Hyperopt: A Python Library for Optimizing the Hyperparameters of Machine Learning Algorithms

James Bergstra[‡*], Dan Yamins[§], David D. Cox[¶]

http://www.youtube.com/watch?v=Mp1xnPfE4PY

Abstract—Sequential model-based optimization (also known as Bayesian optimization) is one of the most efficient methods (per function evaluation) of function minimization. This efficiency makes it appropriate for optimizing the hyperparameters of machine learning algorithms that are slow to train. The Hyperopt library provides algorithms and parallelization infrastructure for performing hyperparameter optimization (model selection) in Python. This paper presents an introductory tutorial on the usage of the Hyperopt library, including the description of search spaces, minimization (in serial and parallel), and the analysis of the results collected in the course of minimization. The paper closes with some discussion of ongoing and future work.

Index Terms—Bayesian optimization, hyperparameter optimization, model selection

Introduction

Sequential model-based optimization (SMBO, also known as Bayesian optimization) is a general technique for function optimization that includes some of the most call-efficient (in terms of function evaluations) optimization methods currently available. Originally developed for experiment design (and oil exploration, [Mockus78]) SMBO methods are generally applicable to scenarios in which a user wishes to minimize some scalar-valued function $f(x)$ that is costly to evaluate, often in terms of time or money. Compared with standard optimization strategies such as conjugate gradient descent methods, model-based optimization algorithms invest more time between function evaluations in order to reduce the number of function evaluations overall.

The advantages of SMBO are that it:

- leverages smoothness without analytic gradient,
- handles real-valued, discrete, and conditional variables,
- handles parallel evaluations of $f(x)$,
- copes with hundreds of variables, even with budget of just a few hundred function evaluations.

Many widely-used machine learning algorithms take a significant amount of time to train from data. At the same time, these same algorithms must be configured prior to training. These configuration variables are called *hyperparameters*. For example, Support Vector Machines (SVMs) have hyperparameters that include the regularization strength (often C) the scaling of input data (and more generally, the preprocessing of input data), the choice of similarity kernel, and the various parameters that are specific to that kernel choice. Decision trees are another machine learning algorithm with hyperparameters related to the heuristic for creating internal nodes, and the pruning strategy for the tree after (or during) training. Neural networks are a classic type of machine learning algorithm but they have so many hyperparameters that they have been considered too troublesome for inclusion in the sklearn library.

Hyperparameters generally have a significant effect on the success of machine learning algorithms. A poorly-configured SVM may perform no better than chance, while a well-configured one may achieve state-of-the-art prediction accuracy. To experts and non-experts alike, adjusting hyperparameters to optimize end-to-end performance can be a tedious and difficult task. Hyperparameters come in many varieties---continuous-valued ones with and without bounds, discrete ones that are either ordered or not, and conditional ones that do not even always apply (e.g., the parameters of an optional pre-processing stage). Because of this variety, conventional continuous and combinatorial optimization algorithms either do not directly apply, or else operate without leveraging valuable structure in the configuration space. Common practice for the optimization of hyperparameters is (a) for algorithm developers to tune them by hand on representative problems to get good rules of thumb and default values, and (b) for algorithm users to tune them manually for their particular prediction problems, perhaps with the assistance of [multi-resolution] grid search. However, when dealing with more than a few hyperparameters (e.g. 5) this standard practice of manual search with grid refinement is not guaranteed to work well; in such cases even random search has been shown to be competitive with domain experts [BB12].

Hyperopt [Hyperopt] provides algorithms and software infrastructure for carrying out hyperparameter optimization for machine learning algorithms. Hyperopt provides an optimization interface that distinguishes a *configuration space* and an *evaluation function* that assigns real-valued *loss values* to points within the configuration space. Unlike the standard minimization interfaces provided by scientific programming libraries, Hyperopt's `fmin` interface requires users to specify the configuration space as a probability distribution. Specifying a probability distribution rather than just bounds and hard constraints allows domain experts to encode more

* *Corresponding author: james.bergstra@uwaterloo.ca*
‡ *University of Waterloo*
§ *Massachusetts Institute of Technology*
¶ *Harvard University*

of their intuitions regarding which values are plausible for various hyperparameters. Like SciPy's `optimize.minimize` interface, Hyperopt makes the SMBO algorithm itself an interchangeable component so that any search algorithm can be applied to any search problem. Currently two algorithms are provided -- random search and Tree-of-Parzen-Estimators (TPE) algorithm introduced in [BBBK11] -- and more algorithms are planned (including simulated annealing, [SMAC], and Gaussian-process-based [SLA13]).

