paramz Documentation

Max Zwiessele

Sep 02, 2018

Contents

CHAPTER 1

paramz package

1.1 Subpackages

1.1.1 paramz.core package

Submodules

paramz.core.constrainable module

class Constrainable(*name*, *default_constraint=None*, **a*, ***kw*)

Bases: *paramz.core.indexable.Indexable*

constrain(*transform*, *warning=True*, *trigger_parent=True*)

Parameters

- **transform** – the *paramz.transformations.Transformation* to constrain the this parameter to.
- **warning** – print a warning if re-constraining parameters.

Constrain the parameter to the given *paramz.transformations.Transformation*.

constrain_bounded(*lower*, *upper*, *warning=True*, *trigger_parent=True*)

Parameters

- **upper** (*lower,*) – the limits to bound this parameter to
- **warning** – print a warning if re-constraining parameters.

Constrain this parameter to lie within the given range.

constrain_fixed(*value=None*, *warning=True*, *trigger_parent=True*)

Constrain this parameter to be fixed to the current value it carries.

This does not override the previous constraints, so unfixing will restore the constraint set before fixing.

Parameters **warning** – print a warning for overwriting constraints.

constrain_negative(*warning=True*, *trigger_parent=True*)

Parameters **warning** – print a warning if re-constraining parameters.

Constrain this parameter to the default negative constraint.

constrain_positive(*warning=True*, *trigger_parent=True*)

Parameters **warning** – print a warning if re-constraining parameters.

Constrain this parameter to the default positive constraint.

fix(*value=None*, *warning=True*, *trigger_parent=True*)
Constrain this parameter to be fixed to the current value it carries.

This does not override the previous constraints, so unfixing will restore the constraint set before fixing.

Parameters **warning** – print a warning for overwriting constraints.

unconstrain(**transforms*)

Parameters **transforms** – The transformations to unconstrain from.

remove all *paramz.transformations.Transformation* transformats of this parameter object.

unconstrain_bounded(*lower*, *upper*)

Parameters **upper** (*lower,*) – the limits to unbound this parameter from

Remove (lower, upper) bounded constrain from this parameter/

unconstrain_fixed()
This parameter will no longer be fixed.

If there was a constraint on this parameter when fixing it, it will be constraint with that previous constraint.

unconstrain_negative()
Remove negative constraint of this parameter.

unconstrain_positive()
Remove positive constraint of this parameter.

unfix()
This parameter will no longer be fixed.

If there was a constraint on this parameter when fixing it, it will be constraint with that previous constraint.

is_fixed

paramz.core.gradcheckable module

class Gradcheckable(**a*, ***kw*)
Bases: *paramz.core.pickleable.Pickleable*, *paramz.core.parentable.Parentable*

Adds the functionality for an object to be gradcheckable. It is just a thin wrapper of a call to the highest parent for now. TODO: Can be done better, by only changing parameters of the current parameter handle, such that object hierarchy only has to change for those.

checkgrad(*verbose=0*, *step=1e-06*, *tolerance=0.001*, *df_tolerance=1e-12*)
Check the gradient of this parameter with respect to the highest parent's objective function. This is a three point estimate of the gradient, wiggling at the parameters with a stepsize step. The check passes if either the ratio or the difference between numerical and analytical gradient is smaller then tolerance.

Parameters

- **verbose** (*bool*) – whether each parameter shall be checked individually.
- **step** (*float*) – the stepsize for the numerical three point gradient estimate.
- **tolerance** (*float*) – the tolerance for the gradient ratio or difference.
- **df_tolerance** (*float*) – the tolerance for df_tolerance

Note: The *dF_ratio* indicates the limit of accuracy of numerical gradients. If it is too small, e.g., smaller than 1e-12, the numerical gradients are usually not accurate enough for the tests (shown with blue).

paramz.core.index_operations module

class ParameterIndexOperations(*constraints=None*)

Bases: object

This object wraps a dictionary, whos keys are _operations_ that we'd like to apply to a parameter array, and whose values are np integer arrays which index the parameter array appropriately.

A model instance will contain one instance of this class for each thing that needs indexing (i.e. constraints, ties and priors). Parameters within the model constain instances of the ParameterIndexOperationsView class, which can map from a 'local' index (starting 0) to this global index.

Here's an illustration:

```
#=======================================================================
model : 0 1 2 3 4 5 6 7 8 9
key1: 4 5
key2: 7 8

param1: 0 1 2 3 4 5
key1: 2 3
key2: 5

param2: 0 1 2 3 4
key1: 0
key2: 2 3
#=======================================================================
```

The views of this global index have a subset of the keys in this global (model) index.

Adding a new key (e.g. a constraint) to a view will cause the view to pass the new key to the global index, along with the local index and an offset. This global index then stores the key and the appropriate global index (which can be seen by the view).

See also: ParameterIndexOperationsView

add(*prop*, *indices*)

clear()

copy()

indices()

items()

properties()

properties_dict_for(*index*)
Return a dictionary, containing properties as keys and indices as index Thus, the indices for each constraint, which is contained will be collected as one dictionary

Example: let properties: 'one':[1,2,3,4], 'two':[3,5,6]

```
>>> properties_dict_for([2,3,5])
{'one':[2,3], 'two':[3,5]}
```

properties_for(*index*)
Returns a list of properties, such that each entry in the list corresponds to the element of the index given.

Example: let properties: 'one':[1,2,3,4], 'two':[3,5,6]

```
>>> properties_for([2,3,5])
[['one'], ['one', 'two'], ['two']]
```

remove(*prop*, *indices*)

shift_left(*start*, *size*)

shift_right(*start*, *size*)

update(*parameter_index_view*, *offset=0*)

size

class ParameterIndexOperationsView(*param_index_operations*, *offset*, *size*)
Bases: object

add(*prop*, *indices*)

clear()

copy()

indices()

items()

properties()

properties_dict_for(*index*)
Return a dictionary, containing properties as keys and indices as index Thus, the indices for each constraint, which is contained will be collected as one dictionary

Example: let properties: 'one':[1,2,3,4], 'two':[3,5,6]

```
>>> property_dict_for([2,3,5])
{'one':[2,3], 'two':[3,5]}
```

properties_for(*index*)
Returns a list of properties, such that each entry in the list corresponds to the element of the index given.

Example: let properties: 'one':[1,2,3,4], 'two':[3,5,6]

```
>>> properties_for([2,3,5])
[['one'], ['one', 'two'], ['two']]
```

remove(*prop*, *indices*)

shift_left(*start*, *size*)

shift_right(*start*, *size*)

update(*parameter_index_view*, *offset=0*)

size

combine_indices(*arr1*, *arr2*)

extract_properties_to_index(*index*, *props*)

index_empty(*index*)

remove_indices(*arr*, *to_remove*)

paramz.core.indexable module

class Indexable(*name*, *default_constraint=None*, **a*, ***kw*)
Bases: *paramz.core.nameable.Nameable*, *paramz.core.updateable.Updateable*

Make an object constrainable with Priors and Transformations.

