GPyOpt Documentation

GPyOpt Authors

Mar 27, 2018

Contents:

CHAPTER 1

GPyOpt.acquisitions package

1.1 Submodules

1.2 GPyOpt.acquisitions.EI module

class GPyOpt.acquisitions.EI.**AcquisitionEI**(*model*, *space*, *optimizer=None*, *cost_withGradients=None*, *jitter=0.01*)

Bases: *GPyOpt.acquisitions.base.AcquisitionBase*

Expected improvement acquisition function

Parameters

- **model** – GPyOpt class of model
- **space** – GPyOpt class of domain
- **optimizer** – optimizer of the acquisition. Should be a GPyOpt optimizer
- **cost_withGradients** – function
- **jitter** – positive value to make the acquisition more explorative.

Note: allows to compute the Improvement per unit of cost

---

analytical_gradient_prediction = True

static fromConfig(*space*, *optimizer*, *cost_withGradients*, *config*)

1.3 GPyOpt.acquisitions.EI_mcmc module

class GPyOpt.acquisitions.EI_mcmc.**AcquisitionEI_MCMC**(*model*, *space*, *optimizer=None*, *cost_withGradients=None*, *jitter=0.01*)

Bases: *GPyOpt.acquisitions.EI.AcquisitionEI*

Integrated Expected improvement acquisition function

Parameters

- **model** – GPyOpt class of model
- **space** – GPyOpt class of domain
- **optimizer** – optimizer of the acquisition. Should be a GPyOpt optimizer
- **cost_withGradients** – function
- **jitter** – positive value to make the acquisition more explorative

---

Note: allows to compute the Improvement per unit of cost

---

analytical_gradient_prediction = True

1.4 GPyOpt.acquisitions.LCB module

class GPyOpt.acquisitions.LCB.**AcquisitionLCB**(*model*, *space*, *optimizer=None*, *cost_withGradients=None*, *exploration_weight=2*)

Bases: *GPyOpt.acquisitions.base.AcquisitionBase*

GP-Lower Confidence Bound acquisition function

Parameters

- **model** – GPyOpt class of model
- **space** – GPyOpt class of domain
- **optimizer** – optimizer of the acquisition. Should be a GPyOpt optimizer
- **cost_withGradients** – function
- **jitter** – positive value to make the acquisition more explorative

---

Note: does not allow to be used with cost

---

analytical_gradient_prediction = True

1.5 GPyOpt.acquisitions.LCB_mcmc module

class GPyOpt.acquisitions.LCB_mcmc.**AcquisitionLCB_MCMC**(*model*, *space*, *optimizer=None*, *cost_withGradients=None*, *exploration_weight=2*)

Bases: *GPyOpt.acquisitions.LCB.AcquisitionLCB*

Integrated GP-Lower Confidence Bound acquisition function

Parameters

- **model** – GPyOpt class of model
- **space** – GPyOpt class of domain
- **optimizer** – optimizer of the acquisition. Should be a GPyOpt optimizer
- **cost_withGradients** – function
- **exploration_weight** – positive parameter to control exploration / exploitation

Note: allows to compute the Improvement per unit of cost

analytical_gradient_prediction = True

1.6 GPyOpt.acquisitions.LP module

class GPyOpt.acquisitions.LP.**AcquisitionLP**(*model*, *space*, *optimizer*, *acquisition*, *transform='none'*)

Bases: *GPyOpt.acquisitions.base.AcquisitionBase*

Class for Local Penalization acquisition. Used for batch design. :param model: model of the class GPyOpt :param space: design space of the class GPyOpt. :param optimizer: optimizer of the class GPyOpt. :param acquisition: acquisition function of the class GPyOpt :param transform: transformation applied to the acquisition (default, none).

Note: irrespective of the transformation applied the penalized acquisition is always mapped again to the log space.

This way gradients can be computed additively and are more stable.

acquisition_function(*x*)
Returns the value of the acquisition function at x.

acquisition_function_withGradients(*x*)
Returns the acquisition function and its its gradient at x.

analytical_gradient_prediction = True

d_acquisition_function(*x*)
Returns the gradient of the acquisition function at x.

update_batches(*X_batch*, *L*, *Min*)
Updates the batches internally and pre-computes the

1.7 GPyOpt.acquisitions.MPI module

class GPyOpt.acquisitions.MPI.**AcquisitionMPI**(*model*, *space*, *optimizer=None*, *cost_withGradients=None*, *jitter=0.01*)

Bases: *GPyOpt.acquisitions.base.AcquisitionBase*

Maximum probability of improvement acquisition function

Parameters

- **model** – GPyOpt class of model
- **space** – GPyOpt class of domain
- **optimizer** – optimizer of the acquisition. Should be a GPyOpt optimizer
- **cost_withGradients** – function
- **jitter** – positive value to make the acquisition more explorative

Note: allows to compute the Improvement per unit of cost

analytical_gradient_prediction = True

static fromConfig(*space*, *optimizer*, *cost_withGradients*, *config*)

1.8 GPyOpt.acquisitions.MPI_mcmc module

class GPyOpt.acquisitions.MPI_mcmc.**AcquisitionMPI_MCMC**(*model*, *space*, *optimizer=None*, *cost_withGradients=None*, *jitter=0.01*)

Bases: *GPyOpt.acquisitions.MPI.AcquisitionMPI*

Integrated Maximum Probability of Improvement acquisition function

Parameters

- **model** – GPyOpt class of model
- **space** – GPyOpt class of domain
- **optimizer** – optimizer of the acquisition. Should be a GPyOpt optimizer
- **cost_withGradients** – function
- **jitter** – positive value to make the acquisition more explorative

Note: allows to compute the Improvement per unit of cost

analytical_gradient_prediction = True

1.9 GPyOpt.acquisitions.base module

class GPyOpt.acquisitions.base.**AcquisitionBase**(*model*, *space*, *optimizer*, *cost_withGradients=None*)

Bases: object

Base class for acquisition functions in Bayesian Optimization

Parameters

- **model** – GPyOpt class of model
- **space** – GPyOpt class of domain
- **optimizer** – optimizer of the acquisition. Should be a GPyOpt optimizer

acquisition_function(*x*)
Takes an acquisition and weights it so the domain and cost are taken into account.

acquisition_function_withGradients(*x*)
Takes an acquisition and it gradient and weights it so the domain and cost are taken into account.

analytical_gradient_prediction = False

static fromDict(*space*, *optimizer*, *cost_withGradients*, *config*)

optimize(*duplicate_manager=None*)
Optimizes the acquisition function (uses a flag from the model to use gradients or not).