We are motivated to make hyperparameter optimization more reliable for four reasons:

Reproducibile research
: Hyperopt formalizes the practice of model evaluation, so that benchmarking experiments can be reproduced at later dates, and by different people.

Empowering users
: Learning algorithm designers can deliver flexible fully-configurable implementations to non-experts (e.g. deep learning systems), so long as they also provide a corresponding Hyperopt driver.

Designing better algorithms
: As algorithm designers, we appreciate Hyperopt's capacity to find successful configurations that we might not have considered.

Fuzz testing
: As algorithm designers, we appreciate Hyperopt's capacity to find failure modes via configurations that we had not considered.

This paper describes the usage and architecture of Hyperopt, for both sequential and parallel optimization of expensive functions. Hyperopt can in principle be used for any SMBO problem, but our development and testing efforts have been limited so far to the optimization of hyperparameters for deep neural networks [hp-dbn] and convolutional neural networks for object recognition [hp-convnet].

Getting Started with Hyperopt

This section introduces basic usage of the `hyperopt.fmin` function, which is Hyperopt's basic optimization driver. We will look at how to write an objective function that `fmin` can optimize, and how to describe a configuration space that `fmin` can search.

Hyperopt shoulders the responsibility of finding the best value of a scalar-valued, possibly-stochastic function over a set of possible arguments to that function. Whereas most optimization packages assume that these inputs are drawn from a vector space, Hyperopt encourages you, the user, to describe your configuration space in more detail. Hyperopt is typically aimed at very difficult search settings, especially ones with many hyperparameters and a small budget for function evaluations. By providing more information about where your function is defined, and where you think the best values are, you allow algorithms in Hyperopt to search more efficiently.

The way to use Hyperopt is to describe:

- the objective function to minimize
- the space over which to search
- a trials database [optional]
- the search algorithm to use [optional]

This section will explain how to describe the objective function, configuration space, and optimization algorithm. Later, Section Trial results: more than just the loss will explain how to use the trials database to analyze the results of a search, and Section Parallel Evaluation with a Cluster will explain how to use parallel computation to search faster.

Step 1: define an objective function

Hyperopt provides a few levels of increasing flexibility / complexity when it comes to specifying an objective function to minimize. In the simplest case, an objective function is a Python function that accepts a single argument that stands for x (which can be an arbitrary object), and returns a single scalar value that represents the *loss* ($f(x)$) incurred by that argument.

So for a trivial example, if we want to minimize a quadratic function $q(x,y) := x^2 + y^2$ then we could define our objective `q` as follows:

```
def q(args):
    x, y = args
    return x ** 2 + y ** 2
```

Although Hyperopt accepts objective functions that are more complex in both the arguments they accept and their return value, we will use this simple calling and return convention for the next few sections that introduce configuration spaces, optimization algorithms, and basic usage of the `fmin` interface. Later, as we explain how to use the Trials object to analyze search results, and how to search in parallel with a cluster, we will introduce different calling and return conventions.

Step 2: define a configuration space

A *configuration space* object describes the domain over which Hyperopt is allowed to search. If we want to search q over values of $x \in [0,1]$, and values of $y \in \mathbb{R}$, then we can write our search space as:

```
from hyperopt import hp

space = [hp.uniform('x', 0, 1), hp.normal('y', 0, 1)]
```

Note that for both x and y we have specified not only the hard bound constraints, but also we have given Hyperopt an idea of what range of values for y to prioritize.

