brainstorm Documentation

Release 0.5

Klaus Greff

January 21, 2016

Contents

Brainstorm: fast, flexible and fun neural networks.

Brainstorm is under active development, so the documentation may be spotty (though we are working on it!). The API documentation is fairly complete, and help may be found on the mailing list.

CHAPTER 1

Installation

1.1 Common Install Notes

A basic requirement to use Brainstorm is Numpy, and we recommend that you make sure that you have a fast BLAS installation which Numpy will use. OpenBLAS is excellent, and can be installed on a Debian family system easily: `sudo apt-get install libopenblas-dev`.

Brainstorm provide a `PyCudaHandler` which can be used to accelerate neural networks using Nvidia GPUs. In order to use it, you need to have CUDA 7.0 or later already installed and setup from https://developer.nvidia.com/cuda-downloads

1.2 Installation variants

When installing from PyPI or GitHub, you can specify the following installation variants to additionally install optional dependencies:

all pycuda tests live_viz draw_net

1.2.1 Ubuntu

Install prerequisites:

```
sudo apt-get install python-dev libhdf5-dev libopenblas-dev
```

Install the latest stable release from PyPI, including all optional dependencies:

```
pip install brainstorm[all]
```

which will additionally install pycuda, scikit-cuda, pygraphviz and bokeh.

To install the latest master branch, you can do:

```
pip install git+git@github.com:IDSIA/brainstorm.git#egg=brainstorm[all]
```

1.2.2 OS X

Instructions coming soon.

1.2.3 Windows

Instructions coming soon.

CHAPTER 2

Walkthrough

In this walkthrough we go over the `cifar10_cnn.py` example in the `examples` directory. This example explains the basics of using brainstorm and helps you get started with your own project.

Prior to running this example, you will need to prepare the CIFAR-10 dataset for brainstorm. This can be done by running the `create_cifar10.py` script in the `data` directory. A detailed description of how to prepare your data for brainstorm can be found in *Data Format*.

We start the `cifar10_cnn.py` example by importing the essential features we will need later:

```
from __future__ import division, print_function, unicode_literals

import os

import h5py

import brainstorm as bs
from brainstorm.data_iterators import Minibatches
from brainstorm.handlers import PyCudaHandler
from brainstorm.initializers import Gaussian
```

Next we set the seed for the global random number generator in Brainstorm. By doing so we are sure that our experiment is reproducible.

```
bs.global_rnd.set_seed(42)
```

Let's now load the CIFAR-10 dataset from the HDF5 file, which we prepared earlier. Next we create a `Minibatches` iterator for the training set and validation set. Here we specify that we want to use a batch size of 100, and that the image data and targets should be named 'default' and 'targets' respectively.

```
data_dir = os.environ.get('BRAINSTORM_DATA_DIR', '../data')
data_file = os.path.join(data_dir, 'CIFAR-10.hdf5')
ds = h5py.File(data_file, 'r')['normalized_split']

getter_tr = Minibatches(100, default=ds['training']['default'][:], targets=ds['training']['targets']
getter_va = Minibatches(100, default=ds['validation']['default'][:], targets=ds['validation']['target
```

In the next step we use a simple helper tool to create two important layers. The first layer is an `Input` layer which takes external inputs named 'default' and 'targets' (these names are the default names used by this tool and can be altered by specifying different names). Every layer in brainstorm has a name, and by default this layer will simply be named 'Input'.

The second layer is a fully-connected layer which produces 10 outputs, and is assigned the name 'Output_projection' by default. In the background, a `SoftmaxCE` layer (named 'Output' by default) is added, which will apply the softmax function and compute the appropriate cross-entropy loss using the targets. At the same time this loss is wired to a `Loss` layer, which marks that this is a value to be minimized.

```
inp, fc = bs.tools.get_in_out_layers('classification', (32, 32, 3), 10)
```

In brainstorm we can wire up our network by using the >> operator. The layer syntaxes below should be self-explanatory. Any layer connected to other layers can now be passed to `from_layer` to create a new network. Note that each layer is assigned a name, which will be used later.

```
(inp >>
    bs.layers.Convolution2D(32, kernel_size=(5, 5), padding=2, name='Conv1') >>
    bs.layers.Pooling2D(type="max", kernel_size=(3, 3), stride=(2, 2)) >>
    bs.layers.Convolution2D(32, kernel_size=(5, 5), padding=2, name='Conv2') >>
    bs.layers.Pooling2D(type="max", kernel_size=(3, 3), stride=(2, 2)) >>
    bs.layers.Convolution2D(64, kernel_size=(5, 5), padding=2, name='Conv3') >>
    bs.layers.Pooling2D(type="max", kernel_size=(3, 3), stride=(2, 2)) >>
    bs.layers.FullyConnected(64, name='FC') >>
    fc)

network = bs.Network.from_layer(fc)
```

We would like to use CUDA to speed up our network training, so we simply set the network's handler to be the `PyCudaHandler`. This line is not needed if we do not have, or do not want to use the GPU – the default handler is the `NumpyHandler`.

```
network.set_handler(PyCudaHandler())
```

In the next line we initialize the weights of our network with a simple dictionary, using the names that were assigned to the layers before. Note that we can use wildcards here!

We specify that:

- For each layer name beginning with 'Conv', the 'W' parameter should be initialized using a Gaussian distribution with std. dev. 0.01, and the 'bias' parameter should be set to zero.
- The parameter 'W' of the layers named 'FC' and 'Output_projection' should be initialized using a Gaussian distribution with std. dev. 0.1. The 'bias' parameter of these layers should be set to zero.

Note that 'Output_projection' is the default name of the final layer created by the helper over which the softmax is computed.

```
network.initialize({'Conv*': {'W': Gaussian(0.01), 'bias': 0},
                    'FC': {'W': Gaussian(0.1), 'bias': 0},
                    'Output_projection': {'W': Gaussian(0.1), 'bias': 0}})
```

Next we create the trainer for which we specify that we would like to use stochastic gradient descent (SGD) with momentum.

Additionally we add a **hook** to the trainer, which will produce a progress bar during each epoch, to keep track of training.

```
trainer = bs.Trainer(bs.training.MomentumStepper(learning_rate=0.01, momentum=0.9))
trainer.add_hook(bs.hooks.ProgressBar())
```

We would like to check the accuracy of the network on our validation set after each epoch. In order to do so we will make use of a hook.

The `SoftmaxCE` layer named 'Output' produces an output named 'probabilities' (the other output it produces is named 'loss'). We tell the `Accuracy` scorer that this output should be used for computing the accuracy using the dotted notation `<layer_name>.<view_type>.<view_name>`.

Next we set the scorers in the trainer and create a `MonitorScores` hook. Here we specify that the trainer will provide access to a data iterator named 'valid_getter', as well as the scorers which will make use of this data.

```
scorers = [bs.scorers.Accuracy(out_name='Output.outputs.probabilities')]
trainer.train_scorers = scorers
trainer.add_hook(bs.hooks.MonitorScores('valid_getter', scorers, name='validation'))
```

Additionally we would like to save the network every time the validation accuracy improves, so we add a hook for this too. We tell the hook that another hook named 'validation' is logging something called 'Accuracy' and that the network should be saved whenever this value is at its maximum.

```
trainer.add_hook(bs.hooks.SaveBestNetwork('validation.Accuracy',
                                          filename='cifar10_cnn_best.hdf5',
                                          name='best weights',
                                          criterion='max'))
```

Finally, we add a hook to stop training after 20 epochs.

```
trainer.add_hook(bs.hooks.StopAfterEpoch(20))
```

We are now ready to train! We provide the trainer with the network to train, the training data iterator, and the validation data iterator (to be used by the hook for monitoring the validation accuracy).

```
trainer.train(network, getter_tr, valid_getter=getter_va)
```

All quantities logged by the hooks are collected by the trainer, which we can examine post training.

```
print("Best validation accuracy:", max(trainer.logs["validation"]["Accuracy"]))
```

CHAPTER 3

Data Format

3.1 Data Shapes

All data passed to a network in Brainstorm by a data iterator must match the template `(T, B, ...)` where `T` is the maximum sequence length and `B` is the number of sequences (or batch size, in other words).

To simplify handling both sequential and non-sequential data, these shapes should also be used when the data is not sequential. In such cases the shape simply becomes `(1, B, ...)`. As an example, the MNIST training images for classification with an MLP should be shaped `(1, 60000, 784)` and the corresponding targets should be shaped `(1, 60000, 1)`.

Data for images/videos should be stored in the `TNHWC` format. For example, the training images for CIFAR-10 should be shaped `(1, 50000, 32, 32, 3)` and the targets should be shaped `(1, 50000, 1)`.

3.2 Example

A network in brainstorm accepts a dictionary of named data items as input. The keys of this dictionary and the shapes of the data should match those which were specified when the network was built.

Consider a simple network built as follows:

```
import numpy as np
from brainstorm import Network, layers

inp = layers.Input({'my_inputs': ('T', 'B', 50),
                    'my_targets': ('T','B', 2)})
hid = layers.FullyConnected(100, name='Hidden')
out = layers.SoftmaxCE(name='Output')
loss = layers.Loss()
inp - 'my_inputs' >> hid >> out
inp - 'my_targets' >> 'targets' - out - 'loss' >> loss
network = Network.from_layer(loss)
```

The same network can be quickly build

Here's how you can provide some data to a network in brainstorm and run a forward pass on it.

3.3 File Format

There is no requirement on how to store the data in `brainstorm`, but we highly recommend the HDF5 format using the h5py library.

It is very simple to create hdf5 files:

```
import h5py
import numpy as np

with h5py.File('demo.hdf5', 'w') as f:
    f['training/input_data'] = np.random.randn(7, 100, 15)
    f['training/targets'] = np.random.randn(7, 100, 2)
    f['training/static_data'] = np.random.randn(1, 100, 4)
```

Having such a file available you can then set-up your data iterator like this:

```
import h5py
import brainstorm as bs

ds = h5py.File('demo.hdf5', 'r')

online_train_iter = bs.Online(**ds['training'])
minibatch_train_iter = bs.Minibatches(100, **ds['training'])
```

These iterators will then provide named data items (a dictionary) to the network with names 'input_data', 'targets' and 'static_data'.

H5py offers many more features, which can be utilized to improve data storage and access such as chunking and compression.

Network

Networks are the central structure in brainstorm. They contain and manage a directed acyclic graph of layers, manage the memory and provide access to all the internal states.

4.1 Creating

There are essentially 4 different ways of creating a network in brainstorm.

1. with the `create_network_from_spec` tool
2. using layer wiring in python (with and without helpers)
3. writing an architecture yourself (advanced)
4. instantiating the layers by hand and setting up a layout (don't do this)

4.1.1 Setting the Handler

If you want to run on CPU in 32bit mode you don't need to do anything. For GPU you need to do:

```
from brainstorm.handlers import PyCudaHandler
net.set_handler(PyCudaHandler())
```

4.1.2 Initializing

Just use the *`initialize()`* method.

4.1.3 Weight and Gradient Modifiers

`net.set_weight_modifiers()`

`net.set_gradient_modifiers()`

4.2 Running

Normally a trainer will run the network for you. But if you want to run a network yourself you have to do this in order:

1. `net.provide_external_data(my_data)`

2. `net.forward_pass()`
3. (optional) `net.backward_pass()`

4.3 Accessing Internals

The recommended way is to always use `net.get(PATH)` because that returns a copy of the buffer in numpy format. If you for some reason want to tamper with the memory that the network is actually using you can get access with: `net.buffer[PATH]`.

4.3.1 Parameters

- `'parameters'` for an array of all parameters
- `'LAYER_NAME.parameters.PARAM_NAME'` for a specific parameter buffer

For the corresponding derivatives calculated during the backward pass:

  - `'gradients'`
  - `'LAYER_NAME.gradients.GRAD_NAME'`

4.3.2 Inputs and Outputs

To access the inputs that have been passed to the network you can use the shortcut `net.get_inputs(IN_NAME)`.