TODO: Mappings!! (As in ties etc.)

Adding a constraint to a Parameter means to tell the highest parent that the constraint was added and making sure that all parameters covered by this object are indeed conforming to the constraint.

`constrain` and `unconstrain` are main methods here

add_index_operation(*name*, *operations*)
Add index operation with name to the operations given.

raises: attribute error if operations exist.

remove_index_operation(*name*)

paramz.core.lists_and_dicts module

class ArrayList
Bases: `list`

List to store ndarray-likes in. It will look for 'is' instead of calling __eq__ on each element.

index(*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

class IntArrayDict(*default_factory=None*)
Bases: `collections.defaultdict`

Default will be self._default, if not set otherwise

class ObserverList
Bases: `object`

A list which containts the observables. It only holds weak references to observers, such that unbound observers dont dangle in memory.

add(*priority*, *observer*, *callble*)
Add an observer with priority and callble

flush()
Make sure all weak references, which point to nothing are flushed (deleted)

remove(*priority*, *observer*, *callble*)
Remove one observer, which had priority and callble.

intarray_default_factory()

paramz.core.nameable module

class Nameable(*name*, **a*, ***kw*)
Bases: *paramz.core.gradcheckable.Gradcheckable*

Make an object nameable inside the hierarchy.

hierarchy_name(*adjust_for_printing=True*)
return the name for this object with the parents names attached by dots.

Parameters **adjust_for_printing** (*bool*) – whether to call `adjust_for_printing` on the names, recursively

name
The name of this object

adjust_name_for_printing(*name*)
Make sure a name can be printed, alongside used as a variable name.

paramz.core.observable module

class Observable(**args*, ***kwargs*)
Bases: `object`

Observable pattern for parameterization.

This Object allows for observers to register with self and a (bound!) function as an observer. Every time the observable changes, it sends a notification with self as only argument to all its observers.

add_observer(*observer*, *callble*, *priority=0*)
Add an observer *observer* with the callback *callble* and priority *priority* to this observers list.

change_priority(*observer*, *callble*, *priority*)

notify_observers(*which=None*, *min_priority=None*)
Notifies all observers. Which is the element, which kicked off this notification loop. The first argument will be self, the second *which*.

Note: notifies only observers with priority p > min_priority!

Parameters **min_priority** – only notify observers with priority > min_priority if min_priority is None, notify all observers in order

remove_observer(*observer*, *callble=None*)
Either (if callble is None) remove all callables, which were added alongside observer, or remove callable *callble* which was added alongside the observer *observer*.

set_updates(*on=True*)

paramz.core.observable_array module

class ObsAr(**a*, ***kw*)
Bases: `numpy.ndarray`, *paramz.core.pickleable.Pickleable*, *paramz.core.observable.Observable*

An ndarray which reports changes to its observers.

Warning: ObsAr tries to not ever give back an observable array itself. Thus, if you want to preserve an ObsAr you need to work in memory. Let *a* be an ObsAr and you want to add a random number *r* to it. You need to make sure it stays an ObsAr by working in memory (see numpy for details):

```
a[:] += r
```

The observers can add themselves with a callable, which will be called every time this array changes. The callable takes exactly one argument, which is this array itself.

copy()
Make a copy. This means, we delete all observers and return a copy of this array. It will still be an ObsAr!

values
Return the ObsAr underlying array as a standard ndarray.

paramz.core.parameter_core module

Core module for parameterization. This module implements all parameterization techniques, split up in modular bits.

Observable: Observable Pattern for patameterization

class OptimizationHandlable(*name*, *default_constraint=None*, **a*, ***kw*)
Bases: *paramz.core.constrainable.Constrainable*

This enables optimization handles on an Object as done in GPy 0.4.

. . . *_optimizer_copy_transformed*: make sure the transformations and constraints etc are handled

parameter_names(*add_self=False*, *adjust_for_printing=False*, *recursive=True*, *intermediate=False*)
Get the names of all parameters of this model or parameter. It starts from the parameterized object you are calling this method on.

Note: This does not unravel multidimensional parameters, use parameter_names_flat to unravel parameters!

Parameters

- **add_self** (*bool*) – whether to add the own name in front of names
- **adjust_for_printing** (*bool*) – whether to call *adjust_name_for_printing* on names
- **recursive** (*bool*) – whether to traverse through hierarchy and append leaf node names
- **intermediate** (*bool*) – whether to add intermediate names, that is parameterized objects

parameter_names_flat(*include_fixed=False*)
Return the flattened parameter names for all subsequent parameters of this parameter. We do not include the name for self here!

If you want the names for fixed parameters as well in this list, set include_fixed to True.

if not hasattr(obj, 'cache'): obj.cache = FunctionCacher()

Parameters **include_fixed** (*bool*) – whether to include fixed names here.

randomize(*rand_gen=None*, **args*, ***kwargs*)
Randomize the model. Make this draw from the rand_gen if one exists, else draw random normal(0,1)

Parameters

- **rand_gen** – np random number generator which takes args and kwargs
- **loc** (*flaot*) – loc parameter for random number generator
- **scale** (*float*) – scale parameter for random number generator
- **kwargs** (*args,*) – will be passed through to random number generator

gradient_full
Note to users: This does not return the gradient in the right shape! Use self.gradient for the right gradient array.

To work on the gradient array, use this as the gradient handle. This method exists for in memory use of parameters. When trying to access the true gradient array, use this.

num_params
Return the number of parameters of this parameter_handle. Param objects will always return 0.

optimizer_array
Array for the optimizer to work on. This array always lives in the space for the optimizer. Thus, it is untransformed, going from Transformations.

Setting this array, will make sure the transformed parameters for this model will be set accordingly. It has to be set with an array, retrieved from this method, as e.g. fixing will resize the array.

The optimizer should only interfere with this array, such that transformations are secured.

class Parameterizable(**args*, ***kwargs*)
Bases: *paramz.core.parameter_core.OptimizationHandlable*

A parameterisable class.

This class provides the parameters list (ArrayList) and standard parameter handling, such as {link|unlink}_parameter(), traverse hierarchy and param_array, gradient_array and the empty parameters_changed().

This class is abstract and should not be instantiated. Use paramz.Parameterized() as node (or leaf) in the parameterized hierarchy. Use paramz.Param() for a leaf in the parameterized hierarchy.

disable_caching()

enable_caching()

initialize_parameter()
Call this function to initialize the model, if you built it without initialization.