Step 3: choose a search algorithm

Choosing the search algorithm is currently as simple as passing `algo=hyperopt.tpe.suggest` or `algo=hyperopt.rand.suggest` as a keyword argument to `hyperopt.fmin`. To use random search to our search problem we can type:

```
from hyperopt import hp, fmin, rand, tpe, space_eval
best = fmin(q, space, algo=rand.suggest)
print best
# =>  XXX
print space_eval(space, best)
# =>  XXX

best = fmin(q, space, algo=tpe.suggest)
print best
# =>  XXX
print space_eval(space, best)
# =>  XXX
```

The search algorithms are global functions which may generally have extra keyword arguments that control their operation beyond the ones used by `fmin` (they represent hyper-hyperparameters!). The intention is that these hyper-hyperparameters are set to default that work for a range of configuration problems, but if you wish to change them you can do it like this:

```
from functools import partial
from hyperopt import hp, fmin, tpe
algo = partial(tpe.suggest, n_startup_jobs=10)
best = fmin(q, space, algo=algo)
print best
# =>  XXX
```

In a nutshell, these are the steps to using Hyperopt. Implement an objective function that maps configuration points to a real-valued loss value, define a configuration space of valid configuration points, and then call `fmin` to search the space to optimize the objective function. The remainder of the paper describes (a) how to describe more elaborate configuration spaces, especially ones that enable more efficient search by expressing *conditional variables*, (b) how to analyze the results of a search as stored in a `Trials` object, and (c) how to use a cluster of computers to search in parallel.

Configuration Spaces

Part of what makes Hyperopt a good fit for optimizing machine learning hyperparameters is that it can optimize over general Python objects, not just e.g. vector spaces. Consider the simple function `w` below, which optimizes over dictionaries with `'type'` and either `'x'` and `'y'` keys:

```
def w(pos):
    if pos['use_var'] == 'x':
        return pos['x'] ** 2
    else:
        return math.exp(pos['y'])
```

To be efficient about optimizing `w` we must be able to (a) describe the kinds of dictionaries that `w` requires and (b) correctly associate `w`'s return value to the elements of `pos` that actually contributed to that return value. Hyperopt's configuration space description objects address both of these requirements. This section describes the nature of configuration space description objects, and how the description language can be extended with new expressions, and how the `choice` expression supports the creation of *conditional variables* that support efficient evaluation of structured search spaces of the sort we need to optimize `w`.

Configuration space primitives

A search space is a stochastic expression that always evaluates to a valid input argument for your objective function. A search space consists of nested function expressions. The stochastic expressions are the hyperparameters. (Random search is implemented by simply sampling these stochastic expressions.)

The stochastic expressions currently recognized by Hyperopt's optimization algorithms are in the `hyperopt.hp` module. The simplest kind of search spaces are ones that are not nested at all. For example, to optimize the simple function `q` (defined above) on the interval $[0,1]$, we could type `fmin(q, space=hp.uniform('a', 0, 1))`.

The first argument to `hp.uniform` here is the *label*. Each of the hyperparameters in a configuration space must be labeled like this with a unique string. The other hyperparameter distributions at our disposal as modelers are as follows:

`hp.choice(label, options)`
: Returns one of the options, which should be a list or tuple. The elements of `options` can themselves be [nested] stochastic expressions. In this case, the stochastic choices that only appear in some of the options become *conditional* parameters.

`hp.pchoice(label, p_options)`
: Return one of the `option` terms listed in `p_options`, a list of pairs `(prob, option)` in which the sum of all `prob` elements should sum to 1. The `pchoice` lets a user bias random search to choose some options more often than others.

`hp.uniform(label, low, high)`
: Draws uniformly between `low` and `high`. When optimizing, this variable is constrained to a two-sided interval.

`hp.quniform(label, low, high, q)`
: Drawn by `round(uniform(low, high) / q) * q`, Suitable for a discrete value with respect to which the objective is still somewhat smooth.

`hp.loguniform(label, low, high)`
: Drawn by `exp(uniform(low, high))`. When optimizing, this variable is constrained to the interval $[e^{\text{low}}, e^{\text{high}}]$.

`hp.qloguniform(label, low, high, q)`
: Drawn by `round(exp(uniform(low, high)) / q) * q`. Suitable for a discrete variable with respect to which the objective is smooth and gets smoother with the increasing size of the value.

`hp.normal(label, mu, sigma)`
: Draws a normally-distributed real value. When optimizing, this is an unconstrained variable.

`hp.qnormal(label, mu, sigma, q)`
: Drawn by `round(normal(mu, sigma) / q) * q`. Suitable for a discrete variable that probably takes a value around mu, but is technically unbounded.

