26 CET 2023 for version 0.1b

---

1.3.1 Identity

$$x$$

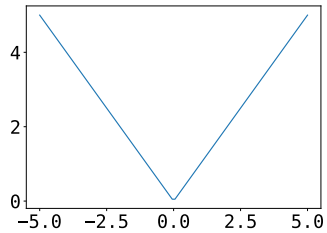
id



1.3.2 Absolute value

$$|x|$$

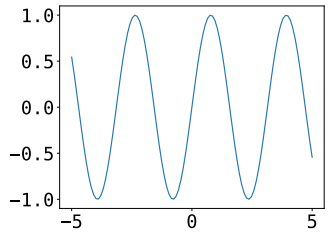
abs



1.3.3 Sinusoidal

$$sin(2x)$$

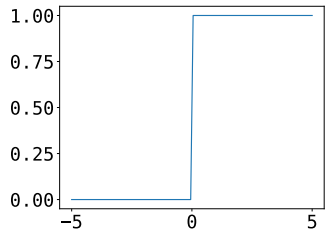
sin



1.3.4 Step function

$$0 \text{ if } x \leq 0$$
$$1 \text{ otherwise}$$

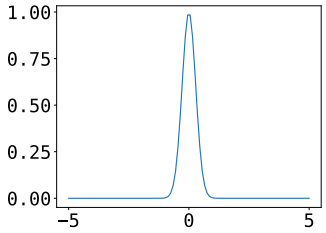
step



1.3.5 Gaussian function

$$e^{-6.25x^2}$$

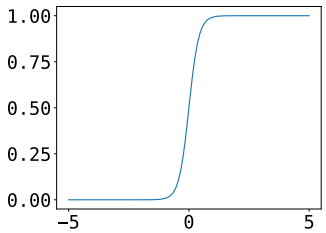
gaus



1.3.6 Soft sigmoid

$$\frac{1}{1+e^{-4.9x}}$$

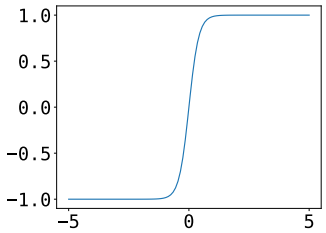
ssgm



1.3.7 Bimodal sigmoid

$$\frac{2}{1+e^{-4.9x}} - 1$$

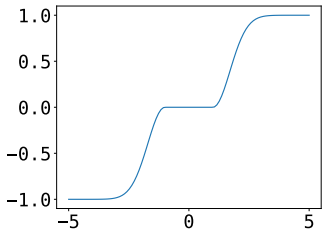
bsgm



1.3.8 Activation function

$$e^{-(x+1)^2} - 1 \text{ if } x - 1$$
$$1 - e^{-(x-1)^2} \text{ if } x1$$
$$0 \text{ otherwise}$$

ssgn



1.4 Miscellaneous

1.4.1 Todo list

Todo: Find a way to reference all warnings/errors (not just mine)

Todo: Missing reference to plotly (hacked something in but fails with 404)

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/abrain/checkouts/latest/doc/src/api/ann.rst, line 5.)

Todo: Implement read-/write-through I/O Buffers

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/abrain/checkouts/latest/doc/src/api/ann.rst, line 6.)

Todo: Examining the CPPN

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/abrain/checkouts/latest/doc/src/usage/advanced/cppn.
line 6.)

Todo: Discuss mutations

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/abrain/checkouts/latest/doc/src/usage/advanced/muta
line 6.)

Todo: Find a way to reference all warnings/errors (not just mine)

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/abrain/checkouts/latest/doc/src/misc.rst,
line 8.)

1.4.2 Build Errors

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

a

INDEX

Symbols

A

B

C

D

E

F

G

H