1.10 Module contents

GPyOpt.acquisitions.**select_acquisition**(*name*)
Acquisition selector

CHAPTER 2

GPyOpt.core package

2.1 Subpackages

2.1.1 GPyOpt.core.evaluators package

Submodules

GPyOpt.core.evaluators.base module

class GPyOpt.core.evaluators.base.**EvaluatorBase**(*acquisition*, *batch_size*, ***kwargs*)
Bases: object

Base class for the evaluator of the function. This class handles both sequential and batch evaluators.

compute_batch(*duplicate_manager=None*, *context_manager=None*)

class GPyOpt.core.evaluators.base.**SamplingBasedBatchEvaluator**(*acquisition*, *batch_size*, ***kwargs*)
Bases: *GPyOpt.core.evaluators.base.EvaluatorBase*

This class handles specific types of batch evaluators, based on the sampling of anchor points (examples are random and Thompson sampling).

compute_batch(*duplicate_manager=None*, *context_manager=None*)

compute_batch_without_duplicate_logic(*context_manager=None*)

get_anchor_points(*duplicate_manager=None*, *context_manager=None*)

initialize_batch(*duplicate_manager=None*, *context_manager=None*)

optimize_anchor_point(*a*, *duplicate_manager=None*, *context_manager=None*)

zip_and_tuple(*x*)
convenient helper :param x: input configuration in the model space :return: zipped x as a tuple

GPyOpt.core.evaluators.batch_local_penalization module

class GPyOpt.core.evaluators.batch_local_penalization.**LocalPenalization**(*acquisition*, *batch_size*)

Bases: *GPyOpt.core.evaluators.base.EvaluatorBase*

Class for the batch method on 'Batch Bayesian optimization via local penalization' (Gonzalez et al., 2016).

Parameters

- **acquisition** – acquisition function to be used to compute the batch.
- **size** (*batch*) – the number of elements in the batch.

compute_batch(*duplicate_manager=None*, *context_manager=None*)

Computes the elements of the batch sequentially by penalizing the acquisition.

GPyOpt.core.evaluators.batch_local_penalization.**estimate_L**(*model*, *bounds*, *store-history=True*)

Estimate the Lipschitz constant of f by taking maximizing the norm of the expectation of the gradient of *f*.

GPyOpt.core.evaluators.batch_random module

class GPyOpt.core.evaluators.batch_random.**RandomBatch**(*acquisition*, *batch_size*)

Bases: *GPyOpt.core.evaluators.base.SamplingBasedBatchEvaluator*

Class for a random batch method. The first element of the batch is selected by optimizing the acquisition in a standard way. The remaining elements are selected uniformly random in the domain of the objective.

Parameters

- **acquisition** – acquisition function to be used to compute the batch.
- **size** (*batch*) – the number of elements in the batch.

compute_batch_without_duplicate_logic(*context_manager=None*)

get_anchor_points(*duplicate_manager=None*, *context_manager=None*)

initialize_batch(*duplicate_manager=None*, *context_manager=None*)

optimize_anchor_point(*a*, *duplicate_manager=None*, *context_manager=None*)

GPyOpt.core.evaluators.batch_thompson module

class GPyOpt.core.evaluators.batch_thompson.**ThompsonBatch**(*acquisition*, *batch_size*)

Bases: *GPyOpt.core.evaluators.base.SamplingBasedBatchEvaluator*

Class for a Thompson batch method. Elements are selected iteratively using the current acquistion function but exploring the models by using Thompson sampling

Parameters

- **acquisition** – acquisition function to be used to compute the batch.
- **size** (*batch*) – the number of elements in the batch.

compute_batch_without_duplicate_logic(*context_manager=None*)

get_anchor_points(*duplicate_manager=None*, *context_manager=None*)

initialize_batch(*duplicate_manager=None*, *context_manager=None*)

optimize_anchor_point(*a*, *duplicate_manager=None*, *context_manager=None*)

GPyOpt.core.evaluators.sequential module

class GPyOpt.core.evaluators.sequential.**Sequential**(*acquisition*, *batch_size=1*)
Bases: *GPyOpt.core.evaluators.base.EvaluatorBase*

Class for standard Sequential Bayesian optimization methods.

Parameters

- **acquisition** – acquisition function to be used to compute the batch.
- **size** (*batch*) – it is 1 by default since this class is only used for sequential methods.

compute_batch(*duplicate_manager=None*, *context_manager=None*)
Selects the new location to evaluate the objective.

Module contents

GPyOpt.core.evaluators.**select_evaluator**(*name*)

2.1.2 GPyOpt.core.task package

Submodules

GPyOpt.core.task.cost module

class GPyOpt.core.task.cost.**CostModel**(*cost_withGradients*)
Bases: object

Class to handle the cost of evaluating the function.

param cost_withGradients: function that returns the cost of evaluating the function and its gradient. By default no cost is used. Options are:

- cost_withGradients is some pre-defined cost function. Should return numpy array as outputs.
- cost_withGradients = 'evaluation_time'.

Note: if cost_withGradients = 'evaluation time' the evaluation time of the function is used to model a GP whose

---

mean is used as cost.

update_cost_model(*x*, *cost_x*)
Updates the GP used to handle the cost.

param x: input of the GP for the cost model. param x_cost: values of the time cost at the input locations.

GPyOpt.core.task.cost.**constant_cost_withGradients**(*x*)
Constant cost function used by default: cost=1, d_cost =0.

GPyOpt.core.task.objective module

class GPyOpt.core.task.objective.**Objective**
Bases: object

General class to handle the objective function internally.

evaluate(*x*)

class GPyOpt.core.task.objective.**SingleObjective**(*func*, *num_cores=1*, *objective_name='no_name'*, *batch_type='synchronous'*, *space=None*)
Bases: *GPyOpt.core.task.objective.Objective*

Class to handle problems with one single objective function.

param func: objective function. param batch_size: size of the batches (default, 1) param num_cores: number of cores to use in the process of evaluating the objective (default, 1). param objective_name: name of the objective function. param batch_type: Type of batch used. Only 'synchronous' evaluations are possible at the moment. param space: Not in use.

Note: the objective function should take 2-dimensional numpy arrays as input and outputs. Each row should

---

contain a location (in the case of the inputs) or a function evaluation (in the case of the outputs).

evaluate(*x*)
Performs the evaluation of the objective at x.

GPyOpt.core.task.space module

class GPyOpt.core.task.space.**Design_space**(*space*, *constraints=None*, *store_noncontinuous=False*)
Bases: object

Class to handle the input domain of the function. The format of a input domain, possibly with restrictions: The domain is defined as a list of dictionaries contains a list of attributes, e.g.:

- Arm bandit

space =[{'name': 'var_1', 'type': 'bandit', 'domain': [(-1,1),(1,0),(0,1)]}, {'name': 'var_2', 'type': 'bandit', 'domain': [(-1,4),(0,0),(1,2)]}]

- Continuous domain

space =[{'name': 'var_1', 'type': 'continuous', 'domain':(-1,1), 'dimensionality':1}, {'name': 'var_2', 'type': 'continuous', 'domain':(-3,1), 'dimensionality':2}, {'name': 'var_3', 'type': 'bandit', 'domain': [(-1,1),(1,0),(0,1)], 'dimensionality':2}, {'name': 'var_4', 'type': 'bandit', 'domain': [(-1,4),(0,0),(1,2)]}, {'name': 'var_5', 'type': 'discrete', 'domain': (0,1,2,3)}]

- Discrete domain

space =[{'name': 'var_3', 'type': 'discrete', 'domain': (0,1,2,3)}] {'name': 'var_3', 'type': 'discrete', 'domain': (-10,10)}]

- Mixed domain

space =[{'name': 'var_1', 'type': 'continuous', 'domain':(-1,1), 'dimensionality' :1}, {'name': 'var_4', 'type': 'continuous', 'domain':(-3,1), 'dimensionality' :2}, {'name': 'var_3', 'type': 'discrete', 'domain': (0,1,2,3)}]

Restrictions can be added to the problem. Each restriction is of the form c(x) <= 0 where c(x) is a function of the input variables previously defined in the space. Restrictions should be written as a list of dictionaries. For instance, this is an example of an space coupled with a constraint

space =[{'name': 'var_1', 'type': 'continuous', 'domain':(-1,1), 'dimensionality' :2}] constraints = [{'name': 'const_1', 'constraint': 'x[:,0]**2 + x[:,1]**2 - 1'}]

If no constraints are provided the hypercube determined by the bounds constraints are used.