`hp.lognormal(label, mu, sigma)`
: Drawn by `exp(normal(mu, sigma))`. When optimizing, this variable is constrained to be positive.

`hp.qlognormal(label, mu, sigma, q)`
: Drawn by `round(exp(normal(mu, sigma)) / q) * q`. Suitable for a discrete variable with respect to which the objective is smooth and gets smoother with the size of the variable, which is non-negative.

`hp.randint(label, upper)`
: Returns a random integer in the range $[0, upper)$. In contrast to `quniform` optimization algorithms should assume *no* additional correlation in the loss function between nearby integer values, as compared with more distant integer values (e.g. random seeds).

Structure in configuration spaces

Search spaces can also include lists, and dictionaries. Using these containers make it possible for a search space to include multiple variables (hyperparameters). The following code fragment illustrates the syntax:

```
from hyperopt import hp

list_space = [
    hp.uniform('a', 0, 1),
    hp.loguniform('b', 0, 1)]

tuple_space = (
    hp.uniform('a', 0, 1),
    hp.loguniform('b', 0, 1))

dict_space = {
```

```
    'a': hp.uniform('a', 0, 1),
    'b': hp.loguniform('b', 0, 1)}
```

There should be no functional difference between using list and tuple syntax to describe a sequence of elements in a configuration space, but both syntaxes are supported for everyone's convenience.

Creating list, tuple, and dictionary spaces as illustrated above is just one example of nesting. Each of these container types can be nested to form deeper configuration structures:

```
nested_space = [
    [ {'case': 1, 'a': hp.uniform('a', 0, 1)},
      {'case': 2, 'b': hp.loguniform('b', 0, 1)}],
    'extra literal string',
    hp.randint('r', 10) ]
```

There is no requirement that list elements have some kind of similarity, each element can be any valid configuration expression. Note that Python values (e.g. numbers, strings, and objects) can be embedded in the configuration space. These values will be treated as constants from the point of view of the optimization algorithms, but they will be included in the configuration argument objects passed to the objective function.

Sampling from a configuration space

The previous few code fragments have defined various configuration spaces. These spaces are not objective function arguments yet, they are simply a description of *how to sample* objective function arguments. You can use the routines in `hyperopt.pyll.stochastic` to sample values from these configuration spaces.

```
from hyperopt.pyll.stochastic import sample

print sample(list_space)
# => [0.13, .235]

print sample(nested_space)
# => [[{'case': 1, 'a', 0.12}, {'case': 2, 'b': 2.3}],
#     'extra_literal_string',
#     3]
```

Note that the labels of the random configuration variables have no bearing on the sampled values themselves, the labels are only used internally by the optimization algorithms. Later when we look at the `trials` parameter to `fmin` we will see that the labels are used for analyzing search results too. For now though, simply note that the labels are not for the objective function.

Deterministic expressions in configuration spaces

It is also possible to include deterministic expressions within the description of a configuration space. For example, we can write

```
from hyperopt.pyll import scope

def foo(x):
    return str(x) * 3

expr_space = {
    'a': 1 + hp.uniform('a', 0, 1),
    'b': scope.minimum(hp.loguniform('b', 0, 1), 10),
    'c': scope.call(foo, args=(hp.randint('c', 5),)),
    }
```

The `hyperopt.pyll` submodule implements an expression language that stores this logic in a symbolic representation. Significant processing can be carried out by these intermediate expressions. In fact, when you call `fmin(f, space)`, your arguments are quickly combined into a single objective-and-configuration evaluation graph of the form: `scope.call(f, space)`. Feel free to move computations between these intermediate functions and the final objective function as you see fit in your application.

You can add new functions to the `scope` object with the `define` decorator:

```
from hyperopt.pyll import scope

@scope.define
def foo(x):
    return str(x) * 3

# -- This will print "000"; foo is called as usual.
print foo(0)

expr_space = {
    'a': 1 + hp.uniform('a', 0, 1),
    'b': scope.minimum(hp.loguniform('b', 0, 1), 10),
    'c': scope.foo(hp.randint('cbase', 5)),
    }

# -- This will draw a sample by running foo(x)
#    on a random integer x.
print sample(expr_space)
```

Read through `hyperopt.pyll.base` and `hyperopt.pyll.stochastic` to see the functions that are available, and feel free to add your own. One important caveat is that functions used in configuration space descriptions must be serializable (with pickle module) in order to be compatible with parallel search (discussed below).