To access inputs and outputs of layers use the following paths:

- `'LAYER_NAME.inputs.IN_NAME'` (often IN_NAME = default)
- `'LAYER_NAME.outputs.OUT_NAME'` (often OUT_NAME = default)

For the corresponding derivatives calculated during the backward pass:

- `'LAYER_NAME.input_deltas.IN_NAME'`
- `'LAYER_NAME.output_deltas.OUT_NAME'`

4.3.3 Internals

Some layers also expose some internal buffers. You can access them with this path:

- `'LAYER_NAME.internals.INTERNAL_NAME'`

4.4 Loading and Saving

`net.save_as_hdf5(filename)`

`net = Network.from_hdf5(filename)`

CHAPTER 5

Layers

This is a construction site.

CHAPTER 6

Trainer

This is a construction site.

CHAPTER 7

Hooks

This is a construction site.

CHAPTER 8

API Documentation

This is a construction site...

8.1 Network

class `brainstorm.structure.network.`**`Network`**(*layers*, *buffer_manager*, *architecture*, *seed=None*, *handler=<brainstorm.handlers.numpy_handler.NumpyHandler object>*)

`backward_pass`()

Perform a backward pass on all provided data and targets.

Note: All the targets to be used during this backward pass have to be passed to the network beforehand using provide_external_data. Also this backward pass depends on the internal state produced by a forward pass. So you have to always run a forward_pass first.

`forward_pass`(*training_pass=False*, *context=None*)

Perform a forward pass on all the provided data.

Note: All the input data to be used during this forward pass have to be passed to the network beforehand using `provide_external_data()`

Parameters

- **`training_pass`** (*Optional[bool]*) – Indicates whether this forward pass belongs to training or not. This might change the behaviour of some layers.
- **`context`** (*Optional[dict]*) – An optional network state as created by net.get_context(). If provided the network will treat this as if it was the the state of the network at the t=-1. This is useful for continuing the computations of a recurrent neural network. Defaults to None.

classmethod **`from_architecture`**(*architecture*)

Create Network instance from given architecture.

Parameters **`architecture`** (*dict*) – JSON serializable Architecture description.

Returns A fully functional Network instance.

Return type *Network*

classmethod **`from_hdf5`**(*filename*)

Load network from HDF5 file.

Parameters **`filename`** (*str*) – Name of the file that the network should be loaded from.

Returns The loaded network.

Return type *Network*

See also:

`save_as_hdf5()`

classmethod **`from_layer`**(*some_layer*)

Create Network instance from a construction layer.

Parameters **`some_layer`** (*brainstorm.construction.ConstructionWrapper*) – Some layer used to wire up an architecture with >>

Returns A fully functional Network instance.

Return type *Network*

`get`(*buffer_path*)

Get a numpy copy of the buffer corresponding to buffer_path.

Examples

```
>>> parameters = net.get('parameters')
>>> outputs = net.get('OutputLayer.outputs.probabilities')
>>> forget_gates = net.get('Lstm.internals.Fb')
```

Parameters **`buffer_path`** (*str*) – A dotted path to the buffer that should be copied and returned.

Returns A numpy array copy of the specified buffer.

Return type numpy.ndarray

Raises `KeyError` –

If no buffer is found for the given path.

`get_context`()

Get the last timestep internal state of this network. (after a forward pass) This can be passed to the forward_pass method as context to continue a batch of sequences.

Returns Internal state of this network at the last timestep.

Return type dict

`get_input`(*input_name*)

Get a numpy copy of one of the named inputs that are currently used.

Parameters **`input_name`** (*str*) – The name of the input that should be retrieved.

Returns A numpy array copy of the specified input.

Return type numpy.ndarray

`get_loss_values`()

Get a dictionary of all the loss values that resulted from a forward pass.

For simple networks with just one loss the dictionary has only a single entry called 'total_loss'.

If there are multiple Loss layers the dictionary will also contain an entry for each Loss layer mapping its name to its loss, and the 'total_loss' entry will contain the sum of all of them.

Returns A dictionary of all loss values that this network produced.

Return type dict[str, float]

initialize(*default_or_init_dict=None*, *seed=None*, ***kwargs*)
Initialize the weights of the network.

Initialization can be specified in three equivalent ways:

1.just a default initializer:

```
>>> net.initialize(Gaussian())
```

Note that this is equivalent to:

```
>>> net.initialize(default=Gaussian())
```

2.by passing a dictionary:

```
>>> net.initialize({'RegularLayer': Uniform(),
...                 'LstmLayer': Gaussian()})
```

3.by using keyword arguments:

```
>>> net.initialize(RegularLayer=Uniform(),
...                LstmLayer=Uniform())
```

All following explanations will be with regards to the dictionary style of initialization, because it is the most general one.

Note: It is not recommended to combine 2. and 3. but if they are, then keyword arguments take precedence.

Each initialization consists of a layer-pattern and that maps to an initializer or a weight-pattern dictionary.

Layer patterns can take the following forms:

1.`{'layer_name': INIT_OR_SUBDICT}` Matches all the weights of the layer named layer_name

2.`{'layer_*': INIT_OR_SUBDICT}` Matches all layers with a name that starts with `layer_` The wild-card `*` can appear at arbitrary positions and even multiple times in one path.

There are two special layer patterns:

3.`{'default': INIT}` Matches all weights that are not matched by any other path-pattern

4.`{'fallback': INIT}` Set a fallback initializer for every weight. It will only be evaluated for the weights for which the regular initializer failed with an InitializationError.

This is useful for initializers that require a certain shape of weights and will not work otherwise. The fallback will then be used for all cases when that initializer failed.

The weight-pattern sub-dictionary follows the same form as the layer- pattern:

1.`{'layer_pattern': {'a': INIT_A, 'b': INIT_B}}`

2.`{'layer_pattern': {'a*': INIT}`

3. `{'layer_pattern': {'default': INIT}`
4. `{'layer_pattern': {'fallback': INIT}`

An initializer can either be a scalar, something that converts to a numpy array of the correct shape or an `Initializer` object. So for example:

```
>>> net.initialize(default=0,
...                RnnLayer={'b': [1, 2, 3, 4, 5]},
...                ForwardLayer=bs.Gaussian())
```

Note: Each view must match exactly one initialization and up to one fallback to be unambiguous. Otherwise the initialization will fail.

You can specify a seed to make the initialization reproducible:

```
>>> net.initialize({'default': bs.Gaussian()}, seed=1234)
```

`provide_external_data` (*data*, *all_inputs=True*)
Provide the data for this network to perform its forward and backward passes on.

Parameters

- **`data`** (*dict*) – A dictionary of the data that will be copied to the outputs of the Input layer.
- **`all_inputs`** (*bool*) – If set to False this method will NOT check that all inputs are provided. Defaults to True.

`save_as_hdf5` (*filename*, *comment=''*)
Save this network as an HDF5 file. The file will contain a description of this network and the parameters.

Parameters

- **`filename`** (*str*) – Name of the file this network should be saved to. All directories have to exist already.
- **`comment`** (*Optional[str]*) – An optional comment that will be saved inside the file.

`set_gradient_modifiers` (*default_or_mod_dict=None*, ***kwargs*)
Install *`ValueModifiers`* in the network to change the gradient.

They can be run manually using `apply_gradient_modifiers()`, but they will also be called by the network after each backward pass.

Gradient modifiers can be set for specific weights in the same way as initializers can, but there is no fallback. (see *`initialize()`* for details)

A modifier can be a ValueModifiers object or a list of them. So for example:

```
>>> net.set_gradient_modifiers(
...     default=bs.value_modifiers.ClipValues(-1, 1)
...     FullyConnectedLayer={'W': [bs.value_modifiers.ClipValues(),
...                                bs.value_modifiers.MaskValues(MASK)]}
...     )
```

Note: The order in which ValueModifiers appear in the list matters, because it is the same order in which they will be executed.

`set_handler` (*new_handler*)
Change the handler of this network.

Examples

Use this to run a network on the GPU using the pycuda:

```
>>> from brainstorm.handlers import PyCudaHandler
>>> net.set_handler(PyCudaHandler())
```

Parameters **new_handler** (brainstorm.handlers.base_handler.Handler) – The new handler this network should use.

set_weight_modifiers(*default_or_mod_dict=None*, ***kwargs*)

Install `ValueModifiers` in the network to change the weights.

They can be run manually using `apply_weight_modifiers()`, but they will also be called by the trainer after each weight update.

Value modifiers can be set for specific weights in the same way initializers can, but there is no fallback. (see `initialize()` for details)

A modifier can be a ValueModifiers object or a list of them. So for example:

```
>>> net.set_weight_modifiers(
...     default=bs.ClipValues(-1, 1)
...     FullyConnectedLayer={'W': [bs.RescaleIncomingWeights(),
...                               bs.MaskValues(my_mask)]}
...     )
```

Note: The order in which ValueModifiers appear in the list matters, because it is the same order in which they will be executed.

8.2 Trainer

class brainstorm.training.trainer.**Trainer**(*stepper*, *verbose=True*)

Trainer objects organize the process of training a network. They can employ different training methods (`Steppers`) and call `Hooks`.

__init__(*stepper*, *verbose=True*)

Create a new Trainer.

Parameters

- **stepper** (*brainstorm.training.steppers.TrainingStepper*) –
- **verbose** (*bool*) –

add_hook(*hook*)

Add a hook to this trainer.

Hooks add a variety of functionality to the trainer and can be called after every specified number of parameter updates or epochs. See documentation for ::class::*Hook* for more details.

Note: During training, hooks will be called in the same order that they were added. This should be kept in mind when using a hook which relies on another hook having been called.

Parameters **hook** (*brainstorm.hooks.Hook*) – Any ::class::*Hook* object that should be called by this trainer.

Raises `ValueError` – If a hook with the same name has already been added.

`train`(*net*, *training_data_iter*, ***named_data_iters*)

Train a network using a data iterator and further named data iterators.

8.3 Tools

`brainstorm.tools.`**`draw_network`**(*network*, *file_name='network.png'*)

Write a diagram for a network to a file.

Parameters

- **`network`** (*brainstorm.structure.Network*) – Network to be drawn.
- **`file_name`** (*Optional[str]*) – Defaults to 'network.png'.

Note: This tool requires the pygraphviz library to be installed.

Raises `ImportError` – If pygraphviz can not be imported.

`brainstorm.tools.`**`evaluate`**(*network*, *iter*, *scorers=()*, *out_name=''*, *targets_name='targets'*, *mask_name=None*)

Evaluate one or more scores for a network.

This tool can be used to evaluate scores of a trained network on test data.

Parameters

- **`network`** (*brainstorm.structure.Network*) – Network to be evaluated.
- **`iter`** (*brainstorm.DataIterator*) – A data iterator which produces the data on which the scores are computed.
- **`scorers`** (*tuple[brainstorm.scorers.Scorer]*) – A list or tuple of Scorers.
- **`out_name`** (*Optional[str]*) – Name of the network output which is scored against the targets.
- **`targets_name`** (*Optional[str]*) – Name of the targets data provided by the data iterator (`iter`).
- **`mask_name`** (*Optional[str]*) – Name of the mask data provided by the data iterator (`iter`).

`brainstorm.tools.`**`extract`**(*network*, *iter*, *buffer_names*)

Apply the network to some data and return the requested buffers.

Batches are returned as a dictionary, with one entry for each requested buffer, with the data in (T, B, ...) order.

Parameters

- **`network`** (*brainstorm.structure.Network*) – Network using which the features should be generated.
- **`iter`** (*brainstorm.DataIterator*) – A data iterator which produces the data on which the features are computed.
- **`buffer_names`** (*list[unicode]*) – Name of the buffer views to be saved (in dotted notation).