This HAS to be called manually before optmizing or it will be causing unexpected behaviour, if not errors!

parameters_changed()
This method gets called when parameters have changed. Another way of listening to param changes is to add self as a listener to the param, such that updates get passed through. See :py:function:`paramz.param.Observable.add_observer`

save(*filename*, *ftype='HDF5'*)
Save all the model parameters into a file (HDF5 by default).

This is not supported yet. We are working on having a consistent, human readable way of saving and loading GPy models. This only saves the parameter array to a hdf5 file. In order to load the model again, use the same script for building the model you used to build this model. Then load the param array from this hdf5 file and set the parameters of the created model:

```
>>> m[:] = h5_file['param_array']
```

This is less then optimal, we are working on a better solution to that.

traverse(*visit*, **args*, ***kwargs*)
Traverse the hierarchy performing *visit(self, *args, **kwargs)* at every node passed by downwards. This function includes self!

See *visitor pattern* in literature. This is implemented in pre-order fashion.

Example:

```
#Collect all children:

children = []
self.traverse(children.append)
print children
```

traverse_parents(*visit*, **args*, ***kwargs*)
Traverse the hierarchy upwards, visiting all parents and their children except self. See "visitor pattern" in literature. This is implemented in pre-order fashion.

Example:

parents = [] self.traverse_parents(parents.append) print parents

gradient

num_params

param_array
Array representing the parameters of this class. There is only one copy of all parameters in memory, two during optimization.

!WARNING!: setting the parameter array MUST always be done in memory: m.param_array[:] = m_copy.param_array

unfixed_param_array
Array representing the parameters of this class. There is only one copy of all parameters in memory, two during optimization.

!WARNING!: setting the parameter array MUST always be done in memory: m.param_array[:] = m_copy.param_array

paramz.core.parentable module

class Parentable(**args*, ***kwargs*)
Bases: object

Enable an Object to have a parent.

Additionally this adds the parent_index, which is the index for the parent to look for in its parameter list.

has_parent()
Return whether this parentable object currently has a parent.

paramz.core.pickleable module

class Pickleable(**a*, ***kw*)
Bases: object

Make an object pickleable (See python doc 'pickling').

This class allows for pickling support by Memento pattern. _getstate returns a memento of the class, which gets pickled. _setstate(<memento>) (re-)sets the state of the class to the memento

copy(*memo=None*, *which=None*)
Returns a (deep) copy of the current parameter handle.

All connections to parents of the copy will be cut.

Parameters

- **memo** (*dict*) – memo for deepcopy
- **which** (*Parameterized*) – parameterized object which started the copy process [default: self]

pickle(*f*, *protocol=-1*)

Parameters

- **f** – either filename or open file object to write to. if it is an open buffer, you have to make sure to close it properly.
- **protocol** – pickling protocol to use, python-pickle for details.

paramz.core.updateable module

class Updateable(**args*, ***kwargs*)
Bases: *paramz.core.observable.Observable*

A model can be updated or not. Make sure updates can be switched on and off.

toggle_update()

trigger_update(*trigger_parent=True*)
Update the model from the current state. Make sure that updates are on, otherwise this method will do nothing

Parameters **trigger_parent** (*bool*) – Whether to trigger the parent, after self has updated

update_model(*updates=None*)
Get or set, whether automatic updates are performed. When updates are off, the model might be in a non-working state. To make the model work turn updates on again.

Parameters **updates** (*bool|None*) – bool: whether to do updates None: get the current update state

update_toggle()

Module contents

Core

This package holds the core modules for the parameterization

HierarchyError: raised when an error with the hierarchy occurs (circles etc.)

exception HierarchyError
Bases: exceptions.Exception

Gets thrown when something is wrong with the parameter hierarchy.

1.1.2 paramz.examples package

Submodules

paramz.examples.ridge_regression module

Created on 16 Oct 2015

@author: Max Zwiessele

class Basis(*degree*, *name='basis'*)
Bases: *paramz.parameterized.Parameterized*

Basis class for computing the design matrix phi(X). The weights are held in the regularizer, so that this only represents the design matrix.

basis(*X*, *i*)
Return the ith basis dimension. In the polynomial case, this is X**index. You can write your own basis function here, inheriting from this class and the gradients will still check.

Note: i will be zero for the first degree. This means we have also a bias in the model, which makes the problem of having an explicit bias obsolete.

class Lasso(*lambda_*, *name='Lasso'*)
Bases: *paramz.examples.ridge_regression.Regularizer*

update_error()

class Polynomial(*degree*, *name='polynomial'*)
Bases: *paramz.examples.ridge_regression.Basis*

basis(*X*, *i*)
Return the ith basis dimension. In the polynomial case, this is X**index. You can write your own basis function here, inheriting from this class and the gradients will still check.

Note: i will be zero for the first degree. This means we have also a bias in the model, which makes the problem of having an explicit bias obsolete.

class Regularizer(*lambda_*, *name='regularizer'*)
Bases: *paramz.parameterized.Parameterized*

init(*basis*, *input_dim*)

parameters_changed()
This method gets called when parameters have changed. Another way of listening to param changes is to add self as a listener to the param, such that updates get passed through. See :py:function:`paramz.param.Observable.add_observer`

update_error()

class Ridge(*lambda_*, *name='Ridge'*)
Bases: *paramz.examples.ridge_regression.Regularizer*

update_error()

class RidgeRegression(*X*, *Y*, *regularizer=None*, *basis=None*, *name='ridge_regression'*)
Bases: *paramz.model.Model*

Ridge regression with regularization.

For any regularization to work we to gradient based optimization.

Parameters

- **X** (*array-like*) – the inputs X of the regression problem
- **Y** (*array-like*) – the outputs Y

:param *paramz.examples.ridge_regression.Regularizer* regularizer: the regularizer to use
:param str name: the name of this regression object

objective_function()
The objective function for the given algorithm.

This function is the true objective, which wants to be minimized. Note that all parameters are already set and in place, so you just need to return the objective function here.