Defining conditional variables with `choice` and `pchoice`

Having introduced nested configuration spaces, it is worth coming back to the `hp.choice` and `hp.pchoice` hyperparameter types. An `hp.choice(label, options)` hyperparameter *chooses* one of the options that you provide, where the `options` must be a list. We can use `choice` to define an appropriate configuration space for the `w` objective function (introduced in Section Configuration Spaces).

```
w_space = hp.choice('case', [
    {'use_var': 'x', 'x': hp.normal('x', 0, 1)},
    {'use_var': 'y', 'y': hp.uniform('y', 1, 3)}])

print sample(w_space)
# ==> {'use_var': 'x', 'x': -0.89}

print sample(w_space)
# ==> {'use_var': 'y', 'y': 2.63}
```

Recall that in `w`, the `'y'` key of the configuration is not used when the `'use_var'` value is `'x'`. Similarly, the `'x'` key of the configuration is not used when the `'use_var'` value is `'y'`. The use of `choice` in the `w_space` search space reflects the conditional usage of keys `'x'` and `'y'` in the `w` function. We have used the `choice` variable to define a space that never has more variables than is necessary.

The choice variable here plays more than a cosmetic role; it can make optimization much more efficient. In terms of `w` and `w_space`, the choice node prevents `y` for being *blamed* (in terms of the logic of the search algorithm) for poor performance when `'use_var'` is `'x'`, or *credited* for good performance when `'use_var'` is `'x'`. The choice variable creates a special node in the expression graph that prevents the conditionally unnecessary part of the expression graph from being evaluated at all. During optimization, similar special-case logic prevents any association between the return value of the objective function and irrelevant hyperparameters (ones that were not chosen, and hence not involved in the creation of the configuration passed to the objective function).

The `hp.pchoice` hyperparameter constructor is similar to `choice` except that we can provide a list of probabilities corresponding to the options, so that random sampling chooses some of the options more often than others.

```
w_space_with_probs = hp.pchoice('case', [
    (0.8, {'use_var': 'x',
           'x': hp.normal('x', 0, 1)}),
    (0.2, {'use_var': 'y',
           'y': hp.uniform('y', 1, 3)})])
```

Using the `w_space_with_probs` configuration space expresses to `fmin` that we believe the first case (using `'x'`) is five times as likely to yield an optimal configuration that the second case. If your objective function only uses a subset of the configuration space on any given evaluation, then you should use `choice` or `pchoice` hyperparameter variables to communicate that pattern of inter-dependencies to `fmin`.

Sharing a configuration variable across choice branches

When using choice variables to divide a configuration space into many mutually exclusive possibilities, it can be natural to reuse some configuration variables across a few of those possible branches. Hyperopt's configuration space supports this in a natural way, by allowing the objects to appear in multiple places within a nested configuration expression. For example, if we wanted to add a `randint` choice to the returned dictionary that did not depend on the `'use_var'` value, we could do it like this:

```
c = hp.randint('c', 10)

w_space_c = hp.choice('case', [
    {'use_var': 'x',
     'x': hp.normal('x', 0, 1),
     'c': c},
    {'use_var': 'y',
     'y': hp.uniform('y', 1, 3),
     'c': c}])
```

Optimization algorithms in Hyperopt would see that `c` is used regardless of the outcome of the `choice` value, so they would correctly associate `c` with all evaluations of the objective function.