Note about the internal representation of the vatiables: for variables in which the dimaensionality has been specified in the domain, a subindex is internally asigned. For instance if the variables is called 'var1' and has dimensionality 3, the first three positions in the internal representation of the domain will be occupied by variables 'var1_1', 'var1_2' and 'var1_3'. If no dimensionality is added, the internal naming remains the same. For instance, in the example above 'var3' should be fixed its original name.

param space: list of dictionaries as indicated above. param constraints: list of dictionaries as indicated above (default, none)

find_variable(*variable_name*)

static fromConfig(*constraints*)

get_bandit()
Extracts the arms of the bandit if any.

get_bounds()
Extracts the bounds of all the inputs of the domain of the *model*

get_continuous_bounds()
Extracts the bounds of the continuous variables.

get_continuous_dims()
Returns the dimension of the continuous components of the domain.

get_continuous_space()
Extracts the list of dictionaries with continuous components

get_discrete_dims()
Returns the dimension of the discrete components of the domain.

get_discrete_grid()
Computes a Numpy array with the grid of points that results after crossing the possible outputs of the discrete variables

get_discrete_space()
Extracts the list of dictionaries with continuous components

get_subspace(*dims*)
Extracts subspace from the reference of a list of variables in the inputs of the model.

has_constraints()
Checks if the problem has constraints. Note that the coordinates of the constraints are defined in terms of the model inputs and not in terms of the objective inputs. This means that if bandit or discre varaibles are in place, the restrictions should reflect this fact (TODO: implement the mapping of constraints defined on the objective to constraints defined on the model).

has_continuous()
Returns *true* if the space contains at least one continuous variable, and *false* otherwise

indicator_constraints(*x*)
Returns array of ones and zeros indicating if x is within the constraints

input_dim()
Extracts the input dimension of the domain.

model_to_objective(*x_model*)
This function serves as interface between model input vectors and objective input vectors

objective_to_model(*x_objective*)
This function serves as interface between objective input vectors and model input vectors

round_optimum(*x*)
Rounds some value x to a feasible value in the design space. x is expected to be a vector or an array with a single row

supported_types = ['continuous', 'discrete', 'bandit', 'categorical']

unzip_inputs(*X*)

zip_inputs(*X*)

GPyOpt.core.task.space.**bounds_to_space**(*bounds*)
Takes as input a list of tuples with bounds, and create a dictionary to be processed by the class Design_space. This function us used to keep the compatibility with previous versions of GPyOpt in which only bounded continuous optimization was possible (and the optimization domain passed as a list of tuples).

GPyOpt.core.task.variables module

class GPyOpt.core.task.variables.**BanditVariable**(*name*, *domain*, *dimensionality=None*)
Bases: *GPyOpt.core.task.variables.Variable*

expand()
Builds a list of single dimensional variables representing current variable.

Examples: For single dimensional variable, it is returned as is discrete of (0,2,4) -> discrete of (0,2,4) For multi dimensional variable, a list of variables is returned, each representing a single dimension continuous {0<=x<=1, 2<=y<=3} -> continuous {0<=x<=1}, continuous {2<=y<=3}

get_bounds()
Returns a list of tuples representing bounds of the variable

get_possible_values()
Returns a list of possible variable values

is_bandit()

model_to_objective(*x_model*)
Translates model input to objective input with respect to current variable

objective_to_model(*x_objective*)
Translates objective input to model input with respect to current variable

round(*value_array*)
Rounds a bandit variable by selecting the closest point in the domain Closest here is defined by euclidian distance Assumes an 1d array of the same length as the single variable value

class GPyOpt.core.task.variables.**CategoricalVariable**(*name*, *domain*, *dimensionality=1*)
Bases: *GPyOpt.core.task.variables.Variable*

expand()
Builds a list of single dimensional variables representing current variable.

Examples: For single dimensional variable, it is returned as is discrete of (0,2,4) -> discrete of (0,2,4) For multi dimensional variable, a list of variables is returned, each representing a single dimension continuous {0<=x<=1, 2<=y<=3} -> continuous {0<=x<=1}, continuous {2<=y<=3}

get_bounds()
Returns a list of tuples representing bounds of the variable

get_possible_values()
Returns a list of possible variable values

is_bandit()

model_to_objective(*x_model*, *index_in_model*)
Translates model input to objective input with respect to current variable

objective_to_model(*x_objective*)
Translates objective input to model input with respect to current variable

round(*value_array*)
Rounds a categorical variable by setting to one the max of the given vector and to zero the rest of the entries. Assumes an 1x[number of categories] array (due to one-hot encoding) as an input

class GPyOpt.core.task.variables.**ContinuousVariable**(*name*, *domain*, *dimensionality=1*)
Bases: *GPyOpt.core.task.variables.Variable*

get_bounds()
Returns a list of tuples representing bounds of the variable

get_possible_values()
Returns a list of possible variable values

is_bandit()

is_continuous()

round(*value_array*)
If value falls within bounds, just return it otherwise return min or max, whichever is closer to the value Assumes an 1d array with a single element as an input.

class GPyOpt.core.task.variables.**DiscreteVariable**(*name*, *domain*, *dimensionality=1*)
Bases: *GPyOpt.core.task.variables.Variable*

get_bounds()
Returns a list of tuples representing bounds of the variable

get_possible_values()
Returns a list of possible variable values

is_bandit()

round(*value_array*)
Rounds a discrete variable by selecting the closest point in the domain Assumes an 1d array with a single element as an input.

class GPyOpt.core.task.variables.**Variable**(*name*, *var_type*, *domain*, *dimensionality*)
Bases: object

expand()
Builds a list of single dimensional variables representing current variable.

Examples: For single dimensional variable, it is returned as is discrete of (0,2,4) -> discrete of (0,2,4) For multi dimensional variable, a list of variables is returned, each representing a single dimension continuous {0<=x<=1, 2<=y<=3} -> continuous {0<=x<=1}, continuous {2<=y<=3}

get_bounds()
Returns a list of tuples representing bounds of the variable

get_possible_values()
Returns a list of possible variable values

is_continuous()

model_to_objective(*x_model*, *index_in_model*)
Translates model input to objective input with respect to current variable

objective_to_model(*x_objective*)
Translates objective input to model input with respect to current variable

round(*value_array*)
Rounds the given value to the variable's domain. Value is assumed to be in a 1x[variable dimentionality] numpy array

set_index_in_model(*index*)
Allows to set the index of this variable in the model space

set_index_in_objective(*index*)
Allows to set the index of this variable in the objective space

GPyOpt.core.task.variables.**create_variable**(*descriptor*)
Creates a variable from a dictionary descriptor

Module contents

2.2 Submodules

2.3 GPyOpt.core.bo module

class GPyOpt.core.bo.**BO**(*model*, *space*, *objective*, *acquisition*, *evaluator*, *X_init*, *Y_init=None*, *cost=None*, *normalize_Y=True*, *model_update_interval=1*, *de_duplication=False*)
Bases: object