For probabilistic models this is the negative log_likelihood (including the MAP prior), so we return it here. If your model is not probabilistic, just return your objective to minimize here!

parameters_changed()
This method gets called when parameters have changed. Another way of listening to param changes is to add self as a listener to the param, such that updates get passed through. See :py:function:`paramz.param.Observable.add_observer`

phi(*Xpred*, *degrees=None*)
Compute the design matrix for this model using the degrees given by the index array in degrees

Parameters

- **Xpred** (*array-like*) – inputs to compute the design matrix for
- **degrees** (*array-like*) – array of degrees to use [default=range(self.degree+1)]

Returns array-like phi The design matrix [degree x #samples x #dimensions]

predict(*Xnew*)

degree

weights

Module contents

1.1.3 paramz.optimization package

Submodules

paramz.optimization.optimization module

class Adam(*step_rate=0.0002*, *decay=0*, *decay_mom1=0.1*, *decay_mom2=0.001*, *momentum=0*, *offset=1e-08*, **args*, ***kwargs*)
Bases: *paramz.optimization.optimization.Optimizer*

opt(*x_init*, *f_fp=None*, *f=None*, *fp=None*)

class Opt_Adadelta(*step_rate=0.1*, *decay=0.9*, *momentum=0*, **args*, ***kwargs*)
Bases: *paramz.optimization.optimization.Optimizer*

opt(*x_init*, *f_fp=None*, *f=None*, *fp=None*)

class Optimizer(*messages=False*, *max_f_eval=10000.0*, *max_iters=1000.0*, *ftol=None*, *gtol=None*, *xtol=None*, *bfgs_factor=None*)
Bases: object

Superclass for all the optimizers.

Parameters

- **x_init** – initial set of parameters
- **f_fp** – function that returns the function AND the gradients at the same time
- **f** – function to optimize
- **fp** – gradients
- **messages** (*(True | False)*) – print messages from the optimizer?
- **max_f_eval** – maximum number of function evaluations

Return type optimizer object.

opt(*x_init*, *f_fp=None*, *f=None*, *fp=None*)

run(*x_init*, ***kwargs*)

class RProp(*step_shrink=0.5*, *step_grow=1.2*, *min_step=1e-06*, *max_step=1*, *changes_max=0.1*, **args*, ***kwargs*)

Bases: *paramz.optimization.optimization.Optimizer*

opt(*x_init*, *f_fp=None*, *f=None*, *fp=None*)

class opt_SCG(**args*, ***kwargs*)

Bases: *paramz.optimization.optimization.Optimizer*

opt(*x_init*, *f_fp=None*, *f=None*, *fp=None*)

class opt_bfgs(**args*, ***kwargs*)

Bases: *paramz.optimization.optimization.Optimizer*

opt(*x_init*, *f_fp=None*, *f=None*, *fp=None*)

Run the optimizer

class opt_lbfgsb(**args*, ***kwargs*)

Bases: *paramz.optimization.optimization.Optimizer*

opt(*x_init*, *f_fp=None*, *f=None*, *fp=None*)

Run the optimizer

class opt_simplex(**args*, ***kwargs*)

Bases: *paramz.optimization.optimization.Optimizer*

opt(*x_init*, *f_fp=None*, *f=None*, *fp=None*)

The simplex optimizer does not require gradients.

class opt_tnc(**args*, ***kwargs*)

Bases: *paramz.optimization.optimization.Optimizer*

opt(*x_init*, *f_fp=None*, *f=None*, *fp=None*)

Run the TNC optimizer

get_optimizer(*f_min*)

paramz.optimization.scg module

Scaled Conjuagte Gradients, originally in Matlab as part of the Netlab toolbox by I. Nabney, converted to python N. Lawrence and given a pythonic interface by James Hensman.

Edited by Max Zwiessele for efficiency and verbose optimization.

SCG(*f*, *gradf*, *x*, *optargs=()*, *maxiters=500*, *max_f_eval=inf*, *xtol=None*, *ftol=None*, *gtol=None*)
Optimisation through Scaled Conjugate Gradients (SCG)

f: the objective function gradf : the gradient function (should return a 1D np.ndarray) x : the initial condition

Returns x the optimal value for x flog : a list of all the objective values function_eval number of fn evaluations status: string describing convergence status

paramz.optimization.verbose_optimization module

class VerboseOptimization(*model*, *opt*, *maxiters*, *verbose=False*, *current_iteration=0*, *ipython_notebook=True*, *clear_after_finish=False*)
Bases: `object`

finish(*opt*)

print_out(*seconds*)

print_status(*me*, *which=None*)

update()

exponents(*fnow*, *current_grad*)

Module contents

1.1.4 paramz.tests package

Submodules

paramz.tests.array_core_tests module

class ArrayCoreTest(*methodName='runTest'*)
Bases: `unittest.case.TestCase`

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

setUp()
Hook method for setting up the test fixture before exercising it.

test_init()

test_slice()

paramz.tests.cacher_tests module

Created on 4 Sep 2015

@author: maxz

class Test(*methodName='runTest'*)
Bases: `unittest.case.TestCase`

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

setUp()
Hook method for setting up the test fixture before exercising it.

test_cached_ObsAr()

test_cached_atomic_int()

test_cached_atomic_str()

test_caching_non_cachables()

test_chached_ObsAr_atomic()

test_copy()

test_force_kwargs()

test_name()

test_pickling()

test_reset()

test_reset_on_operation_error()

test_sum_ObsAr()

class TestDecorator(*methodName='runTest'*)
Bases: unittest.case.TestCase

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

setUp()
Hook method for setting up the test fixture before exercising it.

test_cacher_cache()

test_offswitch()

test_opcalls()

test_reset()

test_signature()

paramz.tests.examples_tests module

class Test2D(*methodName='runTest'*)
Bases: unittest.case.TestCase

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

testLassoRegression()

testRidgeRegression()

paramz.tests.index_operations_tests module

class Test(*methodName='runTest'*)
Bases: unittest.case.TestCase

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

setUp()
Hook method for setting up the test fixture before exercising it.

test_clear()

test_index_conversions()

test_index_view()

test_indexview_remove()

test_misc()

test_print()

test_remove()

test_shift_left()

test_shift_right()

test_view_of_view()

paramz.tests.init_tests module

class InitTests(*methodName='runTest'*)

Bases: unittest.case.TestCase

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

setUp()

Hook method for setting up the test fixture before exercising it.

test_initialize()

test_load_initialized()

test_constraints_in_init()

test_parameter_modify_in_init()

paramz.tests.lists_and_dicts_tests module

WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

class Test(*methodName='runTest'*)
Bases: unittest.case.TestCase

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

setUp()
Hook method for setting up the test fixture before exercising it.

tearDown()
Hook method for deconstructing the test fixture after testing it.

testArrayList()

testPrintObserverListObj()

testPrintObserverListObsAr()

testPrintObserverListParameterized()

testPrintPriority()

paramz.tests.model_tests module

class ModelTest(*methodName='runTest'*)
Bases: unittest.case.TestCase

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

setUp()
Hook method for setting up the test fixture before exercising it.