Configuration Example: `sklearn` classifiers

To see how we can use these mechanisms to describe a more realistic configuration space, let's look at how one might describe a set of classification algorithms in [sklearn].

```
from hyperopt import hp
from hyperopt.pyll import scope
from sklearn.naive_bayes import GaussianNB
from sklearn.svm import SVC
from sklearn.tree import DecisionTreeClassifier\
    as DTree

scope.define(GaussianNB)
scope.define(SVC)
scope.define(DTree, name='DTree')

C = hp.lognormal('svm_C', 0, 1)
space = hp.pchoice('estimator', [
    (0.1, scope.GaussianNB()),
    (0.2, scope.SVC(C=C, kernel='linear')),
    (0.3, scope.SVC(C=C, kernel='rbf',
        width=hp.lognormal('svm_rbf_width', 0, 1),
        )),
    (0.4, scope.DTree(
        criterion=hp.choice('dtree_criterion',
            ['gini', 'entropy']),
        max_depth=hp.choice('dtree_max_depth',
            [None, hp.qlognormal('dtree_max_depth_N',
                                 2, 2, 1)],
    ])
```

This example illustrates nesting, the use of custom expression types, the use of `pchoice` to indicate independence among configuration branches, several numeric hyperparameters, a discrete hyperparameter (the Dtree criterion), and a specification of our prior preference among the four possible classifiers. At the top level we have a `pchoice` between four sklearn algorithms: Naive Bayes (NB), a Support Vector Machine (SVM) using a linear kernel, an SVM using a Radial Basis Function (`'rbf'`) kernel, and a decision tree (Dtree). The result of evaluating the configuration space is actually a sklearn estimator corresponding to one of the three possible branches of the top-level choice. Note that the example uses the same C variable for both types of SVM kernel. This is a technique for injecting domain knowledge to assist with search; if each of the SVMs prefers roughly the same value of C then this will buy us some search efficiency, but it may hurt search efficiency if the two SVMs require very different values of C. Note also that the hyperparameters all have unique names; it is tempting to think they should be named automatically by their path to the root of the configuration space, but the configuration space is not a tree (consider the `C` above). These names are also invaluable in analyzing the results of search after `fmin` has been called, as we will see in the next section, on the `Trials` object.

The Trials Object

The `fmin` function returns the best result found during search, but can also be useful to analyze all of the trials evaluated during search. Pass a `trials` argument to `fmin` to retain access to all of the points accessed during search. In this case the call to `fmin` proceeds as before, but by passing in a trials object directly, we can inspect all of the return values that were calculated during the experiment.

```
from hyperopt import (hp, fmin, space_eval,
    Trials)
trials = Trials()
best = fmin(q, space, trials=trials)
print trials.trials
```

Information about all of the points evaluated during the search can be accessed via attributes of the `trials` object. The `.trials` attribute of a Trials object (`trials.trials` here) is a list with an element for every function evaluation made by `fmin`. Each element is a dictionary with at least keys:

- `'tid'`: value of type int
  trial identifier of the trial within the search
- `'results'`: value of type dict
  dict with `'loss'`, `'status'`, and other information returned by the objective function (see below for details)
- `'misc'` value of dict with keys `'idxs'` and `'vals'`
  compressed representation of hyperparameter values

This trials object can be pickled, analyzed with your own code, or passed to Hyperopt's plotting routines (described below).

Trial results: more than just the loss

Often when evaluating a long-running function, there is more to save after it has run than a single floating point loss value. For example there may be statistics of what happened during

the function evaluation, or it might be expedient to pre-compute results to have them ready if the trial in question turns out to be the best-performing one.

Hyperopt supports saving extra information alongside the trial loss. To use this mechanism, an objective function must return a dictionary instead of a float. The returned dictionary must have keys `'loss'` and `'status'`. The status should be either `STATUS_OK` or `STATUS_FAIL` depending on whether the loss was computed successfully or not. If the status is `STATUS_OK`, then the loss must be the objective function value for the trial. Writing a quadratic `f(x)` function in this dictionary-returning style, it might look like:

```
import time
from hyperopt import fmin, Trials
from hyperopt import STATUS_OK, STATUS_FAIL

def f(x):
    try:
        return {'loss': x ** 2,
                'time': time.time(),
                'status': STATUS_OK }
    except Exception, e:
        return {'status': STATUS_FAIL,
                'time': time.time(),
                'exception': str(e)}
trials = Trials()
fmin(f, space=hp.uniform('x', -10, 10),
    trials=trials)
print trials.trials[0]['results']
```

An objective function can use just about any keys to store auxiliary information, but there are a few special keys that are interpreted by Hyperopt routines:

`'loss_variance'`: type float
: variance in a stochastic objective function

`'true_loss'`: type float
: if you pre-compute a test error for a validation error loss, store it here so that Hyperopt plotting routines can find it.

