MetaOpt Documentation

Release 0.1.0

Renke Grunwald, Justin Heinermann, Bengt Lüers, Jendrik Poloc

July 23, 2014

Contents

1 Getting Started **3**

1.1 Quick Overview . . . 3
1.2 Installation . . . 4
1.3 Objective functions . . . 4
1.4 Optimization . . . 5
1.5 Optimizers . . . 6
1.6 Invokers . . . 7
1.7 Plugins . . . 8

2 Indices and tables **11**

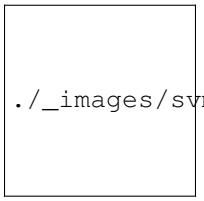
MetaOpt is a library that optimizes black-box functions using a limited amount of time and utilizing multiple processors. The main focus of MetaOpt is the parameter tuning for machine learning and heuristic optimization.

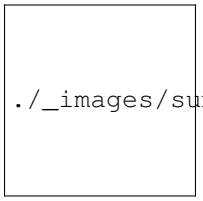
```
from metaopt.core.paramspec.util import param
from metaopt.core.optimize.optimize import optimize

@param.float("a", interval=[-1, 1])
@param.float("b", interval=[0, 1])
def f(a, b):
    return a**2 + b**2

if __name__ == '__main__':
    args = optimize(f, timeout=60)
    print(args);
```

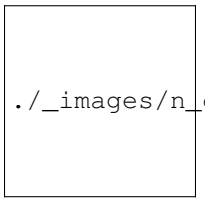
MetaOpt has the following notable features:

- Custom optimization algorithms
- Custom invocation strategies
- A hook-based plugin system
- Different levels of timeouts

To get started with MetaOpt, we recommend reading the following sections. For some examples see below or visit the *examples* page.







CHAPTER 1

Getting Started

1.1 Quick Overview

The example below shows how MetaOpt can be used to find optimal arguments for a (rather simple) objective function `f`. MetaOpt needs no information about the function `f` other than it has two float parameters `a` and `b` (taking values between -1 and 1) and returns *some* value.

```
from metaopt.core.paramspec.util import param
from metaopt.core.optimize.optimize import optimize

@param.float("a", interval=[-1, 1], step=0.02)
@param.float("b", interval=[0, 1], step=0.01)
def f(a, b):
    return a**2 + b**2

if __name__ == '__main__':
    args = optimize(f, timeout=60)
```

MetaOpt will optimize `f` in parellel and finally return a list of arguments for which `f` is minimal.

By default, MetaOpt uses the `metaopt.optimizer.saes.SAESOptimizer` as optimizer. Other optimizers like the `metaopt.optimizer.gridsearch.GridSearchOptimizer` may also be used.

```
from metaopt.optimizer.gridsearch import GridSearchOptimizer

args = optimize(f, optimizer=GridSearchOptimizer())
```

To limit the time an optimizer has to optimize `f` we can pass a timeout in seconds.

```
from metaopt.optimizer.gridsearch import GridSearchOptimizer

args = optimize(f, timeout=60, optimizer=GridSearchOptimizer())
```

MetaOpt will return the optimal arguments it found after 60 seconds.

To also limit the time of an individual computation of `f` we can pass a timeout in seconds by using the `metaopt.plugin.timeout.TimeoutPlugin`.

```
from metaopt.optimizer.gridsearch import GridSearchOptimizer
from metaopt.plugin.timeout import TimeoutPlugin

args = optimize(f, timeout=60, optimizer=GridSearchOptimizer(),
                plugins=[TimeoutPlugin(5)])
```

MetaOpt will abort computations of `f` that take longer than 5 seconds and return the optimal arguments it found after 60 seconds.

This should explain the most basic use cases of MetaOpt. For more details we also recommend reading the next sections.

1.2 Installation

MetaOpt is available on PyPI and can be installed via the following command:

```
$ sudo pip install metaopt
```

1.3 Objective functions

Before MetaOpt can be used, an objective function has to be defined. This includes a specification of its parameters (and optionally its return values). An objective function in MetaOpt is just a regular Python function that is augmented with a number of available that describe its parameters (and return values).

Note: In MetaOpt formal parameters and actual parameters are called *parameters* and *arguments*, respectively.

Let's say a particular objective function takes three parameters `a`, `b` and `g`, where `a` is an integer between -10 and 10, `b` is a float between 0.0 and 1.0 and `g` is a boolean. To specify these, we use the following decorators on the objective function:

- `metaopt.core.paramspec.util.param.int()`
- `metaopt.core.paramspec.util.param.float()`
- `metaopt.core.paramspec.util.param.bool()`

The decorators have to be specified in the same order as the function parameters and should have the same name (the first parameter of the decorator). The topmost decorator has to specify the leftmost function parameter (preferably using the same name) and so forth.

```
from metaopt.core.paramspec.util import param

@param.int("a", interval=[-10, 10], title="α")
@param.float("b", interval=[0, 1], title="β")
@param.bool("g", title="γ")
def f(a, b, g):
    return some_expensive_computation(a, b, g)
```

The `some_expensive_computation` stands for anything from doing I/O to number crunching. If the computation was successful, it should return a numeric value. If the computation was *not* successful (e.g. no value can be computed for the given arguments), it should raise a meaningful exception. You should also make sure that your objective function can run in parallel by avoiding global state and similar things.

