Implementations of Algorithms for Hyper-Parameter Optimization

James Bergstra
The Rowland Institute
Harvard University
bergstra@rowland.harvard.edu

Rémi Bardenet
Laboratoire de Recherche en Informatique
Université Paris-Sud
bardenet@lri.fr

Yoshua Bengio
Dépt. d'Informatique et Recherche Opérationelle
Université de Montréal
yoshua.bengio@umontreal.ca

Balázs Kégl
Linear Accelerator Laboratory
Université Paris-Sud, CNRS
balazs.kegl@gmail.com

Abstract

Several recent advances to the state of the art in image classification benchmarks have come from better configurations of existing techniques rather than novel approaches to feature learning. Traditionally, hyper-parameter optimization has been the job of humans because they can be very efficient in regimes where only a few trials are possible. Presently, computer clusters and GPU processors make it possible to run more trials and we show that algorithmic approaches can find better results. Recently, [1] showed that two sequential model-based optimization algorithms could outperform domain experts in the tuning of Deep Belief Networks (DBNs). This abstract serves as a companion to [1], introducing `hyperopt`, the software that was used to run those experiments. `hyperopt`is a reusable engine for hyper-parameter optimization, and a platform for research in distributed asychronous hyper-parameter optimization algorithms.

1 Introduction

Models such as deep belief networks (DBNs [2]), stacked denoising autoencoders [3], convolutional networks [4], as well as classifiers based on sophisticated feature extraction techniques such as mcRBMs [5] can have dozens of hyper-parameters depending on how many potential hyper-parameters the experimenter chooses to leave fixed to an *a priori* reasonable default. The difficulty of tuning these models makes published results difficult to reproduce, and turns the application of these methods into something more akin to an art than a science. At the NIPS 2011 conference, [1] introduced two algorithms for hyper-parameter optimization, and showed that they were effective at optimizing DBN architectures. This abstract serves as a companion to [1] and introduces `hyperopt`, a free open-source platform (BSD license) for asynchronous distributed hyper-parameter optimization.[1] The `hyperopt` package provides reusable implementations of the algorithms in [1], and an extensible platform for future work in hyper-parameter optimization algorithms.

[1] show that moderately parallelized implementations of sequential model-based optimization (SMBO) algorithms can be effective in high-dimensional spaces (up to 32 dimensions). They introduce two algorithms: GP, based on Gaussian process regression; GM, based on a graphical model derived from the search space. GM and GP are evaluated on the tasks of neural network and DBN hyper-parameter optimization. Some of the results from [1] are summarized in Figure 1 and Table 1. The GM and GP algorithms both outperform manual and random search on the datasets studied. On the tasks of convex shape classification and digit recognition in the MNIST rotated background

[1] `hyperopt` web page: http://www.github.com/jaberg/hyperopt

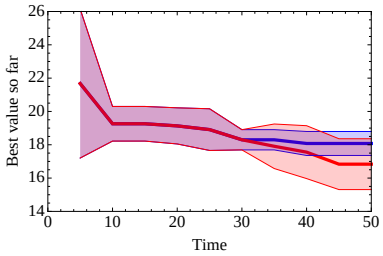

Figure 1: GP optimizing neural network hyper-parameters on the Boston Housing regression task. Shown: best minimum found so far every 5 iterations, against time. Red = GP, Blue = Random. Shaded areas = one-sigma error bars.

	convex	**MRBI**
GM	**14.13** ± 0.30 %	**44.55** ± 0.44%
GP	16.70 ± 0.32%	47.08 ± 0.44%
Manual	18.63 ± 0.34%	47.39 ± 0.44%
Random	18.97 ± 0.34 %	46.22 ± 0.44%

Table 1: The test set classification error of the best model found by each search algorithm on each problem. Each search algorithm was allowed up to 200 trials. The manual searches used 82 trials for **convex** and 27 trials "MNIST rotated background images" (**MRBI**).

images dataset (both introduced in [6]) the DBN model optimized by the GM algorithm achieves the best scores to our knowlege.