Runner of Bayesian optimization loop. This class wraps the optimization loop around the different handlers. :param model: GPyOpt model class. :param space: GPyOpt space class. :param objective: GPyOpt objective class. :param acquisition: GPyOpt acquisition class. :param evaluator: GPyOpt evaluator class. :param X_init: 2d numpy array containing the initial inputs (one per row) of the model. :param Y_init: 2d numpy array containing the initial outputs (one per row) of the model. :param cost: GPyOpt cost class (default, none). :param normalize_Y: whether to normalize the outputs before performing any optimization (default, True). :param model_update_interval: interval of collected observations after which the model is updated (default, 1). :param de_duplication: GPyOpt DuplicateManager class. Avoids re-evaluating the objective at previous, pending or infeasible locations (default, False).

evaluate_objective()
Evaluates the objective

get_evaluations()

plot_acquisition(*filename=None*)

Plots the model and the acquisition function. if self.input_dim = 1: Plots data, mean and variance in one plot and the acquisition function in another plot if self.input_dim = 2: as before but it separates the mean and variance of the model in two different plots

Parameters **filename** – name of the file where the plot is saved

plot_convergence(*filename=None*)

Makes twp plots to evaluate the convergence of the model: plot 1: Iterations vs. distance between consecutive selected x's plot 2: Iterations vs. the mean of the current model in the selected sample.

Parameters **filename** – name of the file where the plot is saved

run_optimization(*max_iter=0*, *max_time=inf*, *eps=1e-08*, *context=None*, *verbosity=False*, *save_models_parameters=True*, *report_file=None*, *evaluations_file=None*, *models_file=None*)

Runs Bayesian Optimization for a number 'max_iter' of iterations (after the initial exploration data)

Parameters

- **max_iter** – exploration horizon, or number of acquisitions. If nothing is provided optimizes the current acquisition.
- **max_time** – maximum exploration horizon in seconds.
- **eps** – minimum distance between two consecutive x's to keep running the model.
- **verbosity** – flag to print the optimization results after each iteration (default, False).
- **report_file** – filename of the file where the results of the optimization are saved (default, None).
- **context** – fixes specified variables to a particular context (values) for the optimization run (default, None).

save_evaluations(*evaluations_file=None*)

Saves a report with the results of the iterations of the optimization

Parameters **evaluations_file** – name of the file in which the results are saved.

save_models(*models_file*)

Saves a report with the results of the iterations of the optimization

Parameters **models_file** – name of the file or a file buffer, in which the results are saved.

save_report(*report_file=None*)

Saves a report with the main resutls of the optimization.

Parameters **report_file** – name of the file in which the results of the optimization are saved.

suggest_next_locations(*context=None*, *pending_X=None*, *ignored_X=None*)

Run a single optimization step and return the next locations to evaluate the objective. Number of suggested locations equals to batch_size.

Parameters

- **context** – fixes specified variables to a particular context (values) for the optimization run (default, None).
- **pending_X** – matrix of input configurations that are in a pending state (i.e., do not have an evaluation yet) (default, None).

- **ignored_X** – matrix of input configurations that the user black-lists, i.e., those configurations will not be suggested again (default, None).

2.4 GPyOpt.core.errors module

exception GPyOpt.core.errors.**FullyExploredOptimizationDomainError**
Bases: exceptions.Exception

exception GPyOpt.core.errors.**InvalidConfigError**
Bases: exceptions.Exception

exception GPyOpt.core.errors.**InvalidVariableNameError**
Bases: exceptions.Exception

2.5 Module contents

CHAPTER 3

GPyOpt.experiment_design package

3.1 Submodules

3.2 GPyOpt.experiment_design.base module

class GPyOpt.experiment_design.base.**ExperimentDesign**(*space*)
Bases: object

Base class for all experiment designs

get_samples(*init_points_count*)

3.3 GPyOpt.experiment_design.grid_design module

class GPyOpt.experiment_design.grid_design.**GridDesign**(*space*)
Bases: *GPyOpt.experiment_design.base.ExperimentDesign*

Grid experiment design. Uses random design for non-continuous variables, and square grid for continuous ones

get_samples(*init_points_count*)
This method may return less points than requested. The total number of generated points is the smallest closest integer of n^d to the selected amount of points.

GPyOpt.experiment_design.grid_design.**iroot**(*k*, *n*)

GPyOpt.experiment_design.grid_design.**multigrid**(*bounds*, *points_count*)
Generates a multidimensional lattice :param bounds: box constraints :param points_count: number of points per dimension.

3.4 GPyOpt.experiment_design.latin_design module

class GPyOpt.experiment_design.latin_design.**LatinDesign**(*space*)
Bases: *GPyOpt.experiment_design.base.ExperimentDesign*

Latin experiment design. Uses random design for non-continuous variables, and latin hypercube for continuous ones

get_samples(*init_points_count*)

3.5 GPyOpt.experiment_design.random_design module

class GPyOpt.experiment_design.random_design.**RandomDesign**(*space*)
Bases: *GPyOpt.experiment_design.base.ExperimentDesign*

Random experiment design. Random values for all variables within the given bounds.

fill_noncontinous_variables(*samples*)
Fill sample values to non-continuous variables in place

get_samples(*init_points_count*)

get_samples_with_constraints(*init_points_count*)
Draw random samples and only save those that satisfy constraints Finish when required number of samples is generated

get_samples_without_constraints(*init_points_count*)

GPyOpt.experiment_design.random_design.**samples_multidimensional_uniform**(*bounds*, *points_count*)
Generates a multidimensional grid uniformly distributed. :param bounds: tuple defining the box constraints. :points_count: number of data points to generate.

3.6 GPyOpt.experiment_design.sobol_design module

class GPyOpt.experiment_design.sobol_design.**SobolDesign**(*space*)
Bases: *GPyOpt.experiment_design.base.ExperimentDesign*

Sobol experiment design. Uses random design for non-continuous variables, and Sobol sequence for continuous ones

get_samples(*init_points_count*)

3.7 Module contents

GPyOpt.experiment_design.**initial_design**(*design_name*, *space*, *init_points_count*)

CHAPTER 4

GPyOpt.interface package

4.1 Submodules

4.2 GPyOpt.interface.config_parser module

GPyOpt.interface.config_parser.**parser**(*input_file_path='config.json'*)
Parser for the .json file containing the configuration of the method.

GPyOpt.interface.config_parser.**update_config**(*config_new*, *config_default*)
Updates the loaded method configuration with default values.

4.3 GPyOpt.interface.driver module

class GPyOpt.interface.driver.**BODriver**(*config=None*, *obj_func=None*, *outputEng=None*)
Bases: object

The class for driving the Bayesian optimization according to the configuration.

run()
Runs the optimization using the previously loaded elements.

4.4 GPyOpt.interface.func_loader module

GPyOpt.interface.func_loader.**load_objective**(*config*)
Loads the objective function from a .json file.