Returns dict[unicode, np.ndarray]

`brainstorm.tools.`**`extract_and_save`**(*network*, *iter*, *buffer_names*, *file_name*)

Save the desired buffer values of a network to an HDF5 file.

In particular, this tool can be used to save the predictions of a network on a dataset. In general, any number of internal, input or output buffers of the network can be extracted.

Examples

```
>>> getter = Minibatches(100, default=x_test)
>>> extract_and_save(network,
...                  getter,
...                  ['Output.outputs.predictions',
...                   'Hid1.internals.H'],
...                  'network_features.hdf5')
```

Parameters

- **`network`** (*brainstorm.structure.Network*) – Network using which the features should be generated.
- **`iter`** (*brainstorm.DataIterator*) – A data iterator which produces the data on which the features are computed.
- **`buffer_names`** (*list[unicode]*) – Name of the buffer views to be saved (in dotted notation). See example.
- **`file_name`** (*unicode*) – Name of the hdf5 file (including extension) in which the features should be saved.

`brainstorm.tools.`**`print_network_info`**(*network*)

Print detailed information about the network.

This tools prints the input, output and parameter shapes for all the layers. It also prints the total number of parameters in each layer and in the full network.

Parameters **`network`** (*brainstorm.structure.Network*) – A network for which the details are printed.

`brainstorm.tools.`**`get_in_out_layers`**(*task_type*, *in_shape*, *out_shape*, *data_name='default'*, *targets_name='targets'*, *projection_name=None*, *outlayer_name=None*, *mask_name=None*, *use_conv=None*)

Prepare input and output layers for building a network.

This is a helper function for quickly building networks. It returns an `Input` layer and a projection layer which is a `FullyConnected` or `Convolution2D` layer depending on the shape of the targets. It creates a mask layer if a mask name is provided, and connects it appropriately.

An appropriate layer to compute the matching loss is connected, depending on the task_type:

classification: The projection layer is connected to a SoftmaxCE layer, which receives targets from the input layer. This is suitable for a single-label classification task.

multi-label: The projection layer is connected to a SigmoidCE layer, which receives targets from the input layer. This is suitable for a multi-label classification task.

regression: The projection layer is connected to a SquaredError layer, which receives targets from the input layer. This is suitable for least squares regression.

Note: The projection layer uses parameters, so it should be initialized after network creation. Check argument descriptions to understand how it will be named.

Example

```
>>> from brainstorm import tools, Network, layers
>>> inp, out = tools.get_in_out_layers('classification', 784, 10)
>>> net = Network.from_layer(inp >> layers.FullyConnected(1000) >> out)
```

Parameters

- **`task_type`** (*str*) – one of ['classification', 'regression', 'multi-label']
- **`in_shape`** (*int or tuple[int]*) – Shape of the input data.
- **`out_shape`** (*int or tuple[int]*) – Shape of the network output.
- **`data_name`** (*Optional[str]*) – Name of the input data which will be provided by a data iterator. Defaults to 'default'.
- **`targets_name`** (*Optional[str]*) – Name of the ground-truth target data which will be provided by a data iterator. Defaults to 'targets'.
- **`projection_name`** (*Optional[str]*) – Name for the projection layer which connects to the softmax layer. If unspecified, will be set to `outlayer_name` + '_projection' if `outlayer_name` is provided, and 'Output_projection' otherwise.
- **`outlayer_name`** (*Optional[str]*) – Name for the output layer. If unspecified, named to 'Output'.
- **`mask_name`** (*Optional[str]*) – Name of the mask data which will be provided by a data iterator. Defaults to None.

  The mask is needed if error should be injected only at certain time steps (for sequential data).
- **`use_conv`** (*Optional[bool]*) – Specify whether the projection layer should be convolutional. If true the projection layer will use 1x1 convolutions otherwise it will be fully connected. Default is to autodetect this based on the output shape.

Returns tuple[Layer]

`brainstorm.tools.`**`create_net_from_spec`**(*task_type, in_shape, out_shape, spec, data_name='default', targets_name='targets', mask_name=None, use_conv=None*)

Create a complete network from a spec line like this "F50 F20 F50".

Spec: Capital letters specify the layer type and are followed by arguments to the layer. Supported layers are:

- F : FullyConnected
- R : Recurrent
- L : Lstm
- B : BatchNorm
- D : Dropout
- C : Convolution2D
- P : Pooling2D

Where applicable the optional first argument is the activation function from the set {l, r, s, t} corresponding to 'linear', 'relu', 'sigmoid' and 'tanh' resp.

FullyConnected, Recurrent and Lstm take their size as mandatory arguments (after the optional activation function argument).

Dropout takes the dropout probability as an optional argument.

Convolution2D takes two mandatory arguments: num_filters and kernel_size like this: 'C32:3' or with activation 'Cs32:3' meaning 32 filters with a kernel size of 3x3. They can be followed by 'p1' for padding and/or 's2' for a stride of (2, 2).

Pooling2D takes an optional first argument for the type of pooling: 'm' for max and 'a' for average pooling. The next (mandatory) argument is the kernel size. As with Convolution2D it can be followed by 'p1' for padding and/or 's2' for setting the stride to (2, 2).

Whitespace is allowed everywhere and will be completely ignored.

Examples

The mnist_pi example can be expressed like this: >>> net = create_net_from_spec('classification', 784, 10, ... 'D.2 F1200 D F1200 D') The cifar10_cnn example can be shortened like this: >>> net = create_net_from_spec(... 'classification', (3, 32, 32), 10, ... 'C32:5p2 P3s2 C32:5p2 P3s2 C64:5p2 P3s2 F64')

Parameters

- **`task_type`** (*str*) – one of ['classification', 'regression', 'multi-label']
- **`in_shape`** (*int or tuple[int]*) – Shape of the input data.
- **`out_shape`** (*int or tuple[int]*) – Output shape / nr of classes
- **`spec`** (*str*) – A line describing the network as explained above.
- **`data_name`** (*Optional[str]*) – Name of the input data which will be provided by a data iterator. Defaults to 'default'.
- **`targets_name`** (*Optional[str]*) – Name of the ground-truth target data which will be provided by a data iterator. Defaults to 'targets'.
- **`mask_name`** (*Optional[str]*) – Name of the mask data which will be provided by a data iterator. Defaults to None.

  The mask is needed if error should be injected only at certain time steps (for sequential data).
- **`use_conv`** (*Optional[bool]*) – Specify whether the projection layer should be convolutional. If true the projection layer will use 1x1 convolutions otherwise it will be fully connected. Default is to autodetect this based on the output shape.

Returns The constructed network initialized with DenseSqrtFanInOut for layers with activation function and a simple Gaussian default and fallback.

Return type *brainstorm.structure.network.Network*

8.4 Data Iterators

class `brainstorm.data_iterators.`**`AddGaussianNoise`**(*iter*, *std_dict*, *mean_dict=None*)

Adds Gaussian noise to data generated by another iterator, which must provide named data items (such as Online, Minibatches, Undivided). Only Numpy data is supported, Supports usage of different means and standard deviations for different named data items.

class `brainstorm.data_iterators.`**`AddSaltNPepper`**(*iter*, *prob_dict*, *ratio_dict=None*)

Adds Salt&Pepper noise to data generated by another iterator, which must provide named data items (such as Online, Minibatches, Undivided). Only Numpy data is supported,

Supports usage of different amounts and ratios of salt VS pepper for different named data items.

class `brainstorm.data_iterators.`**`DataIterator`**(*data_shapes*, *length*)

Base class for Data Iterators.

`data_shapes`

dict[str, tuple[int]]

List of input names that this iterator provides.

`length`

int | None

Number of iterations that this iterator will run.

class `brainstorm.data_iterators.`**`Flip`**(*iter*, *prob_dict=None*)

Randomly flip images horizontally. Images are generated by another iterator, which must provide named data items (such as Online, Minibatches, Undivided). Only 5D Numpy data in TNHWC format is supported.

Defaults to flipping the 'default' named data item with a probability of 0.5. Note that the last dimension is flipped, which typically corresponds to flipping images horizontally.

class `brainstorm.data_iterators.`**`Minibatches`**(*batch_size=1*, *shuffle=True*, *cut_according_to='mask'*, ***named_data*)

Minibatch iterator for inputs and targets.

If either a 'mask' is given or some other means of determining sequence length is specified by *cut_according_to*, this iterator also cuts the sequences in each minibatch to their maximum length (which can be less than the maximum length over the whole dataset).

Note: When shuffling is enabled, this iterator only randomizes the order of minibatches, but doesn't re-shuffle instances across batches.

class `brainstorm.data_iterators.`**`MultiHot`**(*iter*, *vocab_size_dict*)

Convert data to multi hot vectors, according to provided vocabulary sizes. If vocabulary size is not provided for some data item, it is yielded as is.

Currently this iterator only supports 3D data.

class `brainstorm.data_iterators.`**`OneHot`**(*iter*, *vocab_size_dict*)

Convert data to one hot vectors, according to provided vocabulary sizes. If vocabulary size is not provided for some data item, it is yielded as is.

Currently this iterator only supports 3D data where the last (right-most) dimension is sized 1.

class `brainstorm.data_iterators.`**`Pad`**(*iter*, *size_dict*, *value_dict=None*)

Pads images equally on all sides. Images are generated by another iterator, which must provide named data items (such as Online, Minibatches, Undivided). Only 5D Numpy data in TNHWC format is supported.

5D data corresponds to sequences of multi-channel images, which is the typical use case. Zero-padding is used unless specified otherwise.

class `brainstorm.data_iterators.`**`RandomCrop`**(*iter*, *shape_dict*)

Randomly crops image data. Images are generated by another iterator, which must provide named data items (such as Online, Minibatches, Undivided). Only 5D Numpy data in TNHWC format is supported.

5D data corresponds to sequences of multi-channel images, which is the typical use case.

class `brainstorm.data_iterators.`**`Undivided`**`(`*`**named_data`*`)`

Processes the entire data in one block (only one iteration).

8.5 Initializers

class `brainstorm.initializers.`**`ArrayInitializer`**`(`*`array`*`)`

Initializes the parameters as the values of the input array.

class `brainstorm.initializers.`**`DenseSqrtFanIn`**`(`*`scale='rel'`*`)`

Initializes the parameters randomly according to a uniform distribution over the interval [-scale/sqrt(n), scale/sqrt(n)] where n is the number of inputs to each unit. Uses scale=sqrt(6) by default which is appropriate for rel units.

When number of inputs and outputs are the same, this is equivalent to using `DenseSqrtFanInOut`.

Scaling:

- rel: sqrt(6)
- tanh: sqrt(3)
- sigmoid: 4 * sqrt(3)
- linear: 1

Parameters **`scale`** (*Optional(float or str)*) – The activation function dependent scaling factor. Can be either float or one of ['rel', 'tanh', 'sigmoid', 'linear']. Defaults to 'rel'.

class `brainstorm.initializers.`**`DenseSqrtFanInOut`**`(`*`scale='rel'`*`)`

Initializes the parameters randomly according to a uniform distribution over the interval [-scale/sqrt(n1+n2), scale/sqrt(n1+n2)] where n1 is the number of inputs to each unit and n2 is the number of units in the current layer. Uses scale=sqrt(12) by default which is appropriate for rel units.

Scaling:

- rel: sqrt(12)
- tanh: sqrt(6)
- sigmoid: 4 * sqrt(6)
- linear: 1

Parameters **`scale`** (*Optional(float or str)*) – The activation function dependent scaling factor. Can be either float or one of ['rel', 'tanh', 'sigmoid', 'linear']. Defaults to 'rel'.

Reference: Glorot, Xavier, and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks" International conference on artificial intelligence and statistics. 2010.

class `brainstorm.initializers.`**`EchoState`**`(`*`spectral_radius=1.0`*`)`

Classic echo state initialization. Creates a matrix with a fixed spectral radius (default=1.0). Spectral radius should be < 1 to satisfy ES-property. Only works for square matrices.