MetaOpt can both maximize and minimize objective functions. To do this, the following decorators can be used:

- `metaopt.core.returnspec.util.decorator.maximize()`
- `metaopt.core.returnspec.util.decorator.minimize()`

For example, to maximize the objective function, we use the *maximize* decorator as follows.

```
from metaopt.core.returnspec.util.decorator import maximize
from metaopt.core.paramspec.util import param

@maximize("Fitness") # Also give the return value a descriptive name
@param.int("a", interval=[-10, 10])
@param.float("b", interval=[0, 1])
@param.bool("g", interval=[0, 1])
def f(a, b, g):
    return some_expensive_computation(a, b, g)
```

By default, MetaOpt minimizes objective functions and using *minimize* is only required to give the return value a descriptive name.

To also give parameters more descriptive names, the `title` is supported by all parameter decorators. The title will be displayed instead of the name whenever its parameter or argument is shown to the user.

```
from metaopt.core import param

@param.int("a", interval=[-10, 10], title="α")
@param.float("b", interval=[0, 1], title="β")
@param.bool("g", interval=[0, 1], title="γ")
def f(a, b, g):
    return some_expensive_computation(a, b, g)
```

The following parameter decorators are available in MetaOpt. More types may be added in future versions of MetaOpt.

`metaopt.core.paramspec.util.param.`**`int`**`(*vargs, **kwargs)`
A decorator that specifies an int parameter for a function.

See `metaopt.core.paramspec.paramspec.ParamSpec.int()` for the allowed parameters.

`metaopt.core.paramspec.util.param.`**`float`**`(*vargs, **kwargs)`
A decorator that specifies a float parameter for a function.

See `metaopt.core.paramspec.paramspec.ParamSpec.float()` for the allowed parameters.

`metaopt.core.paramspec.util.param.`**`bool`**`(*vargs, **kwargs)`
A decorator that specifies an bool parameter for a function.

See `metaopt.core.paramspec.paramspec.ParamSpec.bool()` for the allowed parameters.

`metaopt.core.paramspec.util.param.`**`multi`**`(other_decorator, names=[], titles=[], *vargs, **kwargs)`
A decorator that specifies multiple parameters of the same type.

All parameters share the same attributes expect for their names and titles which are specified as list of names and titles. The number of parameters added is derived from the length of the former list.

1.4 Optimization

With an objective function defined it can be finally optimized. Given an objective function `f`, the easiest way to optimize it is using the function `metaopt.core.optimize.optimize.optimize()`.

```
from metaopt.core.optimize.optimize import optimize

args = optimize(f)
```

The result of `optimize` is a list of arguments (see `metaopt.core.arg.arg.Arg`) for which the objective function is optimal (either minimial or maximal).

MetaOpt runs multiple computations of the objective functions in parallel (via the `metaopt.concurrent.invoker.multiprocess.MultiProcessInvoker`). However, other invokers can choose different ways to compute objective functions. For more details, see *Invokers*.

By default, MetaOpt uses `metaopt.optimizer.saes.SAESOptimizer` for the optimization. Other optimizers can be selected by passing them to `optimize` (see *Optimizers*).

```
from metaopt.core.optimize.optimize import optimize
from metaopt.optimizer.gridsearch import GridSearchOptimizer

args = optimize(f, optimizer=GridSearchOptimizer())
```

Generally, to optimize an objective function use a suitable function from below.

`metaopt.core.optimize.optimize.`**`optimize`**(*f*, *timeout=None*, *plugins=[]*, *optimizer=SAESOptimizer()*)

Optimizes the given objective function.

Parameters

- **f** – Objective function
- **timeout** – Available time for optimization (in seconds)
- **plugins** – List of plugins
- **optimizer** – Optimizer

`metaopt.core.optimize.optimize.`**`custom_optimize`**(*f*, *invoker*, *timeout=None*, *optimizer=SAESOptimizer()*)

Optimizes the given objective function using the specified invoker.

Parameters

- **f** – Objective function
- **invoker** – Invoker
- **timeout** – Available time for optimization (in seconds)
- **optimizer** – Optimizer

1.5 Optimizers

Optimizers take an objective function and a specification of its parameters and then try to find the optimal parameters. Optimizers are using invokers to actually compute the result of objective functions and are therefore by default parallelized.