4.5 GPyOpt.interface.output module

class GPyOpt.interface.output.**DataSaver**(*config*, *outpath=None*, *prjname=''*, *name=''*)
Bases: object

close()

save_data(*iters*, *times*, *offsets*, *X*, *Y*, *bo*)

class GPyOpt.interface.output.**Logger**(*config*, *outpath*, *prjname=''*, *name=''*)
Bases: *GPyOpt.interface.output.DataSaver*

close()

save_data(*iters*, *times*, *offsets*, *X*, *Y*, *bo*)

class GPyOpt.interface.output.**OutputEng**(*config*)
Bases: object

append_iter(*iters*, *elapsed_time*, *X*, *Y*, *bo*, *final=False*)

close()

class GPyOpt.interface.output.**Report**(*config*, *outpath*, *prjname=''*, *name=''*)
Bases: *GPyOpt.interface.output.DataSaver*

save_data(*iters*, *times*, *offsets*, *X*, *Y*, *bo*)

4.6 Module contents

CHAPTER 5

GPyOpt.methods package

5.1 Submodules

5.2 GPyOpt.methods.bayesian_optimization module

class GPyOpt.methods.bayesian_optimization.**BayesianOptimization**(*f*, *domain=None*, *constraints=None*, *cost_withGradients=None*, *model_type='GP'*, *X=None*, *Y=None*, *initial_design_numdata=5*, *initial_design_type='random'*, *acquisition_type='EI'*, *normalize_Y=True*, *exact_feval=False*, *acquisition_optimizer_type='lbfgs'*, *model_update_interval=1*, *evaluator_type='sequential'*, *batch_size=1*, *num_cores=1*, *verbosity=False*, *verbosity_model=False*, *maximize=False*, *de_duplication=False*, ***kwargs*)

Bases: *GPyOpt.core.bo.BO*

Main class to initialize a Bayesian Optimization method. :param f: function to optimize. It should take 2-dimensional numpy arrays as input and return 2-dimensional outputs (one evaluation per row). :param domain: list of dictionaries containing the description of the inputs variables (See GPyOpt.core.space.Design_space class for details). :param constraints: list of dictionaries containing the description of the problem constraints (See GPyOpt.core.space.Design_space class for details). :cost_withGradients: cost function of the objective. The input can be:

- a function that returns the cost and the derivatives and any set of points in the domain.
- 'evaluation_time': a Gaussian process (mean) is used to handle the evaluation cost.

Model_type type of model to use as surrogate: - 'GP', standard Gaussian process. - 'GP_MCMC', Gaussian process with prior in the hyper-parameters. - 'sparseGP', sparse Gaussian process. - 'warperdGP', warped Gaussian process. - 'InputWarpedGP', input warped Gaussian process - 'RF', random forest (scikit-learn).

Parameters

- **X** – 2d numpy array containing the initial inputs (one per row) of the model.
- **Y** – 2d numpy array containing the initial outputs (one per row) of the model.
- **normalize_Y** – whether to normalize the outputs before performing any optimization (default, True).
- **model_update_interval** – interval of collected observations after which the model is updated (default, 1).
- **evaluator_type** – determines the way the objective is evaluated (all methods are equivalent if the batch size is one) - 'sequential', sequential evaluations. - 'random': synchronous batch that selects the first element as in a sequential policy and the rest randomly. - 'local_penalization': batch method proposed in (Gonzalez et al. 2016). - 'thompson_sampling': batch method using Thompson sampling.
- **batch_size** – size of the batch in which the objective is evaluated (default, 1).
- **num_cores** – number of cores used to evaluate the objective (default, 1).
- **verbosity** – prints the models and other options during the optimization (default, False).
- **maximize** – when True -f maximization of f is done by minimizing -f (default, False).
- ****kwargs** – extra parameters. Can be used to tune the current optimization setup or to use deprecated options in this package release.

Initial_design_numdata number of initial points that are collected jointly before start running the optimization.

Initial_design_type type of initial design: - 'random', to collect points in random locations. - 'latin', to collect points in a Latin hypercube (discrete variables are sampled randomly.)

Acquisition_type type of acquisition function to use. - 'EI', expected improvement. - 'EI_MCMC', integrated expected improvement (requires GP_MCMC model). - 'MPI', maximum probability of improvement. - 'MPI_MCMC', maximum probability of improvement (requires GP_MCMC model). - 'LCB', GP-Lower confidence bound. - 'LCB_MCMC', integrated GP-Lower confidence bound (requires GP_MCMC model).

Exact_feval whether the outputs are exact (default, False).

Acquisition_optimizer_type type of acquisition function to use. - 'lbfgs': L-BFGS. - 'DIRECT': Dividing Rectangles. - 'CMA': covariance matrix adaptation.

Note: The parameters bounds, kernel, numdata_initial_design, type_initial_design, model_optimize_interval, acquisition, acquisition_par model_optimize_restarts, sparseGP, num_inducing and normalize can still be used but will be deprecated in the next version.

5.3 GPyOpt.methods.modular_bayesian_optimization module

class GPyOpt.methods.modular_bayesian_optimization.**ModularBayesianOptimization**(*model*, *space*, *objective*, *acquisition*, *evaluator*, *X_init*, *Y_init=None*, *cost=None*, *normalize_Y=True*, *model_update_interval=1*, *de_duplication=None*)

Bases: *GPyOpt.core.bo.BO*

Modular Bayesian optimization. This class wraps the optimization loop around the different handlers.

Parameters

- **model** – GPyOpt model class.
- **space** – GPyOpt space class.
- **objective** – GPyOpt objective class.
- **acquisition** – GPyOpt acquisition class.
- **evaluator** – GPyOpt evaluator class.
- **X_init** – 2d numpy array containing the initial inputs (one per row) of the model.
- **Y_init** – 2d numpy array containing the initial outputs (one per row) of the model.
- **cost** – GPyOpt cost class (default, none).
- **normalize_Y** – whether to normalize the outputs before performing any optimization (default, True).
- **model_update_interval** – interval of collected observations after which the model is updated (default, 1).
- **de_duplication** – instantiated de_duplication GPyOpt class.

5.4 Module contents

CHAPTER 6

GPyOpt.models package

6.1 Submodules

6.2 GPyOpt.models.base module

class GPyOpt.models.base.**BOModel**
Bases: object

The abstract Model for Bayesian Optimization

MCMC_sampler = False

analytical_gradient_prediction = False

get_fmin()
Get the minimum of the current model.

predict(*X*)
Get the predicted mean and std at X.

predict_withGradients(*X*)
Get the gradients of the predicted mean and variance at X.

updateModel(*X_all*, *Y_all*, *X_new*, *Y_new*)
Augment the dataset of the model

6.3 GPyOpt.models.gpmodel module

class GPyOpt.models.gpmodel.**GPModel**(*kernel=None*, *noise_var=None*, *exact_feval=False*, *optimizer='bfgs'*, *max_iters=1000*, *optimize_restarts=5*, *sparse=False*, *num_inducing=10*, *verbose=True*, *ARD=False*)
Bases: *GPyOpt.models.base.BOModel*

General class for handling a Gaussian Process in GPyOpt.

Parameters

- **kernel** – GPy kernel to use in the GP model.
- **noise_var** – value of the noise variance if known.
- **exact_feval** – whether noiseless evaluations are available. IMPORTANT to make the optimization work well in noiseless scenarios (default, False).
- **optimizer** – optimizer of the model. Check GPy for details.
- **max_iters** – maximum number of iterations used to optimize the parameters of the model.
- **optimize_restarts** – number of restarts in the optimization.
- **sparse** – whether to use a sparse GP (default, False). This is useful when many observations are available.
- **num_inducing** – number of inducing points if a sparse GP is used.
- **verbose** – print out the model messages (default, False).
- **ARD** – whether ARD is used in the kernel (default, False).

Note: This model does Maximum likelihood estimation of the hyper-parameters.