Example

```
>>> net.initialize(default=Gaussian(),
                   Recurrent={'R': EchoState(0.77)})
```

class `brainstorm.initializers.`**`Gaussian`**(*std=0.1*, *mean=0.0*)
Initializes the parameters randomly according to a normal distribution of given mean and standard deviation.

class `brainstorm.initializers.`**`Identity`**(*scale=1.0*, *std=0.01*, *enforce_square=True*)
Initialize a matrix to the (scaled) identity matrix + some noise.

class `brainstorm.initializers.`**`LstmOptInit`**(*input_block=0.0*, *input_gate=0.0*, *forget_gate=0.0*, *output_gate=0.0*)
Used to initialize an LstmOpt layer. This is useful because in an LstmOpt layer all the parameters are concatenated for efficiency.

The parameters (input_block, input_gate, forget_gate, and output_gate) can be scalars or Initializers themselves.

class `brainstorm.initializers.`**`Orthogonal`**(*scale=1.0*)
Orthogonal initialization.

Reference: Saxe, Andrew M., James L. McClelland, and Surya Ganguli. "Exact solutions to the nonlinear dynamics of learning in deep linear neural networks." arXiv preprint arXiv:1312.6120 (2013).

class `brainstorm.initializers.`**`RandomWalk`**(*act_func='linear'*, *scale=None*)
Initializes a (square) weight matrix with the random walk scheme proposed by:

Sussillo, David, and L. F. Abbott. "Random Walk Initialization for Training Very Deep Feedforward Networks." arXiv:1412.6558 [cs, Stat], December 19, 2014. http://arxiv.org/abs/1412.6558.

class `brainstorm.initializers.`**`SparseInputs`**(*sub_initializer*, *connections=15*)
Makes sure every unit only gets activation from a certain number of input units and the rest of the parameters are 0. The connections are initialized by evaluating the passed sub_initializer.

Example

```
>>> net.initialize(FullyConnected=SparseInputs(Gaussian(),
...                                              connections=10))
```

class `brainstorm.initializers.`**`SparseOutputs`**(*sub_initializer*, *connections=15*)
Makes sure every unit is propagating its activation only to a certain number of output units, and the rest of the parameters are 0. The connections are initialized by evaluating the passed sub_initializer.

Example

```
>>> net.initialize(FullyConnected=SparseOutputs(Gaussian(),
                                                 connections=10))
```

class `brainstorm.initializers.`**`Uniform`**(*low=0.1*, *high=None*)
Initializes the parameters randomly according to a uniform distribution over the interval [low, high].

8.6 Hooks

class `brainstorm.hooks.`**`EarlyStopper`**(*log_name*, *patience=1*, *criterion='min'*, *name=None*, *timescale='epoch'*, *interval=1*, *verbose=None*)
Stop the training if a log entry does not improve for some time.

Can stop training when the log entry is at its minimum (such as an error) or maximum (such as accuracy) according to the `criterion` argument.

The `timescale` and `interval` should be the same as those for the monitoring hook which logs the quantity of interest.

Parameters

- **`log_name`** – Name of the log entry to be checked for improvement. It should be in the form <monitorname>.<log_name> where log_name itself may be a nested dictionary key in dotted notation.
- **`patience`** – Number of log updates to wait before stopping training. Default is 1.
- **`criterion`** – Indicates whether training should be stopped when the log entry is at its minimum or maximum value. Must be either 'min' or 'max'. Defaults to 'min'.
- **`name`** (*Optional[str]*) – Name of this monitor. This name is used as a key in the trainer logs. Default is 'EarlyStopper'.
- **`timescale`** (*Optional[str]*) – Specifies whether the Monitor should be called after each epoch or after each update. Default is 'epoch'.
- **`interval`** (*Optional[int]*) – This monitor should be called every `interval` epochs/updates. Default is 1.
- **`verbose`** – bool, optional Specifies whether the logs of this monitor should be printed, and acts as a fallback verbosity for the used data iterator. If not set it defaults to the verbosity setting of the trainer.

Examples

Add a hook to monitor a quantity of interest:

```
>>> scorer = bs.scorers.Accuracy()
>>> trainer.add_hook(bs.hooks.MonitorScores('valid_getter', [scorer],
...                                          name='validation'))
```

Stop training if validation set accuracy does not rise for 10 epochs:

```
>>> trainer.add_hook(bs.hooks.EarlyStopper('validation.Accuracy',
...                                        patience=10,
...                                        criterion='max'))
```

Stop training if loss on validation set does not drop for 5 epochs:

```
>>> trainer.add_hook(bs.hooks.EarlyStopper('validation.total_loss',
...                                        patience=5,
...                                        criterion='min'))
```

class `brainstorm.hooks.`**`InfoUpdater`**(*run*, *name=None*, *timescale='epoch'*, *interval=1*)

Save the information from logs to the Sacred custom info dict

class `brainstorm.hooks.`**`ModifyStepperAttribute`**(*schedule*, *attr_name='learning_rate'*, *timescale='epoch'*, *interval=1*, *name=None*, *verbose=None*)

Modify an attribute of the training stepper.

class `brainstorm.hooks.`**`MonitorLayerDeltas`**(*layer_name*, *name=None*, *timescale='epoch'*, *interval=1*, *verbose=None*)

Monitor some statistics about all the deltas of a layer.

class `brainstorm.hooks.`**`MonitorLayerGradients`**(*layer_name*, *name=None*, *timescale='epoch'*, *interval=1*, *verbose=None*)

Monitor some statistics about all the gradients of a layer.

class `brainstorm.hooks.`**`MonitorLayerInOuts`**(*layer_name*, *name=None*, *timescale='epoch'*, *interval=1*, *verbose=None*)
Monitor some statistics about all the inputs and outputs of a layer.

class `brainstorm.hooks.`**`MonitorLayerParameters`**(*layer_name*, *name=None*, *timescale='epoch'*, *interval=1*, *verbose=None*)
Monitor some statistics about all the parameters of a layer.

class `brainstorm.hooks.`**`MonitorLoss`**(*iter_name*, *name=None*, *timescale='epoch'*, *interval=1*, *verbose=None*)
Monitor the losses computed by the network on a dataset using a given data iterator.

class `brainstorm.hooks.`**`MonitorScores`**(*iter_name*, *scorers*, *name=None*, *timescale='epoch'*, *interval=1*, *verbose=None*)
Monitor the losses and optionally several scores using a given data iterator.

Parameters

- **`iter_name`** (*str*) – name of the data iterator to use (as specified in the train() call)
- **`scorers`** (*List[brainstorm.scorers.Scorer]*) – List of Scorers to evaluate.
- **`name`** (*Optional[str]*) – Name of this monitor. This name is used as a key in the trainer logs. Default is 'MonitorScores'
- **`timescale`** (*Optional[str]*) – Specifies whether the Monitor should be called after each epoch or after each update. Default is 'epoch'.
- **`interval`** (*Optional[int]*) – This monitor should be called every `interval` epochs/updates. Default is 1.
- **`verbose`** – bool, optional Specifies whether the logs of this monitor should be printed, and acts as a fallback verbosity for the used data iterator. If not set it defaults to the verbosity setting of the trainer.

See also:

MonitorLoss: monitor the overall loss of the network.

class `brainstorm.hooks.`**`ProgressBar`**
Adds a progress bar to show the training progress.

class `brainstorm.hooks.`**`SaveBestNetwork`**(*log_name*, *filename=None*, *criterion='max'*, *name=None*, *timescale='epoch'*, *interval=1*, *verbose=None*)
Check to see if the specified log entry is at it's best value and if so, save the network to a specified file.

Can save the network when the log entry is at its minimum (such as an error) or maximum (such as accuracy) according to the `criterion` argument.

The `timescale` and `interval` should be the same as those for the monitoring hook which logs the quantity of interest.

Parameters

- **`log_name`** – Name of the log entry to be checked for improvement. It should be in the form <monitorname>.<log_name> where log_name itself may be a nested dictionary key in dotted notation.
- **`filename`** – Name of the HDF5 file to which the network should be saved.
- **`criterion`** – Indicates whether training should be stopped when the log entry is at its minimum or maximum value. Must be either 'min' or 'max'. Defaults to 'min'.

- **name** (*Optional[str]*) – Name of this monitor. This name is used as a key in the trainer logs. Default is 'SaveBestNetwork'.
- **timescale** (*Optional[str]*) – Specifies whether the Monitor should be called after each epoch or after each update. Default is 'epoch'.
- **interval** (*Optional[int]*) – This monitor should be called every `interval` epochs/updates. Default is 1.
- **verbose** – bool, optional Specifies whether the logs of this monitor should be printed, and acts as a fallback verbosity for the used data iterator. If not set it defaults to the verbosity setting of the trainer.

Examples

Add a hook to monitor a quantity of interest:

```
>>> scorer = bs.scorers.Accuracy()
>>> trainer.add_hook(bs.hooks.MonitorScores('valid_getter', [scorer],
...                                          name='validation'))
```

Check every epoch and save the network if validation accuracy rises:

```
>>> trainer.add_hook(bs.hooks.SaveBestNetwork('validation.Accuracy',
...                                            filename='best_acc.h5',
...                                            criterion='max'))
```

Check every epoch and save the network if validation loss drops:

```
>>> trainer.add_hook(bs.hooks.SaveBestNetwork('validation.total_loss',
    ...                                        filename='best_loss.h5',
...                                            criterion='min'))
```

class `brainstorm.hooks.`**`SaveLogs`**(*filename*, *name=None*, *timescale='epoch'*, *interval=1*)
Periodically Save the trainer logs dictionary to an HDF5 file. Default behavior is to save every epoch.

class `brainstorm.hooks.`**`SaveNetwork`**(*filename*, *name=None*, *timescale='epoch'*, *interval=1*)
Periodically save the weights of the network to the given file. Default behavior is to save the network after every training epoch.

class `brainstorm.hooks.`**`StopAfterEpoch`**(*max_epochs*, *name=None*, *timescale='epoch'*, *interval=1*, *verbose=None*)
Stop the training after a specified number of epochs.

Parameters

- **max_epochs** (*int*) – The number of epochs to train.
- **name** (*Optional[str]*) – Name of this monitor. This name is used as a key in the trainer logs. Default is 'StopAfterEpoch'.
- **timescale** (*Optional[str]*) – Specifies whether the Monitor should be called after each epoch or after each update. Default is 'epoch'.
- **interval** (*Optional[int]*) – This monitor should be called every `interval` epochs/updates. Default is 1.
- **verbose** – bool, optional Specifies whether the logs of this monitor should be printed, and acts as a fallback verbosity for the used data iterator. If not set it defaults to the verbosity setting of the trainer.

class `brainstorm.hooks.`**`StopAfterThresholdReached`**(*log_name*, *threshold*, *criterion='min'*, *name=None*, *timescale='epoch'*, *interval=1*, *verbose=None*)

Stop the training if a log entry reaches the given threshold

Can stop training when the log entry becomes sufficiently small (such as an error) or sufficiently large (such as accuracy) according to the threshold.

Parameters

- **`log_name`** – Name of the log entry to be checked for improvement. It should be in the form <monitorname>.<log_name> where log_name itself may be a nested dictionary key in dotted notation.
- **`threshold`** – The threshold value to reach
- **`criterion`** – Indicates whether training should be stopped when the log entry is at its minimum or maximum value. Must be either 'min' or 'max'. Defaults to 'min'.
- **`name`** (*Optional[str]*) – Name of this monitor. This name is used as a key in the trainer logs. Default is 'StopAfterThresholdReached'.
- **`timescale`** (*Optional[str]*) – Specifies whether the Monitor should be called after each epoch or after each update. Default is 'epoch'.
- **`interval`** (*Optional[int]*) – This monitor should be called every `interval` epochs/updates. Default is 1.
- **`verbose`** – bool, optional Specifies whether the logs of this monitor should be printed, and acts as a fallback verbosity for the used data iterator. If not set it defaults to the verbosity setting of the trainer.