test_caching_offswitch()

test_checkgrad()

test_constraints_set_direct()

test_constraints_testmodel()

test_empty_parameterized()

test_fix_constrain()

test_fix_unfix()

test_fix_unfix_constraints()

test_fixing_optimize()

test_get_by_name()

test_hierarchy_error()

test_likelihood_replicate()

test_likelihood_set()

test_optimize_ada()

test_optimize_adam()

test_optimize_cgd()

test_optimize_error()

test_optimize_fix()

test_optimize_org_bfgs()

test_optimize_preferred()

test_optimize_restarts()

test_optimize_restarts_parallel()

test_optimize_rprop()

test_optimize_scg()

test_optimize_simplex()

test_optimize_tnc()

test_printing()

test_pydot()

test_raveled_index()

test_regular_expression_misc()

test_set_empty()

test_set_error()

test_set_get()

test_set_gradients()

test_updates()

paramz.tests.observable_tests module

class ParamTestParent(*name=None*, *parameters=[]*)
Bases: *paramz.parameterized.Parameterized*

parameters_changed()
This method gets called when parameters have changed. Another way of listening to param changes is to add self as a listener to the param, such that updates get passed through. See :py:function:paramz.param.Observable.add_observer

parent_changed_count = -1

class ParameterizedTest(*name=None*, *parameters=[]*)
Bases: *paramz.parameterized.Parameterized*

parameters_changed()
This method gets called when parameters have changed. Another way of listening to param changes is to add self as a listener to the param, such that updates get passed through. See :py:function:paramz.param.Observable.add_observer

params_changed_count = -1

class Test(*methodName='runTest'*)
Bases: unittest.case.TestCase

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

setUp()
Hook method for setting up the test fixture before exercising it.

testObsAr()

test_observable()

test_priority()

test_priority_notify()

test_set_params()

class TestMisc(*methodName='runTest'*)
Bases: unittest.case.TestCase

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

test_casting()

paramz.tests.parameterized_tests module

Created on Feb 13, 2014

@author: maxzwiessele

class M(*name*, ***kwargs*)
Bases: *paramz.model.Model*

log_likelihood()

objective_function()
The objective function for the given algorithm.

This function is the true objective, which wants to be minimized. Note that all parameters are already set and in place, so you just need to return the objective function here.

For probabilistic models this is the negative log_likelihood (including the MAP prior), so we return it here. If your model is not probabilistic, just return your objective to minimize here!

parameters_changed()
This method gets called when parameters have changed. Another way of listening to param changes is to add self as a listener to the param, such that updates get passed through. See :py:function:paramz.param.Observable.add_observer

class P(*name*, ***kwargs*)
Bases: *paramz.parameterized.Parameterized*

heres_johnny(*ignore=1*)

class ParameterizedTest(*methodName='runTest'*)
Bases: unittest.case.TestCase

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

setUp()
Hook method for setting up the test fixture before exercising it.

test_add_parameter()

test_add_parameter_already_in_hirarchy()

test_add_parameter_in_hierarchy()

test_checkgrad_hierarchy_error()

test_constraints()

test_constraints_link_unlink()

test_constraints_views()

test_default_constraints()

test_fixed_optimizer_copy()

test_fixes()

test_fixing_randomize()

test_fixing_randomize_parameter_handling()

test_index_operations()

test_names()

test_names_already_exist()

test_num_params()

test_original()

test_param_names()

test_printing()

test_randomize()

test_recursion_limit()

test_remove_parameter()

test_remove_parameter_param_array_grad_array()

test_set_param_array()

test_traverse_parents()

test_unfixed_param_array()

paramz.tests.pickle_tests module

Created on 13 Mar 2014

@author: maxz

class ListDictTestCase(*methodName='runTest'*)
Bases: unittest.case.TestCase

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

assertArrayListEquals(*l1*, *l2*)

assertListDictEquals(*d1*, *d2*, *msg=None*)

class Test(*methodName='runTest'*)
Bases: *paramz.tests.pickle_tests.ListDictTestCase*

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

test_observable_array()

test_param()

test_parameter_index_operations()

test_parameterized()

paramz.tests.verbose_optimize_tests module

class Test(*methodName='runTest'*)
Bases: unittest.case.TestCase

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

setUp()
Hook method for setting up the test fixture before exercising it.

test_finish()

test_timestrings()

Module contents

1.2 Submodules

1.3 paramz.__version__ module

1.4 paramz.caching module

class Cache_this(*limit=5*, *ignore_args=()*, *force_kwargs=()*)
Bases: object

A decorator which can be applied to bound methods in order to cache them

class Cacher(*operation*, *limit=3*, *ignore_args=()*, *force_kwargs=()*, *cacher_enabled=True*)
Bases: object

Cache an *operation*. If the operation is a bound method we will create a cache (FunctionCache) on that object in order to keep track of the caches on instances.

Warning: If the instance already had a Cacher for the operation that Cacher will be overwritten by this Cacher!

Parameters

- **operation** (*callable*) – function to cache
- **limit** (*int*) – depth of cacher
- **ignore_args** (*[int]*) – list of indices, pointing at arguments to ignore in **args* of *operation(*args)*. This includes self, so make sure to ignore self, if it is not cachable and you do not want this to prevent caching!
- **force_kwargs** (*[str]*) – list of kwarg names (strings). If a kwarg with that name is given, the cacher will force recompute and wont cache anything.

- **verbose** (*int*) – verbosity level. 0: no print outs, 1: casual print outs, 2: debug level print outs

add_to_cache(*cache_id*, *inputs*, *output*)
This adds cache_id to the cache, with inputs and output

combine_inputs(*args*, *kw*, *ignore_args*)
Combines the args and kw in a unique way, such that ordering of kwargs does not lead to recompute

disable_cacher()
Disable the caching of this cacher. This also removes previously cached results

enable_cacher()
Enable the caching of this cacher.

ensure_cache_length()
Ensures the cache is within its limits and has one place free

id(*obj*)
returns the self.id of an object, to be used in caching individual self.ids

on_cache_changed(*direct*, *which=None*)
A callback funtion, which sets local flags when the elements of some cached inputs change

this function gets 'hooked up' to the inputs when we cache them, and upon their elements being changed we update here.

prepare_cache_id(*combined_args_kw*)
get the cacheid (conc. string of argument self.ids in order)

reset()
Totally reset the cache

class FunctionCache(**args*, ***kwargs*)
Bases: dict

disable_caching()
Disable the cache of this object. This also removes previously cached results

enable_caching()
Enable the cache of this object.

reset()
Reset (delete) the cache of this object

1.5 paramz.domains module

(Hyper-)Parameter domains defined for *paramz.transformations*. These domains specify the legitimate realm of the parameters to live in.