---

analytical_gradient_prediction = True

copy()
Makes a safe copy of the model.

static fromConfig()

get_fmin()
Returns the location where the posterior mean is takes its minimal value.

get_model_parameters()
Returns a 2D numpy array with the parameters of the model

get_model_parameters_names()
Returns a list with the names of the parameters of the model

predict(*X*)
Predictions with the model. Returns posterior means and standard deviations at X. Note that this is different in GPy where the variances are given.

predict_withGradients(*X*)
Returns the mean, standard deviation, mean gradient and standard deviation gradient at X.

updateModel(*X_all*, *Y_all*, *X_new*, *Y_new*)
Updates the model with new observations.

class GPyOpt.models.gpmodel.**GPModel_MCMC**(*kernel=None*, *noise_var=None*, *exact_feval=False*, *n_samples=10*, *n_burnin=100*, *subsample_interval=10*, *step_size=0.1*, *leapfrog_steps=20*, *verbose=False*)

Bases: *GPyOpt.models.base.BOModel*

General class for handling a Gaussian Process in GPyOpt.

Parameters

- **kernel** – GPy kernel to use in the GP model.
- **noise_var** – value of the noise variance if known.
- **exact_feval** – whether noiseless evaluations are available. IMPORTANT to make the optimization work well in noiseless scenarios (default, False).
- **n_samples** – number of MCMC samples.
- **n_burnin** – number of samples not used.
- **subsample_interval** – sub-sample interval in the MCMC.
- **step_size** – step-size in the MCMC.
- **leapfrog_steps** – ??
- **verbose** – print out the model messages (default, False).

Note: This model does MCMC over the hyperparameters.

MCMC_sampler = True

analytical_gradient_prediction = True

copy()
Makes a safe copy of the model.

get_fmin()
Returns the location where the posterior mean is takes its minimal value.

get_model_parameters()
Returns a 2D numpy array with the parameters of the model

get_model_parameters_names()
Returns a list with the names of the parameters of the model

predict(*X*)
Predictions with the model for all the MCMC samples. Returns posterior means and standard deviations at X. Note that this is different in GPy where the variances are given.

predict_withGradients(*X*)
Returns the mean, standard deviation, mean gradient and standard deviation gradient at X for all the MCMC samples.

updateModel(*X_all*, *Y_all*, *X_new*, *Y_new*)
Updates the model with new observations.

6.4 GPyOpt.models.input_warped_gpmodel module

class GPyOpt.models.input_warped_gpmodel.**InputWarpedGPModel**(*space*, *warping_function=None*, *kernel=None*, *noise_var=None*, *exact_feval=False*, *optimizer='bfgs'*, *max_iters=1000*, *optimize_restarts=5*, *verbose=False*, *ARD=False*)

Bases: *GPyOpt.models.gpmodel.GPModel*

Bayesian Optimization with Input Warped GP using Kumar Warping

The Kumar warping only applies to the numerical variables: continuous and discrete

space [object] Instance of Design_space defined in GPyOpt.core.task.space

warping_function [object, optional] Warping function defined in GPy.util.input_warping_functions.py. Default is Kumar warping

kernel [object, optional] An instance of kernel function defined in GPy.kern. Default is Matern 52

noise_var [float, optional] Value of the noise variance if known

exact_feval [bool, optional] Whether noiseless evaluations are available. IMPORTANT to make the optimization work well in noiseless scenarios, Default is False

optimizer [string, optional] Optimizer of the model. Check GPy for details. Default to bfgs

max_iter [int, optional] Maximum number of iterations used to optimize the parameters of the model. Default is 1000

optimize_restarts [int, optional] Number of restarts in the optimization. Default is 5

verbose [bool, optional] Whether to print out the model messages. Default is False

analytical_gradient_prediction = False

6.5 GPyOpt.models.rfmodel module

class GPyOpt.models.rfmodel.**RFModel**(*bootstrap=True*, *criterion='mse'*, *max_depth=None*, *max_features='auto'*, *max_leaf_nodes=None*, *min_samples_leaf=1*, *min_samples_split=2*, *min_weight_fraction_leaf=0.0*, *n_estimators=500*, *n_jobs=1*, *oob_score=False*, *random_state=None*, *verbose=0*, *warm_start=False*)

Bases: *GPyOpt.models.base.BOModel*

General class for handling a Ramdom Forest in GPyOpt.

Note: The model has beed wrapper 'as it is' from Scikit-learn. Check

http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestRegressor.html for further details.

analytical_gradient_prediction = False

get_fmin()
Get the minimum of the current model.

predict(*X*)
Predictions with the model. Returns posterior means and standard deviations at X.

updateModel(*X_all*, *Y_all*, *X_new*, *Y_new*)
Updates the model with new observations.

6.6 GPyOpt.models.warpedgpmodel module

class GPyOpt.models.warpedgpmodel.**WarpedGPModel**(*kernel=None*, *noise_var=None*, *exact_feval=False*, *optimizer='bfgs'*, *max_iters=1000*, *optimize_restarts=5*, *warping_function=None*, *warping_terms=3*, *verbose=False*)

Bases: *GPyOpt.models.base.BOModel*

analytical_gradient_prediction = False

get_fmin()
Get the minimum of the current model.

predict(*X*)
Get the predicted mean and std at X.

updateModel(*X_all*, *Y_all*, *X_new*, *Y_new*)
Augment the dataset of the model

6.7 Module contents

GPyOpt.models.**select_model**(*name*)

CHAPTER 7

GPyOpt.objective_examples package

7.1 Submodules

7.2 GPyOpt.objective_examples.experiments1d module

class GPyOpt.objective_examples.experiments1d.**forrester**(*sd=None*)
Bases: *GPyOpt.objective_examples.experiments1d.function1d*

Forrester function.

Parameters **sd** – standard deviation, to generate noisy evaluations of the function.

f(*X*)

class GPyOpt.objective_examples.experiments1d.**function1d**
This is a benchmark of unidimensional functions interesting to optimize. :param bounds: the box constraints to define the domain in which the function is optimized.

plot(*bounds=None*)

7.3 GPyOpt.objective_examples.experiments2d module

class GPyOpt.objective_examples.experiments2d.**beale**(*bounds=None*, *sd=None*)
Bases: *GPyOpt.objective_examples.experiments2d.function2d*

Cosines function

Parameters

- **bounds** – the box constraints to define the domain in which the function is optimized.
- **sd** – standard deviation, to generate noisy evaluations of the function.

f(*X*)

class GPyOpt.objective_examples.experiments2d.**branin**(*bounds=None*, *a=None*, *b=None*, *c=None*, *r=None*, *s=None*, *t=None*, *sd=None*)
Bases: *GPyOpt.objective_examples.experiments2d.function2d*

Branin function

Parameters

- **bounds** – the box constraints to define the domain in which the function is optimized.
- **sd** – standard deviation, to generate noisy evaluations of the function.

f(*X*)

class GPyOpt.objective_examples.experiments2d.**cosines**(*bounds=None*, *sd=None*)
Bases: *GPyOpt.objective_examples.experiments2d.function2d*

Cosines function

Parameters

- **bounds** – the box constraints to define the domain in which the function is optimized.
- **sd** – standard deviation, to generate noisy evaluations of the function.

f(*X*)

class GPyOpt.objective_examples.experiments2d.**dropwave**(*bounds=None*, *sd=None*)
Bases: *GPyOpt.objective_examples.experiments2d.function2d*