Examples

Stop training if validation set accuracy is at least 97 %:

```
>>> trainer.add_hook(StopAfterThresholdReached('validation.Accuracy',
...                                            threshold=0.97,
...                                            criterion='max'))
```

Stop training if loss on validation set goes below 0.2:

```
>>> trainer.add_hook(StopAfterThresholdReached('validation.total_loss',
...                                            threshold=0.2,
...                                            criterion='min'))
```

class `brainstorm.hooks.`**`StopOnNan`**(*logs_to_check=()*, *check_parameters=True*, *check_training_loss=True*, *name=None*, *timescale='epoch'*, *interval=1*, *verbose=None*)

Stop the training if infinite or NaN values are found in parameters.

This hook can also check a list of logs for invalid values.

Parameters

- **`logs_to_check`** (*Optional[list, tuple]*) – A list of trainer logs to check in dotted notation. Defaults to ().
- **`check_parameters`** (*Optional[bool]*) – Indicates whether the parameters should be checked for NaN. Defaults to True.

- **name** (*Optional[str]*) – Name of this monitor. This name is used as a key in the trainer logs. Default is 'StopOnNan'.
- **timescale** (*Optional[str]*) – Specifies whether the Monitor should be called after each epoch or after each update. Default is 'epoch'.
- **interval** (*Optional[int]*) – This monitor should be called every `interval` epochs/updates. Default is 1.
- **verbose** – bool, optional Specifies whether the logs of this monitor should be printed, and acts as a fallback verbosity for the used data iterator. If not set it defaults to the verbosity setting of the trainer.

class `brainstorm.hooks.`**`StopOnSigQuit`**(*name=None*, *timescale='epoch'*, *interval=1*, *verbose=None*)

Stop training after the next call if it received a SIGQUIT (Ctrl +).

This hook makes it possible to exit the training loop and continue with the rest of the program execution.

Parameters

- **name** (*Optional[str]*) – Name of this monitor. This name is used as a key in the trainer logs. Default is 'StopOnSigQuit'.
- **timescale** (*Optional[str]*) – Specifies whether the Monitor should be called after each epoch or after each update. Default is 'epoch'.
- **interval** (*Optional[int]*) – This monitor should be called every `interval` epochs/updates. Default is 1.
- **verbose** – bool, optional Specifies whether the logs of this monitor should be printed, and acts as a fallback verbosity for the used data iterator. If not set it defaults to the verbosity setting of the trainer.

8.7 Value Modifiers

class `brainstorm.value_modifiers.`**`ClipValues`**(*low=-1.0*, *high=1.0*)

Clips (limits) the weights to be between low and high. Defaults to low=-1 and high=1.

Should be added to the network via the set_weight_modifiers method like so:

>> net.set_weight_modifiers(RnnLayer={'HR': ClipValues()})

See Network.set_weight_modifiers for more information on how to control which weights to affect.

class `brainstorm.value_modifiers.`**`ConstrainL2Norm`**(*limit*)

Constrains the L2 norm of the incoming weights to every neuron/unit to be less than or equal to a limit. If the L2 norm for any unit exceeds the limit, the weights are rescaled such that the squared L2 norm equals the limit. Ignores Biases.

Should be added to the network via the set_weight_modifiers method like so:

>> net.set_weight_modifiers(RnnLayer={'HX': ConstrainL2Norm()})

See Network.set_weight_modifiers for more information on how to control which weights to affect.

class `brainstorm.value_modifiers.`**`FreezeValues`**(*weights=None*)

Prevents the weights from changing at all.

If the weights argument is left at None it will remember the first weights it sees and resets them to that every time.

Should be added to the network via the set_constraints method like so: >> net.set_constraints(RnnLayer={'HR': FreezeValues()}) See Network.set_constraints for more information on how to control which weights to affect.

class `brainstorm.value_modifiers.`**`L1Decay`**(*factor*)
Applies L1 weight decay.

New gradients = gradients + factor * sign(parameters)

class `brainstorm.value_modifiers.`**`L2Decay`**(*factor*)
Applies L2 weight decay.

New gradients = gradients + factor * parameters

class `brainstorm.value_modifiers.`**`MaskValues`**(*mask*)
Multiplies the weights elementwise with the mask.

This can be used to clamp some of the weights to zero.

Should be added to the network via the set_weight_modifiers method like so:

>> net.set_weight_modifiers(RnnLayer={'HR': MaskValues(M)})

See Network.set_weight_modifiers for more information on how to control which weights to affect.

class `brainstorm.value_modifiers.`**`ValueModifier`**
ValueModifiers can be installed in a `Network` to affect either the parameters or the gradients.

8.8 Scorers

8.9 Handler

class `brainstorm.handlers.base_handler.`**`Handler`**
Abstract base class for all handlers.

This base is used mainly to ensure a common interface and provide documentation for derived handlers. When implementing new methods one should adhere to the naming scheme. Most mathematical operations should have a suffix or suffixes indicating the shapes of inputs it expects:

s for scalar, *v* for vector (a 2D array with at least dimension equal to 1), *m* for matrix (a 2D array), *t* for tensor (which means arbitrary shape, synonym for *array*).

Note that these shapes are not checked by each handler itself. However, the DebugHandler can be used to perform these checks to ensure that operations are not abused.

`dtype`
Data type that this handler works with.

`context`
Context which may be used by this handler for operation.

`EMPTY`
An empty array matching this handler's type.

`rnd`
A random state maintained by this handler.

`array_type`
The type of array object that this handler works with.

`__describe__`()

Returns a description of this object. That is a dictionary containing the name of the class as `@type` and all members of the class. This description is json-serializable.

If a sub-class of Describable contains non-describable members, it has to override this method to specify how it should be described.

Returns Description of this object

Return type dict

`__new_from_description__`(*description*)

Creates a new object from a given description.

If a sub-class of Describable contains non-describable fields, it has to override this method to specify how they should be initialized from their description.

Parameters **`description`** (*dict*) – description of this object

Returns A new instance of this class according to the description.

`abs_t`(*a*, *out*)

Compute the element-wise absolute value.

Parameters

- **`a`** (array_type) – Array whose absolute values are to be computed.
- **`out`** (array_type) – Array into which the output is placed. Must have the same shape as `a`.

Returns None

`add_into_if`(*a*, *out*, *cond*)

Add element of *a* to element of *out* if corresponding element in *cond* is non-zero.

Parameters

- **`a`** (array_type) – Array whose elements (might) be added to *out*.
- **`out`** (array_type) – Output array, whose values might be increased by values from *a*.
- **`cond`** (array_type) – The condition array. Only those entries from *a* are added into *out* whose corresponding *cond* elements are non-zero.

Returns None

`add_mv`(*m*, *v*, *out*)

Add a matrix to a vector with broadcasting.

Add an (M, N) matrix to a (1, N) or (M, 1) vector using broadcasting such that the output is (M, N).

Parameters

- **`m`** (array_type) – The first array to be added. Must be 2D.
- **`v`** (array_type) – The second array to be added. Must be 2D with at least one dimension of size 1 and the other dimension matching the corresponding size of `m`.
- **`out`** (array_type) – Array into which the output is placed. Must have the same shape as `m`.

Returns None

`add_st`(*s*, *t*, *out*)

Add a scalar to each element of a tensor.

Parameters

- **`s`** (dtype) – The scalar value to be added.

- **t** (array_type) – The array to be added.
- **out** (array_type) – Array into which the output is placed. Must have the same shape as `t`.

Returns None

add_tt (*a*, *b*, *out*)
Add two tensors element-wise,

Parameters

- **a** (array_type) – First array.
- **b** (array_type) – Second array.
- **out** (array_type) – Array into which the output is placed. Must have the same shape as `a` and `b`.

Returns None

allocate (*shape*)
Allocate new memory with given shape but arbitrary content.

Parameters **shape** (*tuple[int]*) – Shape of the array.

Returns New array with given shape.

Return type object

avgpool2d_backward_batch (*inputs*, *window*, *outputs*, *padding*, *stride*, *in_deltas*, *out_deltas*)
Computes the gradients for 2D average-pooling on a batch of images.

Parameters

- **inputs** (array_type) –
- **window** (*tuple[int]*) –
- **outputs** (array_type) –
- **padding** (*int*) –
- **stride** (*tuple[int]*) –
- **in_deltas** (array_type) –
- **out_deltas** (array_type) –

Returns None

avgpool2d_forward_batch (*inputs*, *window*, *outputs*, *padding*, *stride*)
Performs 2D average-pooling on a batch of images.

Parameters

- **inputs** (array_type) –
- **window** (*tuple[int]*) –
- **outputs** (array_type) –
- **padding** (*int*) –
- **stride** (*tuple[int]*) –
- **argmax** (array_type) –

Returns None

binarize_v (*v*, *out*)

Convert a column vector into a matrix of one-hot row vectors.

Usually used to convert class IDs into one-hot vectors. Therefore, *out[i, j] = 1*, if j equals v[i, 0] *out[i, j] = 0*, otherwise.

Note that *out* must have enough columns such that all indices in `v` are valid.

Parameters

- **v** (array_type) – Column vector (2D array with a single column).
- **out** (array_type) – Matrix (2D array) into which the output is placed. The number of rows must be the same as `v` and number of columns must be greater than the maximum value in `v`.

Returns None

broadcast_t (*a*, *axis*, *out*)

Broadcast the given axis of an array by copying elements.

This function provides a numpy-broadcast-like operation for the the dimension given by axis. E.g. for axis=3 an array with shape (2, 3, 4, 1) may be broadcasted to shape (2, 3, 4, 5), by copying all the elements 5 times.

Parameters

- **a** (array_type) – Array whose elements should be broadcasted. The dimension corresponding to axis must be of size 1.
- **axis** (*int*) – the axis along which to broadcast
- **out** (array_type) – Array into which the output is placed. Must have same the number of dimensions as *a*. Only the dimension corresponding to axis can differ from *a*.

Returns None

clip_t (*a*, *a_min*, *a_max*, *out*)

Clip (limit) the values in an array.

Given an interval, values outside the interval are clipped to the interval edges. For example, if an interval of [0, 1] is specified, values smaller than 0 become 0, and values larger than 1 become 1.

Parameters

- **a** (array_type) – Array containing the elements to clip.
- **a_min** (dtype) – Minimum value.
- **a_max** (dtype) – Maximum value.
- **out** (array_type) – Array into which the output is placed. Must have the same shape as `a`.

Returns None

conv2d_backward_batch (*inputs*, *weights*, *padding*, *stride*, *in_deltas*, *out_deltas*, *weight_deltas*, *bias_deltas*)

Computes the gradients for a 2D convolution on a batch of images.

Parameters

- **inputs** (array_type) –
- **weights** (array_type) –
- **padding** (*int*) –
- **stride** (*tuple[int]*) –

- **in_deltas** (array_type) –
- **out_deltas** (array_type) –
- **weight_deltas** (array_type) –
- **bias_deltas** (array_type) –

Returns None

conv2d_forward_batch (*inputs*, *weights*, *bias*, *outputs*, *padding*, *stride*)
Performs a 2D convolution on a batch of images.

Parameters

- **inputs** (array_type) –
- **weights** (array_type) –
- **bias** (array_type) –
- **outputs** (array_type) –
- **padding** (*int*) –
- **stride** (*tuple[int]*) –

Returns None

copy_to (*src*, *dest*)
Copy the contents of one array to another.

Both source and destination arrays must be of this handler's supported type and have the same shape.

Parameters

- **dest** (array_type) – Destination array.
- **src** (array_type) – Source array.

Returns None

copy_to_if (*src*, *dest*, *cond*)
Copy element of 'src' to element of 'dest' if cond is not equal to 0.