_REAL : real domain, all values in the real numbers are allowed

_POSITIVE: positive domain, only positive real values are allowed

_NEGATIVE: same as _POSITIVE, but only negative values are allowed

_BOUNDED: only values within the bounded range are allowed, the bounds are specified withing the object with the bounded range

1.6 paramz.model module

class Model(*name*)

Bases: *paramz.parameterized.Parameterized*

objective_function()

The objective function for the given algorithm.

This function is the true objective, which wants to be minimized. Note that all parameters are already set and in place, so you just need to return the objective function here.

For probabilistic models this is the negative log_likelihood (including the MAP prior), so we return it here. If your model is not probabilistic, just return your objective to minimize here!

objective_function_gradients()

The gradients for the objective function for the given algorithm. The gradients are w.r.t. the *negative* objective function, as this framework works with *negative* log-likelihoods as a default.

You can find the gradient for the parameters in self.gradient at all times. This is the place, where gradients get stored for parameters.

This function is the true objective, which wants to be minimized. Note that all parameters are already set and in place, so you just need to return the gradient here.

For probabilistic models this is the gradient of the negative log_likelihood (including the MAP prior), so we return it here. If your model is not probabilistic, just return your *negative* gradient here!

optimize(*optimizer=None*, *start=None*, *messages=False*, *max_iters=1000*, *ipython_notebook=True*, *clear_after_finish=False*, ***kwargs*)

Optimize the model using self.log_likelihood and self.log_likelihood_gradient, as well as self.priors.

kwargs are passed to the optimizer. They can be:

Parameters

- **max_iters** (*int*) – maximum number of function evaluations
- **optimizer** (*string*) – which optimizer to use (defaults to self.preferred optimizer)

Messages True: Display messages during optimisation, "ipython_notebook":

Valid optimizers are:

- **'scg': scaled conjugate gradient method, recommended for stability.** See also GPy.inference.optimization.scg
- 'fmin_tnc': truncated Newton method (see scipy.optimize.fmin_tnc)
- 'simplex': the Nelder-Mead simplex method (see scipy.optimize.fmin),
- 'lbfgsb': the l-bfgs-b method (see scipy.optimize.fmin_l_bfgs_b),
- 'lbfgs': the bfgs method (see scipy.optimize.fmin_bfgs),
- 'sgd': stochastic gradient decsent (see scipy.optimize.sgd). For experts only!

optimize_restarts(*num_restarts=10*, *robust=False*, *verbose=True*, *parallel=False*, *num_processes=None*, ***kwargs*)

Perform random restarts of the model, and set the model to the best seen solution.

If the robust flag is set, exceptions raised during optimizations will be handled silently. If _all_ runs fail, the model is reset to the existing parameter values.

**kwargs are passed to the optimizer.

Parameters

- **num_restarts** (*int*) – number of restarts to use (default 10)
- **robust** (*bool*) – whether to handle exceptions silently or not (default False)
- **parallel** (*bool*) – whether to run each restart as a separate process. It relies on the multiprocessing module.
- **num_processes** – number of workers in the multiprocessing pool
- **max_f_eval** (*int*) – maximum number of function evaluations
- **max_iters** (*int*) – maximum number of iterations
- **messages** (*bool*) – whether to display during optimisation

---

Note: If num_processes is None, the number of workes in the multiprocessing pool is automatically set to the number of processors on the current machine.

---

opt_wrapper(*args*)

1.7 paramz.param module

class Param(*name*, *input_array*, *default_constraint=None*, **a*, ***kw*)

Bases: *paramz.core.parameter_core.Parameterizable*, *paramz.core.observable_array.ObsAr*

Parameter object for GPy models.

Parameters

- **name** (*str*) – name of the parameter to be printed
- **input_array** (*np.ndarray*) – array which this parameter handles
- **default_constraint** – The default constraint for this parameter

You can add/remove constraints by calling constrain on the parameter itself, e.g:

- self[:,1].constrain_positive()
- self[0].tie_to(other)
- self.untie()
- self[:3,:].unconstrain()
- self[1].fix()

Fixing parameters will fix them to the value they are right now. If you change the fixed value, it will be fixed to the new value!

Important Notes:

The array given into this, will be used as the Param object. That is, the memory of the numpy array given will be the memory of this object. If you want to make a new Param object you need to copy the input array!

Multilevel indexing (e.g. self[:2][1:]) is not supported and might lead to unexpected behaviour. Try to index in one go, using boolean indexing or the numpy builtin np.index function.

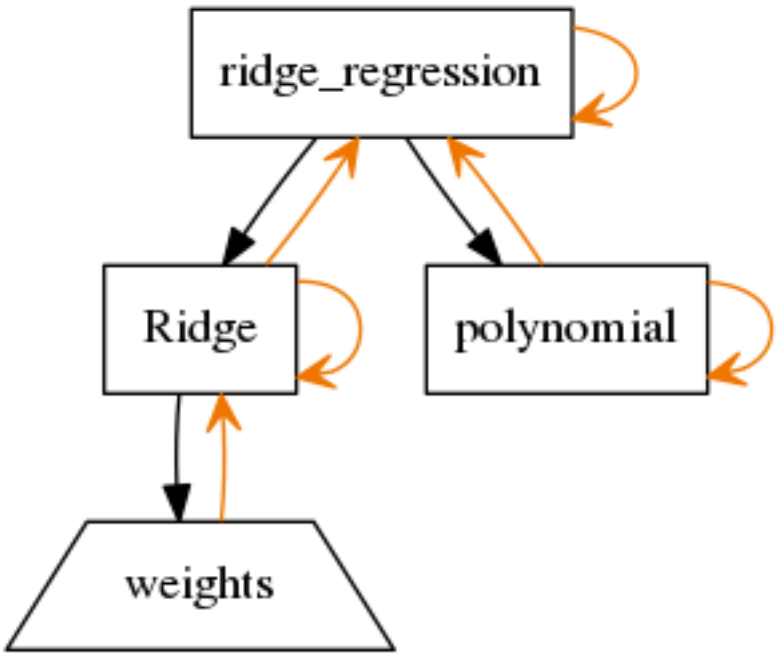
See `GPy.core.parameterized.Parameterized` for more details on constraining etc.

build_pydot(*G*)
Build a pydot representation of this model. This needs pydot installed.

Example Usage:

np.random.seed(1000) X = np.random.normal(0,1,(20,2)) beta = np.random.uniform(0,1,(2,1)) Y = X.dot(beta) m = RidgeRegression(X, Y) G = m.build_pydot() G.write_png('example_hierarchy_layout.png')

The output looks like:



Rectangles are parameterized objects (nodes or leafs of hierarchy).