Cosines function

Parameters

- **bounds** – the box constraints to define the domain in which the function is optimized.
- **sd** – standard deviation, to generate noisy evaluations of the function.

f(*X*)

class GPyOpt.objective_examples.experiments2d.**eggholder**(*bounds=None*, *sd=None*)

f(*X*)

class GPyOpt.objective_examples.experiments2d.**function2d**
This is a benchmark of bi-dimensional functions interesting to optimize.

plot()

class GPyOpt.objective_examples.experiments2d.**goldstein**(*bounds=None*, *sd=None*)
Bases: *GPyOpt.objective_examples.experiments2d.function2d*

Goldstein function

Parameters

- **bounds** – the box constraints to define the domain in which the function is optimized.
- **sd** – standard deviation, to generate noisy evaluations of the function.

f(*X*)

class GPyOpt.objective_examples.experiments2d.**mccormick**(*bounds=None*, *sd=None*)
Bases: *GPyOpt.objective_examples.experiments2d.function2d*

Mccormick function

Parameters

- **bounds** – the box constraints to define the domain in which the function is optimized.
- **sd** – standard deviation, to generate noisy evaluations of the function.

f(*x*)

class GPyOpt.objective_examples.experiments2d.**powers**(*bounds=None*, *sd=None*)
Bases: *GPyOpt.objective_examples.experiments2d.function2d*

Powers function

Parameters

- **bounds** – the box constraints to define the domain in which the function is optimized.
- **sd** – standard deviation, to generate noisy evaluations of the function.

f(*x*)

class GPyOpt.objective_examples.experiments2d.**rosenbrock**(*bounds=None*, *sd=None*)
Bases: *GPyOpt.objective_examples.experiments2d.function2d*

Cosines function

Parameters

- **bounds** – the box constraints to define the domain in which the function is optimized.
- **sd** – standard deviation, to generate noisy evaluations of the function.

f(*X*)

class GPyOpt.objective_examples.experiments2d.**sixhumpcamel**(*bounds=None*, *sd=None*)
Bases: *GPyOpt.objective_examples.experiments2d.function2d*

Six hump camel function

Parameters

- **bounds** – the box constraints to define the domain in which the function is optimized.
- **sd** – standard deviation, to generate noisy evaluations of the function.

f(*x*)

7.4 GPyOpt.objective_examples.experimentsNd module

class GPyOpt.objective_examples.experimentsNd.**ackley**(*input_dim*, *bounds=None*, *sd=None*)
Ackley function

Parameters **sd** – standard deviation, to generate noisy evaluations of the function.

f(*X*)

class GPyOpt.objective_examples.experimentsNd.**alpine1**(*input_dim*, *bounds=None*, *sd=None*)
Alpine1 function

Parameters

- **bounds** – the box constraints to define the domain in which the function is optimized.

- **sd** – standard deviation, to generate noisy evaluations of the function.

f(*X*)

class GPyOpt.objective_examples.experimentsNd.**alpine2**(*input_dim*, *bounds=None*, *sd=None*)

Alpine2 function

Parameters

- **bounds** – the box constraints to define the domain in which the function is optimized.
- **sd** – standard deviation, to generate noisy evaluations of the function.

f(*X*)

class GPyOpt.objective_examples.experimentsNd.**gSobol**(*a*, *bounds=None*, *sd=None*)

gSolbol function

Parameters

- **a** – one-dimensional array containing the coefficients of the function.
- **sd** – standard deviation, to generate noisy evaluations of the function.

f(*X*)

7.5 Module contents

CHAPTER 8

GPyOpt.optimization package

8.1 Submodules

8.2 GPyOpt.optimization.acquisition_optimizer module

class GPyOpt.optimization.acquisition_optimizer.**AcquisitionOptimizer**(*space*, *optimizer='lbfgs'*, ***kwargs*)

Bases: object

General class for acquisition optimizers defined in domains with mix of discrete, continuous, bandit variables

Parameters

- **space** – design space class from GPyOpt.
- **optimizer** – optimizer to use. Can be selected among: - 'lbfgs': L-BFGS. - 'DIRECT': Dividing Rectangles. - 'CMA': covariance matrix adaptation.

optimize(*f=None*, *df=None*, *f_df=None*, *duplicate_manager=None*)

Optimizes the input function.

Parameters

- **f** – function to optimize.
- **df** – gradient of the function to optimize.
- **f_df** – returns both the function to optimize and its gradient.

class GPyOpt.optimization.acquisition_optimizer.**ContextManager**(*space*, *context=None*)

Bases: object

class to handle the context variable in the optimizer :param space: design space class from GPyOpt. :param context: dictionary of variables and their contex values

8.3 GPyOpt.optimization.anchor_points_generator module

class GPyOpt.optimization.anchor_points_generator.**AnchorPointsGenerator**(*space*, *design_type*, *num_samples*)

Bases: object

get(*num_anchor=5*, *duplicate_manager=None*, *unique=False*, *context_manager=None*)

get_anchor_point_scores(*X*)

class GPyOpt.optimization.anchor_points_generator.**ObjectiveAnchorPointsGenerator**(*space*, *design_type*, *objective*, *num_samples*

Bases: *GPyOpt.optimization.anchor_points_generator.AnchorPointsGenerator*

get_anchor_point_scores(*X*)

class GPyOpt.optimization.anchor_points_generator.**RandomAnchorPointsGenerator**(*space*, *design_type*, *num_samples=100*

Bases: *GPyOpt.optimization.anchor_points_generator.AnchorPointsGenerator*

get_anchor_point_scores(*X*)

class GPyOpt.optimization.anchor_points_generator.**ThompsonSamplingAnchorPointsGenerator**(*spa*

design
sign
mod
num

Bases: *GPyOpt.optimization.anchor_points_generator.AnchorPointsGenerator*

get_anchor_point_scores(*X*)

8.4 GPyOpt.optimization.optimizer module

class GPyOpt.optimization.optimizer.**OptCma**(*bounds*, *maxiter=1000*)

Bases: *GPyOpt.optimization.optimizer.Optimizer*

Wrapper the Covariance Matrix Adaptation Evolutionary strategy (CMA-ES) optimization method. It works generating an stochastic search based on multivariate Gaussian samples. Only requires f and the box constraints to work.

optimize(*x0*, *f=None*, *df=None*, *f_df=None*)

Parameters

- **x0** – initial point for a local optimizer.
- **f** – function to optimize.
- **df** – gradient of the function to optimize.
- **f_df** – returns both the function to optimize and its gradient.

class GPyOpt.optimization.optimizer.**OptDirect**(*bounds*, *maxiter=1000*)
Bases: *GPyOpt.optimization.optimizer.Optimizer*

Wrapper for DIRECT optimization method. It works partitioning iteratively the domain of the function. Only requires f and the box constraints to work.

optimize(*x0*, *f=None*, *df=None*, *f_df=None*)

Parameters

- **x0** – initial point for a local optimizer.
- **f** – function to optimize.
- **df** – gradient of the function to optimize.
- **f_df** – returns both the function to optimize and its gradient.

class GPyOpt.optimization.optimizer.**OptLbfgs**(*bounds*, *maxiter=1000*)
Bases: *GPyOpt.optimization.optimizer.Optimizer*

Wrapper for l-bfgs-b to use the true or the approximate gradients.

optimize(*x0*, *f=None*, *df=None*, *f_df=None*)

Parameters

- **x0** – initial point for a local optimizer.
- **f** – function to optimize.
- **df** – gradient of the function to optimize.
- **f_df** – returns both the function to optimize and its gradient.

class GPyOpt.optimization.optimizer.**OptimizationWithContext**(*x0*, *f*, *df=None*, *f_df=None*, *context_manager=None*)
Bases: object

df_nc(*x*)
Wrapper of the derivative of *f*: takes an input x with size of the not fixed dimensions expands it and evaluates the gradient of the entire function.

f_df_nc(*x*)
Wrapper of the derivative of *f*: takes an input x with size of the not fixed dimensions expands it and evaluates the gradient of the entire function.

f_nc(*x*)
Wrapper of *f*: takes an input x with size of the noncontext dimensions expands it and evaluates the entire function.

class GPyOpt.optimization.optimizer.**Optimizer**(*bounds*)
Bases: object

Class for a general acquisition optimizer.