Parameters

- **src** (array_type) – Source array whose elements (might) be copied into *dest*.
- **dest** (array_type) – Destination array.
- **cond** (array_type) – The condition array. Only those *src* elements get copied to *dest* whose corresponding *cond* elements are non-zero.

Returns None

create_from_numpy (*arr*)
Create a new array with the same entries as a Numpy array.

Parameters **arr** (*numpy.ndarray*) – Numpy array whose elements should be used to fill the new array.

Returns New array with same shape and entries as the given Numpy array.

Return type *array_type*

divide_mv (*m*, *v*, *out*)
Divide a matrix by a vector.

Divide a (M, N) matrix element-wise by a (1, N) vector using broadcasting such that the output is (M, N).

Parameters

- **a** (array_type) – First array (dividend). Must be 2D.
- **b** (array_type) – Second array (divisor). Must be 2D with at least one dimension of size 1 and second dimension matching the corresponding size of `m`.
- **out** (array_type) – Array into which the output is placed. Must have the same shape as `m`.

Returns None

divide_tt (*a*, *b*, *out*)
Divide two tensors element-wise.

Parameters

- **a** (array_type) – First array (dividend).
- **b** (array_type) – Second array (divisor). Must have the same shape as `a`.
- **out** (array_type) – Array into which the output is placed. Must have the same shape as `a` and `b`.

Returns None

dot_add_mm (*a*, *b*, *out*, *transa=False*, *transb=False*)
Multiply two matrices and add to a matrix.

Only 2D arrays (matrices) are supported.

Parameters

- **a** (array_type) – First matrix.
- **b** (array_type) – Second matrix. Must have compatible shape to be right-multiplied with `a`.
- **out** (array_type) – Array into which the output is added. Must have correct shape for the product of the two matrices.

Returns None

dot_mm (*a*, *b*, *out*, *transa=False*, *transb=False*)
Multiply two matrices.

Only 2D arrays (matrices) are supported.

Parameters

- **a** (array_type) – First matrix.
- **b** (array_type) – Second matrix. Must have compatible shape to be right-multiplied with `a`.
- **out** (array_type) – Array into which the output is placed. Must have correct shape for the product of the two matrices.

Returns None

fill (*mem*, *val*)
Fill an array with a given value.

Parameters

- **`mem`** (array_type) – Array to be filled.
- **`val`** (dtype) – Value to fill.

Returns None

`fill_gaussian`(*mean*, *std*, *out*)
Fill an array with values drawn from a Gaussian distribution.

Parameters

- **`mean`** (*float*) – Mean of the Gaussian Distribution.
- **`std`** (*float*) – Standard deviation of the Gaussian distribution.
- **`out`** (array_type) – Target array to fill with values.

Returns None

`fill_if`(*mem*, *val*, *cond*)
Set the elements of *mem* to *val* if corresponding *cond* element is non-zero.

Parameters

- **`mem`** (array_type) – Array to be filled.
- **`val`** (dtype) – The scalar which the elements of *mem* (might) be set to.
- **`cond`** (array_type) – The condition array. Only those *mem* elements are set to *val* whose corresponding *cond* elements are non-zero.

Returns None

`generate_probability_mask`(*mask*, *probability*)
Fill an array with zeros and ones.

Fill an array with zeros and ones such that the probability of an element being one is equal to `probability`.

Parameters

- **`mask`** (array_type) – Array to will be filled.
- **`probability`** (*float*) – Probability of an element of `mask`
- **`equal to one.`** (*being*) –

Returns None

`get_numpy_copy`(*mem*)
Return a copy of the given data as a numpy array.

Parameters **`mem`** (array_type) – Source array to be copied.

Returns Numpy array with same content as mem.

Return type numpy.ndarray

`index_m_by_v`(*m*, *v*, *out*)
Get elements from a matrix using indices from a vector.

`v` and `out` must be column vectors of the same size. Elements from the matrix `m` are copied using the indices given by a column vector. From row *i* of the matrix, the element from column *v[i, 0]* is copied to out, such that *out[i, 0] = m[i, v[i, 0]]*.

Note that *m* must have enough columns such that all indices in `v` are valid.

Parameters

- **m** (array_type) – Matrix (2D array) whose elements should be copied.
- **v** (array_type) – Column vector (2D array with a single column) whose values are used as indices into `m`. The number of rows must be the same as `m`.
- **out** (array_type) – Array into which the output is placed. It's shape must be the same as `v`.

Returns None

is_fully_finite(*a*)
Check if all entries of the array are finite (no nans or infs).

Parameters **a** (array_type) – Input array to check.

Returns True if there are no infs or nans, False otherwise.

Return type bool

log_t(*a*, *out*)
Compute the element-wise natural logarithm.

The natural logarithm log is the inverse of the exponential function, so that *log(exp(x)) = x*.

Parameters

- **a** (array_type) – Array whose logarithm is to be computed.
- **out** (array_type) – Array into which the output is placed. Must have the same shape as `a`.

Returns None

maxpool2d_backward_batch(*inputs*, *window*, *outputs*, *padding*, *stride*, *argmax*, *in_deltas*, *out_deltas*)
Computes the gradients for 2D max-pooling on a batch of images.

Parameters

- **inputs** (array_type) –
- **window** (*tuple[int]*) –
- **outputs** (array_type) –
- **padding** (*int*) –
- **stride** (*tuple[int]*) –
- **argmax** (array_type) –
- **in_deltas** (array_type) –
- **out_deltas** (array_type) –

Returns None

maxpool2d_forward_batch(*inputs*, *window*, *outputs*, *padding*, *stride*, *argmax*)
Performs a 2D max-pooling on a batch of images.

Parameters

- **inputs** (array_type) –
- **window** (*tuple[int]*) –
- **outputs** (array_type) –
- **padding** (*int*) –
- **stride** (*tuple[int]*) –

- **argmax** (array_type) –

Returns None

merge_tt (*a*, *b*, *out*)

Merge arrays a and b along their last axis.

Parameters

- **a** (array_type) – Array to be merged.
- **b** (array_type) – Array to be merged.
- **out** (array_type) – Array into which the output is placed.

Returns None

modulo_tt (*a*, *b*, *out*)

Take the modulo between two arrays elementwise. (out = a % b)

Parameters

- **a** (array_type) – First array (dividend).
- **b** (array_type) – Second array (divisor). Must have the same shape as *a*.
- **out** (array_type) – Array into which the remainder is placed. Must have the same shape as `a` and `b`.

Returns None

mult_add_st (*s*, *t*, *out*)

Multiply a scalar with each element of a tensor and add to a tensor.

Parameters

- **s** (dtype) – The scalar value to be multiplied.
- **t** (array_type) – The array to be multiplied.
- **out** (array_type) – Array into which the product is added. Must have the same shape as `t`.

Returns None

mult_add_tt (*a*, *b*, *out*)

Multiply two tensors element-wise and add to a tensor.

Parameters

- **a** (array_type) – First array.
- **b** (array_type) – Second array. Must have the same shape as `a`.
- **out** (array_type) – Array into which the output is added. Must have the same shape as `a` and `b`.

Returns None

mult_mv (*m*, *v*, *out*)

Multiply a matrix with a vector.

Multiply an (M, N) matrix with a (1, N) or (M, 1) vector using broadcasting such that the output is (M, N). Also allows the "vector" to have the same dimension as the matrix in which case it behaves the same as `mult_tt()`.

Parameters

- **m** (array_type) – The first array. Must be 2D.
- **v** (array_type) – The second array, to be multiplied with `a`. Must be 2D with at least one dimension of size 1 and the other dimension matching the corresponding size of `m`.
- **out** (array_type) – Array into which the output is placed. Must have the same shape as `m`.

Returns None

mult_st (*s*, *t*, *out*)
Multiply a scalar with each element of a tensor.

Parameters

- **s** (dtype) – The scalar value to be multiplied.
- **t** (array_type) – The array to be multiplied.
- **out** (array_type) – Array into which the output is placed. Must have the same shape as `t`.

Returns None

mult_tt (*a*, *b*, *out*)
Multiply two tensors of the same shape element-wise.

Parameters

- **a** (array_type) – First array.
- **b** (array_type) – Second array. Must have the same shape as `a`.
- **out** (array_type) – Array into which the output is placed. Must have the same shape as `a` and `b`.

Returns None

ones (*shape*)
Allocate new memory with given shape and filled with ones.

Parameters **shape** (*tuple[int]*) – Shape of the array.

Returns New array with given shape filled with ones.

Return type object

rel (*x*, *y*)
Compute the rel (rectified linear) function.

y = rel(x) = max(0, x)

Parameters

- **x** (array_type) – Input array.
- **y** (array_type) – Output array.

Returns None

rel_deriv (*x*, *y*, *dy*, *dx*)
Backpropagate derivatives through the rectified linear function.

Parameters

- **x** (array_type) – Inputs to the rel function. This argument is not used and is present only to conform with other activation functions.
- **y** (array_type) – Outputs of the rel function.
- **dy** (array_type) – Derivatives with respect to the outputs.

- **dx** (array_type) – Array in which the derivatives with respect to the inputs are placed.

Returns None

set_from_numpy (*mem*, *arr*)
Set the content of an array from a given numpy array.

Parameters

- **mem** (array_type) – Destination array that should be set.
- **arr** (*numpy.ndarray*) – Source numpy array.

Returns None

sigmoid (*x*, *y*)
Compute the sigmoid function.

y = sigmoid(x) = 1 / (1 + exp(-x)) :param x: Input array. :type x: array_type :param y: Output array. :type y: array_type

Returns None

sigmoid_deriv (*x*, *y*, *dy*, *dx*)
Backpropagate derivatives through the sigmoid function.

Parameters

- **x** (array_type) – Inputs to the sigmoid function. This argument is not used and is present only to conform with other activation functions.
- **y** (array_type) – Outputs of the sigmoid function.
- **dy** (array_type) – Derivatives with respect to the outputs.
- **dx** (array_type) – Array in which the derivatives with respect to the inputs are placed.

Returns None

sign_t (*a*, *out*)
Compute an element-wise indication of the sign of a number.

Output has the value 1.0 if an element is positive, 0 if it is zero, and -1.0 if it is negative.

Parameters

- **a** (array_type) – Array whose sign is to be computed.
- **out** (array_type) – Array into which the output is placed. Must have the same shape as `a`.

Returns None

softmax_m (*m*, *out*)
Compute the softmax function over last dimension of a matrix.

Parameters

- **m** (array_type) – Input array.
- **out** (array_type) – Output array.

Returns None

split_add_tt (*x*, *out_a*, *out_b*)
Split array x along the last axis and add the parts to out_i.

Parameters

- **x** (array_type) – Array to be split.

- **out_a** (array_type) – Array to which 1st part of x is added.
- **out_b** (array_type) – Array to which 2nd part of x is added.

Returns None

sqrt_t (*a*, *out*)
Compute the positive square-root of an array, element-wise.

Parameters

- **a** (array_type) – Array whose square root is to be computed.
- **out** (array_type) – Array into which the output is placed. Must have the same shape as `a`.

Returns None

subtract_mv (*m*, *v*, *out*)
Subtract a vector from a matrix with broadcasting.

Parameters

- **m** (array_type) – The first array. Must be 2D.
- **v** (array_type) – The second array, to be subtracted from `a`. Must be 2D with at least one dimension of size 1 and second dimension matching the corresponding size of `m`.
- **out** (array_type) – Array into which the output is placed. Must have the same shape as `m`.

Returns None

subtract_tt (*a*, *b*, *out*)
Subtract a tensor from another element-wise.

Parameters

- **a** (array_type) – First array.
- **b** (array_type) – Second array, to be subtracted from `a`. Must have the same shape as `a`.
- **out** (array_type) – Array into which the output (`a - b`) is placed. Must have the same shape as `a` and `b`.

Returns None

sum_t (*a*, *axis*, *out*)
Sum the elements of an array along a given axis.

If axis is None, the sum is computed over all elements of the array. Otherwise, it is computed along the specified axis.