Trapezoids are param objects, which represent the arrays for parameters.

Black arrows show parameter hierarchical dependence. The arrow points from parents towards children.

Orange arrows show the observer pattern. Self references (here) are the references to the call to parameters changed and references upwards are the references to tell the parents they need to update.

copy()
Returns a (deep) copy of the current parameter handle.

All connections to parents of the copy will be cut.

Parameters

- **memo** (*dict*) – memo for deepcopy
- **which** (Parameterized) – parameterized object which started the copy process [default: self]

get_property_string(*propname*)

parameter_names(*add_self=False*, *adjust_for_printing=False*, *recursive=True*, ***kw*)
Get the names of all parameters of this model or parameter. It starts from the parameterized object you are calling this method on.

Note: This does not unravel multidimensional parameters, use parameter_names_flat to unravel parameters!

Parameters

- **add_self** (*bool*) – whether to add the own name in front of names
- **adjust_for_printing** (*bool*) – whether to call *adjust_name_for_printing* on names
- **recursive** (*bool*) – whether to traverse through hierarchy and append leaf node names
- **intermediate** (*bool*) – whether to add intermediate names, that is parameterized objects

flattened_parameters

gradient

Return a view on the gradient, which is in the same shape as this parameter is. Note: this is not the real gradient array, it is just a view on it.

To work on the real gradient array use: self.full_gradient

is_fixed

num_params

param_array

As we are a leaf, this just returns self

parameters = []

values

Return self as numpy array view

class ParamConcatenation(*params*)

Bases: object

Parameter concatenation for convenience of printing regular expression matched arrays you can index this concatenation as if it was the flattened concatenation of all the parameters it contains, same for setting parameters (Broadcasting enabled).

See GPy.core.parameter.Param for more details on constraining.

checkgrad(*verbose=False*, *step=1e-06*, *tolerance=0.001*)

constrain(*constraint*, *warning=True*)

Parameters

- **transform** – the *paramz.transformations.Transformation* to constrain the this parameter to.
- **warning** – print a warning if re-constraining parameters.

Constrain the parameter to the given *paramz.transformations.Transformation*.

constrain_bounded(*lower*, *upper*, *warning=True*)

Parameters

- **upper** (*lower,*) – the limits to bound this parameter to
- **warning** – print a warning if re-constraining parameters.

Constrain this parameter to lie within the given range.

constrain_fixed(*value=None*, *warning=True*, *trigger_parent=True*)

Constrain this parameter to be fixed to the current value it carries.

This does not override the previous constraints, so unfixing will restore the constraint set before fixing.

Parameters **warning** – print a warning for overwriting constraints.

constrain_negative(*warning=True*)

Parameters **warning** – print a warning if re-constraining parameters.

Constrain this parameter to the default negative constraint.

constrain_positive(*warning=True*)

Parameters **warning** – print a warning if re-constraining parameters.

Constrain this parameter to the default positive constraint.

fix(*value=None*, *warning=True*, *trigger_parent=True*)
Constrain this parameter to be fixed to the current value it carries.

This does not override the previous constraints, so unfixing will restore the constraint set before fixing.

Parameters **warning** – print a warning for overwriting constraints.

unconstrain(**constraints*)

Parameters **transforms** – The transformations to unconstrain from.

remove all *paramz.transformations.Transformation* transformats of this parameter object.

unconstrain_bounded(*lower*, *upper*)

Parameters **upper** (*lower,*) – the limits to unbound this parameter from

Remove (lower, upper) bounded constrain from this parameter/

unconstrain_fixed()
This parameter will no longer be fixed.

If there was a constraint on this parameter when fixing it, it will be constraint with that previous constraint.

unconstrain_negative()
Remove negative constraint of this parameter.

unconstrain_positive()
Remove positive constraint of this parameter.

unfix()
This parameter will no longer be fixed.

If there was a constraint on this parameter when fixing it, it will be constraint with that previous constraint.

update_all_params()

values()

1.8 paramz.parameterized module

class Parameterized(*name=None*, *parameters=[]*)
Bases: *paramz.core.parameter_core.Parameterizable*

Say m is a handle to a parameterized class.

Printing parameters:

```
- print m:           prints a nice summary over all parameters
- print m.name:      prints details for param with name 'name'
- print m[regexp]: prints details for all the parameters
                     which match (!) regexp
- print m['']:       prints details for all parameters
```

Fields:

```
Name:       The name of the param, can be renamed!
Value:      Shape or value, if one-valued
Constrain:  constraint of the param, curly "{c}" brackets indicate
            some parameters are constrained by c. See detailed print
            to get exact constraints.
Tied_to:    which paramter it is tied to.
```

Getting and setting parameters:

```
- Set all values in param to one:      m.name.to.param = 1
- Set all values in parameterized:     m.name[:] = 1
- Set values to random values:         m[:] = np.random.norm(m.size)
```

Handling of constraining, fixing and tieing parameters:

```
- You can constrain parameters by calling the constrain on the param itself, e.g:

   - m.name[:,1].constrain_positive()
   - m.name[0].tie_to(m.name[1])

- Fixing parameters will fix them to the value they are right now. If you change
  the parameters value, the param will be fixed to the new value!

- If you want to operate on all parameters use m[''] to wildcard select all␣
↪paramters
  and concatenate them. Printing m[''] will result in printing of all parameters␣
↪in detail.
```

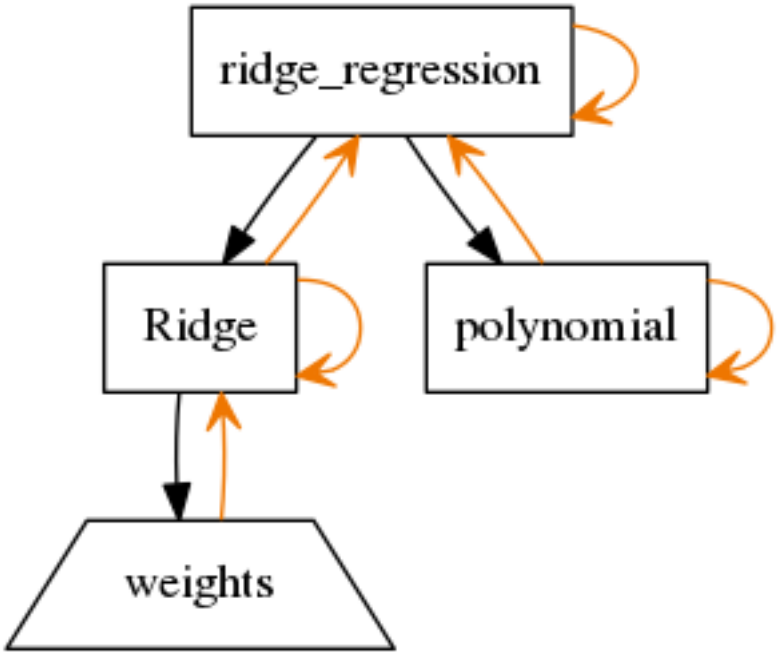
build_pydot(*G=None*)

Build a pydot representation of this model. This needs pydot installed.