Parameters **bounds** – list of tuple with bounds of the optimizer

optimize(*x0*, *f=None*, *df=None*, *f_df=None*)

Parameters

- **x0** – initial point for a local optimizer.
- **f** – function to optimize.
- **df** – gradient of the function to optimize.

- **f_df** – returns both the function to optimize and its gradient.

GPyOpt.optimization.optimizer.**apply_optimizer**(*optimizer*, *x0*, *f=None*, *df=None*, *f_df=None*, *duplicate_manager=None*, *context_manager=None*, *space=None*)

Parameters

- **x0** – initial point for a local optimizer (x0 can be defined with or without the context included).
- **f** – function to optimize.
- **df** – gradient of the function to optimize.
- **f_df** – returns both the function to optimize and its gradient.
- **duplicate_manager** – logic to check for duplicate (always operates in the full space, context included)
- **context_manager** – If provided, x0 (and the optimizer) operates in the space without the context
- **space** – GPyOpt class design space.

GPyOpt.optimization.optimizer.**choose_optimizer**(*optimizer_name*, *bounds*)

Selects the type of local optimizer

8.5 Module contents

CHAPTER 9

GPyOpt.plotting package

9.1 Submodules

9.2 GPyOpt.plotting.plots_bo module

GPyOpt.plotting.plots_bo.**plot_acquisition**(*bounds*, *input_dim*, *model*, *Xdata*, *Ydata*, *acquisition_function*, *suggested_sample*, *filename=None*)

Plots of the model and the acquisition function in 1D and 2D examples.

GPyOpt.plotting.plots_bo.**plot_convergence**(*Xdata*, *best_Y*, *filename=None*)

Plots to evaluate the convergence of standard Bayesian optimization algorithms

9.3 Module contents

CHAPTER 10

GPyOpt.util package

10.1 Submodules

10.2 GPyOpt.util.arguments_manager module

class GPyOpt.util.arguments_manager.**ArgumentsManager**(*kwargs*)

Bases: object

Class to handle extra configurations in the definition of the BayesianOptimization class

acquisition_creator(*acquisition_type*, *model*, *space*, *acquisition_optimizer*, *cost_withGradients*)

Acquisition chooser from the available options. Extra parameters can be passed via **kwargs.

evaluator_creator(*evaluator_type*, *acquisition*, *batch_size*, *model_type*, *model*, *space*, *acquisition_optimizer*)

Acquisition chooser from the available options. Guide the optimization through sequential or parallel evalutions of the objective.

model_creator(*model_type*, *exact_feval*, *space*)

Model chooser from the available options. Extra parameters can be passed via **kwargs.

10.3 GPyOpt.util.duplicate_manager module

class GPyOpt.util.duplicate_manager.**DuplicateManager**(*space*, *zipped_X*, *pending_zipped_X=None*, *ignored_zipped_X=None*)

Bases: object

Class to manage potential duplicates in the suggested candidates.

Parameters

- **space** – object managing all the logic related the domain of the optimization

- **zipped_X** – matrix of evaluated configurations
- **pending_zipped_X** – matrix of configurations in the pending state
- **ignored_zipped_X** – matrix of configurations that the user desires to ignore (e.g., because they may have led to failures)

is_unzipped_x_duplicate(*unzipped_x*)
param: unzipped_x: configuration assumed to be unzipped

is_zipped_x_duplicate(*zipped_x*)
param: zipped_x: configuration assumed to be zipped

10.4 GPyOpt.util.general module

GPyOpt.util.general.**best_guess**(*f*, *X*)
Gets the best current guess from a vector. :param f: function to evaluate. :param X: locations.

GPyOpt.util.general.**best_value**(*Y*, *sign=1*)
Returns a vector whose components i are the minimum (default) or maximum of Y[:i]

GPyOpt.util.general.**compute_integrated_acquisition**(*acquisition*, *x*)
Used to compute the acquisition function when samples of the hyper-parameters have been generated (used in GP_MCMC model).

Parameters

- **acquisition** – acquisition function with GpyOpt model type GP_MCMC.
- **x** – location where the acquisition is evaluated.

GPyOpt.util.general.**compute_integrated_acquisition_withGradients**(*acquisition*, *x*)
Used to compute the acquisition function with gradients when samples of the hyper-parameters have been generated (used in GP_MCMC model).

Parameters

- **acquisition** – acquisition function with GpyOpt model type GP_MCMC.
- **x** – location where the acquisition is evaluated.

GPyOpt.util.general.**evaluate_function**(*f*, *X*)
Returns the evaluation of a function *f* and the time per evaluation

GPyOpt.util.general.**get_d_moments**(*model*, *x*)
Gradients with respect to x of the moments (mean and sdev.) of the GP :param model: GPy model. :param x: location where the gradients are evaluated.

GPyOpt.util.general.**get_moments**(*model*, *x*)
Moments (mean and sdev.) of a GP model at x

GPyOpt.util.general.**get_quantiles**(*acquisition_par*, *fmin*, *m*, *s*)
Quantiles of the Gaussian distribution useful to determine the acquisition function values :param acquisition_par: parameter of the acquisition function :param fmin: current minimum. :param m: vector of means. :param s: vector of standard deviations.

GPyOpt.util.general.**merge_values**(*values1*, *values2*)
Merges two numpy arrays by calculating all possible combinations of rows

GPyOpt.util.general.**normalize**(*Y*, *normalization_type='stats'*)
Normalize the vector Y using statistics or its range.

Parameters

- **Y** – Row or column vector that you want to normalize.
- **normalization_type** – String specifying the kind of normalization

to use. Options are 'stats' to use mean and standard deviation, or 'maxmin' to use the range of function values. :return Y_normalized: The normalized vector.

GPyOpt.util.general.**reshape**(*x*, *input_dim*)
Reshapes x into a matrix with input_dim columns

GPyOpt.util.general.**samples_multidimensional_uniform**(*bounds*, *num_data*)
Generates a multidimensional grid uniformly distributed. :param bounds: tuple defining the box constraints. :num_data: number of data points to generate.

GPyOpt.util.general.**spawn**(*f*)
Function for parallel evaluation of the acquisition function

GPyOpt.util.general.**values_to_array**(*input_values*)
Transforms a values of int, float and tuples to a column vector numpy array

10.5 GPyOpt.util.io module

GPyOpt.util.io.**gen_datestr**()
Returns a string with the yy/mm/dd and hh/mm/ss

10.6 GPyOpt.util.stats module

10.7 Module contents

CHAPTER 11

Indices and tables

- genindex
- modindex
- search

Python Module Index

g

Index

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W

Z