Note: Only 1D and 2D arrays are currently supported.

Parameters

- **a** (array_type) – Array to be summed.
- **axis** (*int*) – Axis over which the summation should be done.
- **out** (array_type) – Array into which the output is placed.

Returns None

tanh (*x*, *y*)

Compute the tanh (hyperbolic tangent) function.

y = tanh(x) = (e^z - e^-z) / (e^z + e^-z)

Parameters

- **x** (array_type) – Input array.
- **y** (array_type) – Output array.

Returns None

tanh_deriv (*x*, *y*, *dy*, *dx*)

Backpropagate derivatives through the tanh function.

Parameters

- **x** (array_type) – Inputs to the tanh function. This argument is not used and is present only to conform with other activation functions.
- **y** (array_type) – Outputs of the tanh function.
- **dy** (array_type) – Derivatives with respect to the outputs.
- **dx** (array_type) – Array in which the derivatives with respect to the inputs are placed.

Returns None

zeros (*shape*)

Allocate new memory with given shape and filled with zeros.

Parameters **shape** (*tuple[int]*) – Shape of the array.

Returns New array with given shape filled with zeros.

Return type object

8.10 Describables

This module provides the core functionality for describing objects.

8.10.1 Description

In brainstorm most objects can be converted into a so called description using the `get_description()` function. A description is a JSON-serializable data structure that contains all *static* information to re-create that object using the `create_from_description()` function. It does not, however, contain any *dynamic* information. This means that an object created from a description is in the same state as if it had been freshly instantiated, without any later modifications of its internal state.

The descriptions for the basic types `int`, `float`, `bool`, `str`, `list` and `dict` are these values themselves. Other objects need to inherit from `Describable` and their description is always a `dict` containing the `'@type': 'ClassName'` key possibly along with other properties.

8.10.2 Conversion to and from descriptions

`brainstorm.describable.`**`create_from_description`** (*description*)

Instantiate a new object from a description.

Parameters **description** (*dict*) – A description of the object.

Returns A new instance of the described object

`brainstorm.describable.`**`get_description`**(*this*)

Create a JSON-serializable description of this object.

This description can be used to create a new instance of this object by calling *`create_from_description()`*.

Parameters **`this`** (Describable) – An object to be described. Must either be a basic datatype or inherit from Describable.

Returns A JSON-serializable description of this object.

8.10.3 Describable Base Class

class `brainstorm.describable.`**`Describable`**

Base class for all objects that can be described and initialized from a description.

Derived classes can specify the *`__undescribed__`* field to prevent certain attributes from being described. This field can be either a set of attribute names, or a dictionary mapping attribute names to their initialization value (used when a new object is created from the description).

Derived classes can also specify an *`__default_values__`* dict. This dict allows for omitting certain attributes from the description if their value is equal to that default value.

`__undescribed__` = {}

Set or dict of attributes that should not be part of the description. If specified as a dict, then these attributes are initialized to the specified values.

`__default_values__` = {}

Dict of attributes with their corresponding default values. They will only be part of the description if their value differs from their default value

`__init_from_description__`(*description*)

Subclasses can override this to provide additional initialization when created from a description.

This method will be called AFTER the object has already been created from the description.

Parameters **`description`** (*dict*) – the description of this object

`__describe__`()

Returns a description of this object. That is a dictionary containing the name of the class as `@type` and all members of the class. This description is json-serializable.

If a sub-class of Describable contains non-describable members, it has to override this method to specify how it should be described.

Returns Description of this object

Return type dict

classmethod **`__new_from_description__`**(*description*)

Creates a new object from a given description.

If a sub-class of Describable contains non-describable fields, it has to override this method to specify how they should be initialized from their description.

Parameters **`description`** (*dict*) – description of this object

Returns A new instance of this class according to the description.

`__describe__`()

Returns a description of this object. That is a dictionary containing the name of the class as `@type` and all members of the class. This description is json-serializable.

If a sub-class of Describable contains non-describable members, it has to override this method to specify how it should be described.

Returns Description of this object

Return type dict

`__init_from_description__`(*description*)

Subclasses can override this to provide additional initialization when created from a description.

This method will be called AFTER the object has already been created from the description.

Parameters **`description`** (*dict*) – the description of this object

classmethod `__new_from_description__`(*description*)

Creates a new object from a given description.

If a sub-class of Describable contains non-describable fields, it has to override this method to specify how they should be initialized from their description.

Parameters **`description`** (*dict*) – description of this object

Returns A new instance of this class according to the description.

CHAPTER 9

Internals

9.1 Overview

Like many deep learning frameworks, a key philosophy behind Brainstorm is modularity.

- Key abstractions in brainstorm are **layers** that are connected to form a **network** and define which computations should be performed and what their parameters are.
- Layers are implemented in terms of operations provided by the **handler**, which makes independent from how and on which device the operations are actually implemented (CPU or GPU).
- The **trainer** coordinates applying an iterative optimization algorithm (**stepper**) with other concerns like monitoring and early stopping (done via so called **hooks**).
- **data iterators** control the way the data is loaded, augmented, and chunked up.
- an important design feature is that each of these parts can easily be changed by user-code.
- So custom layers for example can simply be shared as a single file and imported by others, so there is no need to change brainstorm to support them.

9.1.1 Handler

A *`Handler`* implements basic mathematical operations on arrays of a certain type and can allocate memory. For example, the `NumpyHandler` allocates CPU memory and performs computations on the CPU mainly through Numpy. The `PyCudaHandler` on the other hand allocates GPU memory and implements its operations with `pycuda`. This intermediate abstraction allows changing the low-level implementation of operations without breaking the layers or any other part of brainstorm. Our Handler operations have a more restricted feature set than numpy, which makes it easier to implement new handlers, but of course this comes at the price of less convenience in implementing layers. For the future we plan to provide also a handler that CuDNN4, and one that builds upon NervanaGPU.

9.1.2 Layers

A `Layer` defines how to compute a forward pass and a backward pass and which parameters it uses to do so. Layers don't own any of the memory they use themselves. The memory layout and managing is done by the network using the `BufferManager`. So a layer only needs to specify how it interacts with other layers and how much memory it needs. This design makes implementing a new layer rather simple, and allows for automated testing of layers.

Note that the `...LayerImpl` objects are different from the objects used during network creation with the `>>` operator. The latter are used only to define an architecture, which is then in turn used to instantiate the actual layer objects and create the network.

9.1.3 BufferManager

The BufferManager is part of the Network and is responsible for allocating and structuring the memory needed by all the layers. It computes and allocates the required total amount of memory based on the current batch-size and sequence-length. This memory is then chunked-up, reshaped and structured distributed according to the memory _layout of the network.

9.1.4 Network

The network is a central abstraction that coordinates the layers and the buffer manager, provides an interface for the user to inspect the internals. It is designed to allow for comprehensive inspection and ease of use. Each network holds an ordered dictionary of layers (`net.layers`) sorted topologically. There has to be always exactly one Input layer which is also called `Input`, and the connections between layers have to form a connected acyclic graph.

The network gives access to the internal memory layout created by the BufferManager through `net.buffer`. This interface can be used to inspect(and influence) exactly what is happening inside the network. Note that if using a GPU handler the returned memory views will be on the device. If you want to get a copy of any buffer, use the `net.get(BUFFER_PATH)` method.

The network provides a powerful initialize method that allows to control precisely how all the parameters are initialized. Furthermore it keeps track of so called weight modifiers and gradient modifiers. These simple modifications that will regularly be applied to the weights (e.g. constrain their L2 norm) or the gradient (e.g. clip their values). Similar to initializers these can be added to individually to individual parameters via the `net.set_weight_modifiers` and `net.set_gradint_modifiers` methods.

The Network usually operates in three steps:

1. `net.provide_external_data()` is called first to populate the buffers of the Input layer with values for the input data and the targets (and possibly others)
2. `net.forward_pass()` runs the forward pass of all layers populating all the output buffers
3. `net.backward_pass()` performs a backward pass, thus calculating all the deltas and gradients

9.1.5 Trainer

The Trainer is the second central abstraction in Brainstorm which coordinates the training and collects logs. It provides the Network with data from DataIterators and has its parameters updated by an optimization Stepper. Furthermore it calls the Hooks, collects their logs and prints them if appropriate.

When `trainer.train` is called, the trainer will iterate over the data it gets from the training data iterator and provide it to the network. It will then call the Stepper which is responsible for updating the parameters based on the gradients. The trainer also keeps list of Hooks which will be called regularly (on a timescale they can specify) and which can interact with the training. Some by monitoring certain quantities, some by saving the network or stopping the training if some condition occurs. Hooks that monitor something report their logs back to the trainer, which prints, collects, and stores them.

9.1.6 Stepper

Is responsible for updating the parameters of a Network. Can be Stochastic Gradient Descent or RMSProp.

9.1.7 Hooks

- Interact with the training.

- Will be called by the trainer based on their timescale and interval. (“epoch” or “update”, and arbitrary interval)
- Different stereotypical kinds of Hooks: 1. Monitors 2. Savers 3. Stoppers 4. Visualizers

9.1.8 Scorers

9.1.9 ValueModifiers

9.1.10 Initializers

Let’s say we have some data (usually some *input* data and perhaps some *target* data), and we know what mathematical operations we’d like to do on it (compute the outputs, adjust the parameters etc.). We tell Brainstorm about these operations by building a directed acyclic graph of layers. Brainstorm then computes the memory that is needed for the required operations, and slices it up into parts needed by each layer creating a **memory layout**. This memory can now be allocated, and each layer gets access to the parts of the memory relevant for it (where its parameters, inputs, outputs etc. live).

2. A **Buffer Manager** allocates the memory (using the handler), and decides how to prepare the memory layout for efficient processing. Again, this works independently from how the layers or handlers are implemented.

This means that one can now easily write a new Handler which uses a different array type, and performs basic mathematical operations differently. The rest of Brainstorm simply works with it. Similarly, one may chose a different way of allocating memory given a description of layers, without affecting the rest of the components.

Such a design implies that important components of the library can be improved individually and in principle *hot-swapped* as needed. If a new numerical computations library is available, one can easily integrate it into Brainstorm as long as it satisfies some basic requirement. If we realize that there is a better way to allocate memory for a certain network connectivity, this can be easily be incorporated.

Here you can find some details about the internal design of brainstorm. This description is however very much work in progress and by no means complete.

9.2 Conventions

When naming the extra properties of layers, a couple of conventions should be met. A property name should:

- be a valid python identifier
- be in snake_case (lowercase with underscores)
- be called `size` if it controls the size of the layer directly
- be `activation` if it controls the activation function

9.3 Architecture

Network architecture is a dictionary mapping layer names to their properties. There are two special properties:

1. `@type`: a string that specifies the class of the layer
2. `@outgoing_connections`: a dictionary mapping the named outputs of the layer to a lists of named inputs of other layers it connects to. For specifying the input we use dotted notation: `LAYER_NAME.INPUT_NAME`. If the name of the input is `default` it can be omitted.

There can be more properties that will be passed on to the layer class when instantiating them.

A basic example showcasing most features

```
architecture = {
    'Input': {
        '@type': 'Input',
        '@outgoing_connections': {
            'default': ['hidden_layer'],
            'targets': ['output_layer.targets']
        },
        'out_shapes': {
            'default': ('T', 'B', 784),
            'targets': ('T', 'B', 1)
        }
    },
    'hidden_layer': {
        '@type': 'FullyConnected',
        '@outgoing_connections': {
            'default': ['output_projection']
        },
        'activation': 'rel',
        'size': 100
    },
    'output_projection': {
        '@type': 'FullyConnected',
        '@outgoing_connections': {
            'default': ['output_layer']
        },
        'activation': 'linear',
        'size': (10,)
    },
    'output_layer': {
        '@type': 'SoftmaxCE'
        '@outgoing_connections': {
            'loss': ['loss_layer']
        },
    },
    'loss_layer': {
        '@outgoing_connections': {},
        '@type': 'Loss',
        'importance': 1.0
    }
}
```

9.4 Layout

Layouts describe how the memory for the network should be arranged.