Example Usage:

```
np.random.seed(1000)
X = np.random.normal(0,1,(20,2))
beta = np.random.uniform(0,1,(2,1))
Y = X.dot(beta)
m = RidgeRegression(X, Y)
G = m.build_pydot()
G.write_png('example_hierarchy_layout.png')
```

The output looks like:



Rectangles are parameterized objects (nodes or leafs of hierarchy).

Trapezoids are param objects, which represent the arrays for parameters.

Black arrows show parameter hierarchical dependence. The arrow points from parents towards children.

Orange arrows show the observer pattern. Self references (here) are the references to the call to parameters changed and references upwards are the references to tell the parents they need to update.

copy(*memo=None*)
Returns a (deep) copy of the current parameter handle.

All connections to parents of the copy will be cut.

Parameters

- **memo** (*dict*) – memo for deepcopy
- **which** (*Parameterized*) – parameterized object which started the copy process [default: self]

get_property_string(*propname*)

grep_param_names(*regexp*)
create a list of parameters, matching regular expression regexp

link_parameter(*param*, *index=None*)

Parameters

- **parameters** (list of or one *paramz.param.Param*) – the parameters to add
- **[index]** – index of where to put parameters

Add all parameters to this param class, you can insert parameters at any given index using the `list.insert` syntax

link_parameters(**parameters*)
convenience method for adding several parameters without gradient specification

unlink_parameter(*param*)

Parameters **param** – param object to remove from being a parameter of this parameterized object.

flattened_parameters

class ParametersChangedMeta
Bases: type

1.9 paramz.transformations module

class Exponent
Bases: *paramz.transformations.Transformation*

f(*x*)

finv(*x*)

gradfactor(*f*, *df*)
df(opt_param)_dopt_param evaluated at self.f(opt_param)=model_param, times the gradient dL_dmodel_param,

i.e.: define

rac{ rac{partial L}{partial f}left(left.partial f(x)}{partial x} ight|_{x=f^{-1}(f) ight)}

initialize(*f*)
produce a sensible initial value for f(x)

log_jacobian(*model_param*)
compute the log of the jacobian of f, evaluated at f(x)= model_param

log_jacobian_grad(*model_param*)
compute the drivative of the log of the jacobian of f, evaluated at f(x)= model_param

domain = 'positive'

class Logexp
Bases: *paramz.transformations.Transformation*

f(*x*)

finv(*f*)

gradfactor(*f*, *df*)
df(opt_param)_dopt_param evaluated at self.f(opt_param)=model_param, times the gradient dL_dmodel_param,

i.e.: define

rac{ rac{partial L}{partial f}left(left.partial f(x)}{partial x} ight|_{x=f^{-1}(f) ight)}

initialize(*f*)
produce a sensible initial value for f(x)

log_jacobian(*model_param*)
compute the log of the jacobian of f, evaluated at f(x)= model_param

log_jacobian_grad(*model_param*)
compute the drivative of the log of the jacobian of f, evaluated at f(x)= model_param

domain = 'positive'

class Logistic(*lower*, *upper*)
Bases: *paramz.transformations.Transformation*

f(*x*)

finv(*f*)

gradfactor(*f*, *df*)
df(opt_param)_dopt_param evaluated at self.f(opt_param)=model_param, times the gradient dL_dmodel_param,

i.e.: define

rac{ rac{partial L}{partial f}left(left.partial f(x)}{partial x} ight|_{x=f^{-1}(f) ight)}

initialize(*f*)
produce a sensible initial value for f(x)

domain = 'bounded'

class NegativeExponent
Bases: *paramz.transformations.Exponent*

f(*x*)

finv(*f*)

gradfactor(*f*, *df*)
df(opt_param)_dopt_param evaluated at self.f(opt_param)=model_param, times the gradient dL_dmodel_param,

i.e.: define

rac{ rac{partial L}{partial f}left(left.partial f(x)}{partial x} ight|_{x=f^{-1}(f) ight)}

initialize(*f*)
produce a sensible initial value for f(x)

domain = 'negative'

class NegativeLogexp
Bases: *paramz.transformations.Transformation*

f(*x*)

finv(*f*)

gradfactor(*f*, *df*)
df(opt_param)_dopt_param evaluated at self.f(opt_param)=model_param, times the gradient dL_dmodel_param,

i.e.: define

rac{ rac{partial L}{partial f}left(left.partial f(x)}{partial x} ight|_{x=f^{-1}(f) ight)}

initialize(*f*)
produce a sensible initial value for f(x)

domain = 'negative'

logexp = Logexp

class Square
Bases: *paramz.transformations.Transformation*

f(*x*)

finv(*x*)

gradfactor(*f*, *df*)
df(opt_param)_dopt_param evaluated at self.f(opt_param)=model_param, times the gradient dL_dmodel_param,

i.e.: define

rac{ rac{partial L}{partial f}left(left.partial f(x)}{partial x} ight|_{x=f^{-1}(f) ight)}

initialize(*f*)
produce a sensible initial value for f(x)

domain = 'positive'

class Transformation
Bases: object

f(*opt_param*)

finv(*model_param*)

gradfactor(*model_param*, *dL_dmodel_param*)
df(opt_param)_dopt_param evaluated at self.f(opt_param)=model_param, times the gradient dL_dmodel_param,

i.e.: define

rac{ rac{partial L}{partial f}left(left.partial f(x)}{partial x} ight|_{x=f^{-1}(f) ight)}

gradfactor_non_natural(*model_param*, *dL_dmodel_param*)

initialize(*f*)
produce a sensible initial value for f(x)

log_jacobian(*model_param*)
compute the log of the jacobian of f, evaluated at f(x)= model_param

log_jacobian_grad(*model_param*)
compute the drivative of the log of the jacobian of f, evaluated at f(x)= model_param

plot(*xlabel='transformed $\\theta$'*, *ylabel='$\\theta$'*, *axes=None*, **args*, ***kw*)

domain = None

1.10 paramz.util module

1.11 Module contents

load(*file_or_path*)
Load a previously pickled model, using *m.pickle('path/to/file.pickle)'*

Parameters **file_name** – path/to/file.pickle

Python Module Index

Index

G

H

I

L

M

N

O

P

R

S

T

U

V

W