`'true_loss_variance'`: type float
: variance in test error estimator

`'attachments'`: type dict
: short (string) keys with potentially long (string) values

The `'attachments'` mechanism is primarily useful for reducing data transfer times when using the `MongoTrials` trials object (discussed below) in the context of parallel function evaluation. In that case, any strings longer than a few megabytes actually *have* to be placed in the attachments because of limitations in certain versions of the mongodb database format. Another important consideration when using `MongoTrials` is that the entire dictionary returned from the objective function must be JSON-compatible. JSON allows for only strings, numbers, dictionaries, lists, tuples, and date-times.

HINT: To store NumPy arrays, serialize them to a string, and consider storing them as attachments.

Parallel Evaluation with a Cluster

Hyperopt has been designed to make use of a cluster of computers for faster search. Of course, parallel evaluation of trials sits at odds with *sequential* model-based optimization. Evaluating trials in parallel means that efficiency per function evaluation will suffer (to an extent that is difficult to assess a-priori), but the improvement in efficiency as a function of wall time can make the sacrifice worthwhile.

Hyperopt supports parallel search via a special trials type called `MongoTrials`. Setting up a parallel search is as simple as using `MongoTrials` instead of `Trials`:

```
from hyperopt import fmin
from hyperopt.mongo import MongoTrials
trials = MongoTrials('mongo://host:port/fmin_db/')
best = fmin(q, space, trials=trials)
```

When we construct a `MongoTrials` object, we must specify a running *mongod* database [mongodb] for inter-process communication between the `fmin` producer-process and *worker* processes, which act as the consumers in a producer-consumer processing model. If you simply type the code fragment above, you may find that it either crashes (if no mongod is found) or hangs (if no worker processes are connected to the same database). When used with `MongoTrials` the `fmin` call simply enqueues configurations and waits until they are evaluated. If no workers are running, `fmin` will block after enqueing one trial. To run `fmin` with `MongoTrials` requires that you:

1) Ensure that mongod is running on the specified host and port,
2) Choose a database name to use for a *particular fmin call*, and
3) Start one or more *hyperopt-mongo-worker* processes.

There is a generic *hyperopt-mongo-worker* script in Hyperopt's `scripts` subdirectory that can be run from a command line like this:

```
hyperopt-mongo-worker --mongo=host:port/db
```

To evaluate multiple trial points in parallel, simply start multiple scripts in this way that all work on the same database.

Note that mongodb databases persist until they are deleted, and `fmin` will never delete things from mongodb. If you call `fmin` using a particular database one day, stop the search, and start it again later, then `fmin` will continue where it left off.

The Ctrl Object for Realtime Communication with MongoDB

When running a search in parallel, you may wish to provide your objective function with a handle to the mongodb database used by the search. This mechanism makes it possible for objective functions to:

- update the database with partial results,
- to communicate with concurrent processes, and
- even to enqueue new configuration points.

This is an advanced usage of Hyperopt, but it is supported via syntax like the following:

```
from hyperopt import pyll

@hyperopt.fmin_pass_expr_memo_ctrl
def realtime_objective(expr, memo, ctrl):
    config = pyll.rec_eval(expr, memo=memo)
    # .. config is a configuration point
    # .. ctrl can be used to interact with database
    return {'loss': f(config),
            'status': STATUS_OK, ...}
```

The `fmin_pass_expr_memo_ctrl` decorator tells `fmin` to use a different calling convention for the objective function, in which internal objects `expr`, `memo` and `ctrl` are exposed to the objective function. The `expr` the configuration space, the `memo` is a dictionary mapping nodes in the configuration space description graph to values for those nodes (most importantly, values for the

hyperparameters). The recursive evaluation function `rec_eval` computes the configuration point from the values in the `memo` dictionary. The `config` object produced by `rec_eval` is what would normally have been passed as the argument to the objective function. The `ctrl` object is an instance of `hyperopt.Ctrl`, and it can be used to to communicate with the trials object being used by `fmin`. It is possible to use a `ctrl` object with a (sequential) `Trials` object, but it is most useful when used with `MongoTrials`.