9.4.1 Buffer Types

There are three types of buffers that distinguish the way memory should scale with the input dimensions:

0. **Constant Size:** These buffers do not change their size at all. The most common usecase are parameters.

1. **Batch Sized:** These buffers scale with the number of sequences in the current batch. An example would be sequence-wise targets, i.e. there is should be one target per sequence.
2. **Time Sized:** Scale with *both* batch-size and sequence-length. Most input data, hidden activations and internal states fall into that category.

For all types of buffers we specify their size and shape only for the "feature" dimension, i.e. the dimension that does not change. So only for constant size buffer like parameters the specified size and shape will actually be equal to the final size and shape of the buffer. For time sized buffer there would be two additional dimension added to the front of the final buffer shape. So if we specify the inputs should be of shape `(28, 28)`, then the input buffer will be of shape `(T, B, 28, 28)` where `B` is the number of sequences, and `T` is their (max) length.

The BufferManager for a network will allocate one big buffer for each type, and resize them in response to input-sizes. That big chunk of memory is also split up into a tree of named buffers according to the *layout*.

9.4.2 Shape templates

When implementing a layer there are many places where a shape of a buffer needs to be specified. But the size of the time-size and the batch-size are both unknown at implementation time. So we use so called *shape-templates* to specify which buffer type you are expecting. So for example for feature size of three these would be the templatesfor the 3 buffer types:

- `(3,)` => Constant size buffer
- `('B', 3)` => Batch sized buffer
- `('T', 'B', 3)` => time sized buffer

Here `'T'` is the placeholder for the sequence-length, and `'B'` is the placeholder for the batchsize.

If the feature size is also unknown (e.g. when specifying the input and output shapes of a layer) then `'F'` can be used as a placeholder for those.

9.4.3 The Layout Specification

The layout specification is a tree of nested dictionaries, containing two types of nodes: view-nodes and array-nodes that describe what entries the buffer views should have, how big the arrays at the leaves are, and their position in the big buffer are. Each node has to have an `@type` field that is either `BufferView` or `array`.

9.4.4 View-Nodes

View-nodes will be turned into BufferView objects by the BufferManager. Each of them is a dictionary and has to contain a and a `@index` entry. The entries not starting with an `@` are the child-nodes. The `@index` entry specifies the order among siblings.

A node can also contain a `@slice` entry, if the buffers of all child nodes are of the same type and contiguous in memory. The corresponding array will then be available as `_full_buffer` member in the resulting BufferView object.

Example node:

```
{
    '@type': 'BufferView',
    '@index': 0,

    'child_A': {...},
    'child_B': {...}
}
```

Another example including the optional `@slice`:

```
{
    '@type': 'BufferView',
    '@index': 2,
    '@slice': (0, 50, 110),

    'only_child': {...}
}
```

9.4.5 Array-Nodes

Array-nodes will be turned into arrays (exact type depends on the handler), by the buffer manager. Array-Nodes are also dictionaries but they *must have* a `@slice` and a `@shape` entry, and they cannot have any children. Like view-nodes, an array-node needs an `@index` entry to specify the order among its siblings.

The `@slice` should be a tuple of two integers `(start, stop)`. Where `start` and `stop` specify which slice of the big buffer this array is a view of points to.

The `@shape` entry is a shape-template and describes the dimensionality of the array.

If an array-node has a shape of a type 2 buffer (time-scaled) it can (optionally) contain a `@context_size` entry. This determines how many extra time steps are added to the end of that buffer. Notice that this way you can access the context slices using negative indexing.

Example leaf for a 4 times 5 weight matrix:

```
{'@index': 1, '@slice': (5, 25),  '@shape': (4, 5)}
```

Example leaf for the output of a layer with 10 hidden units:

```
{'@index': 1, '@slice': (19, 29), '@shape': ('T', 'B', 10)}
```

9.4.6 Full Layout Example

We use the following network as an example here:

```
mse = MseLayer(10)
inputs = Input(out_shapes={'input_data': (4,), 'targets':(10,)})
inputs - 'input_data' >> Rnn(5) >> FullyConnected(10, name='OutLayer') >> 'net_out' - mse
inputs - 'targets' >> 'targets' - mse
net = build_net(mse)
```

```
```

```
joint_layout = {
    'Input': {
        '@type': 'BufferView',
        '@index': 0,
        'inputs': {'@type': 'BufferView', '@index': 0},
        'outputs': {
            '@type': 'BufferView',
            '@index': 1,
            '@slice': (0, 14),
            'input_data': {'@type': 'array', '@index': 0, '@slice': (0, 4), '@shape': ('T', 'B', 4)},
            'targets':    {'@type': 'array','@index': 1, '@slice': (10, 14), '@shape': ('T', 'B', 4)
        }},
        'parameters': {'@type': 'BufferView', '@index': 2},
        'internals': {'@type': 'BufferView', '@index': 3},
    },
    'Rnn': {
        '@type': 'BufferView',
        '@index': 1,
        'inputs': {
            '@type': 'BufferView',
            '@index': 0,
            '@slice': (0, 4),
            'default': {'@type': 'array', '@index': 0, '@slice': (0, 4), '@shape': ('T', 'B', 4), '@c
        },
        'outputs': {
            '@type': 'BufferView',
            '@index': 1,
            '@slice': (14, 19),
            'default': {'@type': 'array', '@index': 0, '@slice': (14, 19), '@shape': ('T', 'B', 5),
        },
        'parameters': {
            '@type': 'BufferView',
            '@index': 2,
            '@slice': (0, 50),
            'W': {'@type': 'array', '@index': 0, '@slice': (0, 20),  '@shape': (4, 5)},
            'R': {'@type': 'array', '@index': 1, '@slice': (20, 45), '@shape': (5, 5)},
            'b': {'@type': 'array', '@index': 2, '@slice': (45, 50), '@shape': (5,  )}
        },
        'internals': {
            '@type': 'BufferView',
            '@index': 3,
            '@slice': (30, 35),
            'Ha': {'@type': 'array', '@index': 0, '@slice': (30, 35), '@shape': ('T', 'B', 5), '@cont
        },
    },
    'Out': {
        '@type': 'BufferView',
        '@index': 2,
        'inputs': {
            '@type': 'BufferView',
            '@index': 0,
            '@slice': (14, 19),
            'default': {'@type': 'array', '@index': 0, '@slice': (14, 19), '@shape': ('T', 'B', 5)}
        },
        'outputs': {
            '@type': 'BufferView',
            '@index': 1,
            '@slice': (19, 29),
            'default': {'@type': 'array', '@index': 0, '@slice': (19, 29), '@shape': ('T', 'B', 10)}
        },
        'parameters': {
            '@type': 'BufferView',
            '@index': 2,
            '@slice': (50, 110),
            'W': {'@type': 'array', '@index': 0, '@slice': (50, 100),  '@shape': (5, 10)},
            'b': {'@type': 'array', '@index': 1, '@slice': (100, 110), '@shape': (10,  )}
        },
        'internals': {
            '@type': 'BufferView',
            '@index': 3,
            '@slice': (35, 45),
            'Ha': {'@type': 'array', '@index': 0, '@slice': (35, 55), '@shape': ('T', 'B', 10)}
        }
    },
    'Mse': {
        '@type': 'BufferView',
        '@index': 3,
        'inputs': {
            '@type': 'BufferView',
            '@index': 0,
            'net_out': {'@type': 'array', '@index': 0, '@slice': (19, 29), '@shape': ('T', 'B', 10)},
            'targets': {'@type': 'array', '@index': 1, '@slice': (10, 14), '@shape': ('T', 'B', 10)}
        },
        'outputs': {
            '@type': 'BufferView',
            '@index': 1,
            '@slice': (29, 30),
            'default': {'@type': 'array', '@index': 0, '@slice': (29, 30), '@shape': ('T', 'B', 1)}
        },
        'parameters': {'@type': 'BufferView', '@index': 2},
        'internals': {'@type': 'BufferView', '@index': 3},
    }}
}
```

CHAPTER 10

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:

10.1 Types of Contributions

10.1.1 Report Bugs

Report bugs at https://github.com/IDSIA/brainstorm/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.

10.1.2 Fix Bugs

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

10.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with "feature" is open to whoever wants to implement it.

10.1.4 Write Documentation

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

10.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/IDSIA/brainstorm/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 :)

10.2 Get Started!

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

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

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

3. Create a branch for local development:

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

Now you can make your changes locally.

4. When you're done making changes, check that your changes pass style and unit tests, including testing other Python versions with tox:

```
    $ tox
```

To get tox, just pip install it.

5. 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
```

6. Submit a pull request through the GitHub website.

10.3 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, and 3.3, and for PyPy. Check https://travis-ci.org/IDSIA/brainstorm under pull requests for active pull requests or run the `tox` command and make sure that the tests pass for all supported Python versions.

10.4 Tips

To run a subset of tests:

```
$ py.test test/test_brainstorm.py
```

CHAPTER 11

Credits

11.1 Development Lead

- Klaus Greff
- Rupesh Srivastava

11.2 Contributors

- Thomas Unterthiner
- Julian Zilly
- Sjoerd van Steenkiste

CHAPTER 12

History

12.1 0.5 (2015-12-01)

12.1.1 Changed Behaviour

- examples now run on CPU by default
- added `brainstorm.tools.shuffle_data` and `brainstorm.tools.split` to help with data preparation
- `SigmoidCE` and `SquaredDifference` layers now outputs a loss for each dimension instead of summing over features.
- `SquaredDifference` layer does no longer scale by one half.
- Added a `SquaredLoss` layer that computes half the squared difference and has an interface that is compatible with the `SigmoidCE` and `SigmoidCE` layers.
- Output *probabilities* renamed to *predictions* in `SigmoidCE` and `SigmoidCE` layers.

12.1.2 New Features

- added a *use_conv* option to `brainstorm.tools.create_net_from_spec`
- added *criterion* option to `brainstorm.hooks.EarlyStopper` hook
- added `brainstorm.tools.get_network_info` function that returns information about the network as a string
- added `brainstorm.tools.extract` function that applies a network to some data and saves a set of requested buffers.
- `brainstorm.layers.mask` layer now supports masking individual features
- added `brainstorm.hooks.StopAfterThresholdReached` hook

12.1.3 Improvements

- EarlyStopper now works for any timescale and interval
- Recurrent, Lstm, Clockwork, and ClockworkLstm layers now accept inputs of arbitrary shape by implicitly flattening them.

- several fixes to make building the docs easier
- some performance improvements of NumpyHandler operations `binarize_t` and `index_m_by_v`
- sped up tests
- several improvements to installation scripts

12.1.4 Bugfixes

- fixed *sqrt* operation for `PyCudaHandler`. This should fix problems with BatchNormalization on GPU.
- fixed a bug for task_type='regression' in `brainstorm.tools.get_in_out_layers` and `brainstorm.tools.create_net_from_spec`
- removed defunct name argument from input layer
- fixed a crash when applying `brainstorm.hooks.SaveBestNetwork` to *rolling_training* loss
- various minor fixes of the `brainstorm.hooks.BokehVisualizer`
- fixed a problem with `sum_t` operation in `brainstorm.handlers.PyCudaHandler`
- fixed a blocksize problem in convolutional and pooling operations in `brainstorm.handlers.PyCudaHandler`

12.2 0.5b0 (2015-10-25)

- First release on PyPI.

CHAPTER 13

Feedback

If you have any suggestions or questions about **brainstorm** feel free to email us at mailstorm@googlegroups.com.

If you encounter any errors or problems with **brainstorm**, please let us know! Open an Issue at the GitHub http://github.com/IDSIA/brainstorm main repository.

Python Module Index