MetaOpt comes a range of built-in optimizers that are suitable for most types of objective functions. These are listed below.

class `metaopt.optimizer.cmaes.`**`CMAESOptimizer`**(*mu=15*, *lamb=100*, *global_step_size=1.0*)

Optimization based on the (mu, lambda)-CMA-ES.

This optimizer should be combined with a global timeout, otherwise it will run indefinitely.

Parameters

- **mu** – Number of parent arguments

- **lamb** – Number of offspring arguments

class metaopt.optimizer.gridsearch.**GridSearchOptimizer**
Optimizer that systematically tests parameters in a grid pattern.

class metaopt.optimizer.randomsearch.**RandomSearchOptimizer**
Optimizer that randomly tests parameters.

This optimizer should be combined with a global timeout, otherwise it will run indefinitely.

class metaopt.optimizer.rechenberg.**RechenbergOptimizer**(*mu=15*, *lamb=100*, *a=0.1*)
Optimization based on an ES using Rechenberg's 1/5th success rule

This optimizer should be combined with a global timeout, otherwise it will run indefinitely.

Parameters

- **mu** – Number of parent arguments
- **lamb** – Number of offspring arguments

class metaopt.optimizer.saes.**SAESOptimizer**(*mu=15*, *lamb=100*, *tau0=None*, *tau1=None*)
Optimization based on a self-adaptive evolution strategy (SAES)

This optimizer should be combined with a global timeout, otherwise it will run indefinitely.

Parameters

- **mu** – Number of parent arguments
- **lamb** – Number of offspring arguments

1.6 Invokers

To find optimal parameters MetaOpt has to compare the results of objective function computations for various differing arguments. While optimizers decide for which arguments the objective function is computed, invokers choose the way *how* it is actually computed (or as we call it: invoked).

Other invokers can be used by passing them to `metaopt.core.optimize.optimize.custom_optimize()`.

```
from metaopt.core.optimize.optimize import custom_optimize
from metaopt.concurrent.invoker.multiprocess import MultiProcessInvoker

args = custom_optimize(f, invoker=MultiProcessInvoker())
```

The following invokers are available in MetaOpt.

class metaopt.concurrent.invoker.multiprocess.**MultiProcessInvoker**(*resources=None*)
Invoker that invokes objective functions in parallel using processes.

Parameters **resources** – Number of CPUs to use at most. Will automatically

configure itself, if None.

class metaopt.concurrent.invoker.pluggable.**PluggableInvoker**(*invoker*, *plugins=[]*)
Invoker that uses other invokers and allows plugins to be used.

Parameters

- **invoker** – Other invoker
- **plugins** – List of plugins

1.7 Plugins

Plugins change the way objective functions are invoked. They attach handlers to various events that occur when an optimizer wants to invoke an objective functions via the `metaopt.concurrent.invoker.pluggable.PluggableInvoker`.

Plugins can be used by passing them to `metaopt.core.optimize.optimize.optimize()`.

```
from metaopt.core.optimize.optimize import optimize

from metaopt.plugin.timeout import TimeoutPlugin
from metaopt.plugin.print.status import StatusPrintPlugin

args = optimize(f, plugins=[StatusPrintPlugin(), TimeoutPlugin(2)])
```

If you use your own custom invoker, `metaopt.core.optimize.optimize.custom_optimize()` can be used.

```
from metaopt.core.optimize.optimize import optimize
from metaopt.concurrent.invoker.pluggable import PluggableInvoker

from metaopt.plugin.timeout import TimeoutPlugin
from metaopt.plugin.print.status import StatusPrintPlugin

invoker = PluggableInvoker(CustomInvoker(),
                           plugins=[PrintPlugin(), TimeoutPlugin(2)])

args = custom_optimize(f, invoker=invoker)
```

The following plugins are available in MetaOpt.

class metaopt.plugin.timeout.**TimeoutPlugin**(*timeout*)
Abort an invocation after a certain amount of time.

Use this plugin for objective functions that may take too long to compute a result for certain parameters.

Parameters **timeout** – Available time for invocation (in seconds)

class metaopt.plugin.print.status.**StatusPrintPlugin**
Logs all invocation events to the standard output.

For example:

```
Started f(a=0.1, b=0.2)
Started f(a=0.2, b=0.3)
Finished f(a=0.1, b=0.2) = 0.7
Failed f(a=0.2, b=0.3)
```

class metaopt.plugin.visualization.landscape.**VisualizeLandscapePlugin**(*x_param_index=0*, *y_param_index=1*)
Visualize fitness landscape

show_image_plot()
Show an image plot

show_surface_plot()
Show a surface plot

class metaopt.plugin.visualization.best_fitness.**VisualizeBestFitnessPlugin**
Visualize optimization progess

show_fitness_invocations_plot()
Show a fitness–invocations plot

show_fitness_time_plot()
Show a fitness–time plot

CHAPTER 2

Indices and tables

- *genindex*
- *modindex*
- *search*