To summarize, Hyperopt can be used both purely sequentially, as well as *broadly sequentially* with multiple current candidates under evaluation at a time. In the parallel case, mongodb is used for inter-process communication and doubles as a persistent storage mechanism for post-hoc analysis. Parallel search can be done with the same objective functions as the ones used for sequential search, but users wishing to take advantage of asynchronous evaluation in the parallel case can do so by using a lower-level calling convention for their objective function.

Ongoing and Future Work

Hyperopt is the subject of ongoing and planned future work in the algorithms that it provides, the domains that it covers, and the technology that it builds on.

Related Bayesian optimization software such as Frank Hutter et al's [SMAC], and Jasper Snoek's [Spearmint] implement state-of-the-art algorithms that are different from the TPE algorithm currently implemented in Hyperopt. Questions about which of these algorithms performs best in which circumstances, and over what search budgets remain topics of active research. One of the first technical milestones on the road to answering those research questions is to make each of those algorithms applicable to common search problems.

Hyperopt was developed to support research into deep learning [BBBK11] and computer vision [BYC13]. Corresponding projects [hp-dbn] and [hp-convnet] have been made public on Github to illustrate how Hyperopt can be used to define and optimize large-scale hyperparameter optimization problems. Currently, Hristijan Bogoevski is investigating Hyperopt as a tool for optimizing the suite of machine learning algorithms provided by sklearn; that work is slated to appear in the [hp-sklearn] project in the not-too-distant future.

With regards to implementation decisions in Hyperopt, several people have asked about the possibility of using IPython instead of mongodb to support parallelism. This would allow us to build on IPython's cluster management interface, and relax the constraint that objective function results be JSON-compatible. If anyone implements this functionality, a pull request to Hyperopt's master branch would be most welcome.

Summary and Further Reading

Hyperopt is a Python library for Sequential Model-Based Optimization (SMBO) that has been designed to meet the needs of machine learning researchers performing hyperparameter optimization. It provides a flexible and powerful language for describing search spaces, and supports scheduling asynchronous function evaluations for evaluation by multiple processes and computers. It is BSD-licensed and available for download from PyPI and Github. Further documentation is available at [http://jaberg.github.com/hyperopt].

Acknowledgements

Thanks to Nicolas Pinto for some influential design advice, Hristijan Bogoevski for ongoing work on an sklearn driver, and to many users who have contributed feedback. This project has been supported by the Rowland Institute of Harvard, the National Science Foundation (IIS 0963668), and the NSERC Banting Fellowship program.

References

[BB12] J. Bergstra and Y. Bengio. *Random Search for Hyperparameter Optimization* J. Machine Learning Research, 13:281--305, 2012.

[BBBK11] J. Bergstra, R. Bardenet, Y. Bengio and B. Kégl. *Algorithms for Hyper-parameter Optimization*. Proc. Neural Information Processing Systems 24 (NIPS2011), 2546–2554, 2011.

[BYC13] J. Bergstra, D. Yamins and D. D. Cox. *Making a Science of Model Search: Hyperparameter Optimization in Hundreds of Dimensions for Vision Architectures*. Proc. ICML, 2013.

[Brochu10] E. Brochu. *Interactive Bayesian Optimization: Learning Parameters for Graphics and Animation*, PhD thesis, University of British Columbia, 2010.

[Hyperopt] http://jaberg.github.com/hyperopt

[hp-dbn] https://github.com/jaberg/hyperopt-dbn

[hp-sklearn] https://github.com/jaberg/hyperopt-sklearn

[hp-convnet] https://github.com/jaberg/hyperopt-convnet

[Mockus78] J. Mockus, V. Tiesis, and A. Zilinskas. *The applicatoin of Bayesian methods for seeking the extremum*, Towards Global Optimization, Elsevier, 1978.

[mongodb] www.mongodb.org

[ROAR] http://www.cs.ubc.ca/labs/beta/Projects/SMAC/#software

[sklearn] http://scikit-learn.org

[SLA13] J. Snoek, H. Larochelle and R. P. Adams. *Practical Bayesian Optimization of Machine Learning Algorithms*, NIPS, 2012.

[Spearmint] http://www.cs.toronto.edu/~jasper/software.html

[SMAC] http://www.cs.ubc.ca/labs/beta/Projects/SMAC/#software