The GP and GM algorithms are quite general, not DBN-specific. They apply to a broad range of configuration problems involving fixed or variable numbers of parameters that each might be discrete, ordinal, or continuous. This paper describes how the `hyperopt` software package allows them to be used in new optimization problems, and how to try new optimization strategies on problems such as DBN configuation..

2 Sequential Model-based Global Optimization

The GM and GP algorithms presented in [1] fit into the framework of Sequential Model-Based Global Optimization (SMBO). SMBO algorithms have been used in many applications where evaluation of the fitness function is expensive [7, 8]. In an application where the true fitness function $f : \mathcal{X} \to \mathbb{R}$ is costly to evaluate, model-based algorithms approximate f with a surrogate that is cheaper to evaluate. Typically the inner loop in an SMBO algorithm is the numerical optimization of this surrogate, or some transformation of the surrogate. The point x^* that maximizes the surrogate (or its transformation) becomes the proposal for where the true function f should be evaluated. This active-learning-like algorithm template is summarized in Figure 2.

SMBO(f, M_0, T, S)

1 $\quad \mathcal{H} \leftarrow \emptyset$,
2 $\quad$ For $t \leftarrow 1$ **to** T,
3 $\quad\quad$ Generate candidate x^* by optimizing criterion S on model M_{t-1},
4 $\quad\quad$ Evaluate $f(x^*)$, $\quad \triangleright$ *Expensive step*
5 $\quad\quad \mathcal{H} \leftarrow \mathcal{H} \cup (x^*, f(x^*))$,
6 $\quad\quad$ Fit a new model M_t to $\mathcal{H}$.
7 $\quad$ **return** $\mathcal{H}$

Figure 2: The pseudo-code of a generic Sequential Model-Based Optimization algorithm.

SMBO algorithms are differentiated in how they approximate f, and in the criterion that they optimize to get x^*. [1] looked at two SMBO algorithms for optimizing hyper-parameters: one based on Gaussian Process regression (GP) for modeling $P(y = f(x)|x)$ and one based on a simple non-parametric graphical model (GM) of the joint distribution $P(x, y = f(x))$. Both algorithms optimized the criterion of expected improvement (EI) in some surrogate to identify the proposal x^* in the SMBO template.

2.1 Parallelizing Sequential Optimization

SMBO algorithms are fundamentally *sequential* rather than parallel. However, when there is enough data to warrant a complex algorithm (requiring automatic configuration) it also typically takes a lot of CPU time (tens or hundreds of minutes) to use all of that data in testing that algorithm.

Researchers in machine learning are accustomed to using a compute cluster to optimize hyper-parameters because they can parallelize those calculations. It is of little practical use to use a sequential algorithm that cannot outperform a cluster in terms of quality of solution as a function of wall time. We need the optimization algorithm to parellelize the evaluations on line 4 of Figure 2.

`hyperopt` parallelizes sequential optimization by running the evaluations of line 4 asynchronously, and updating the database of results ($\mathcal{H}$) asynchronously as those evaluation jobs complete. While the evaluation of the t^{th} proposal is in progress, it is marked temporarily as having a constant, disappointing score (the *constant liar* approach, [9]). A disapponting score prevents subsequent proposals $t + \tau$ from exploring nearby locations redundantly.

2.2 Major Implementation Components

Our implementation of asynchronous SMBO employs a client-server architecture. The server is a standard MongoDB. MongoDB is used both to track the state of the experiment $\mathcal{H}$ and as a message board for more general inter-process communication between the clients. The clients are Python programs implemented in the `hyperopt` package. Two scripts are used to conduct a distributed asynchronous optimization: `mongo-worker` and `mongo-search`.

`mongo-worker` - worker loop
: Called from the command-line with options to connect to a particular mongo server, it polls the server for a new candidate h in $\mathcal{H}$ and evaluates it.

`mongo-search` - server loop
: Called from the command-line with options to connect to a particular mongo server, it polls the server and inserts a new candidate into $\mathcal{H}$ whenever the existing ones have all been taken by worker clients.

The `mongo-worker` script polls the mongo server and atomically fetches an un-started candidate h from $\mathcal{H}$, which is a JSON (JavaScript Object Notation) document. It uses a field in h to identify the python function f that is to be used to evaluate the candidate. It extracts the candidate configuration subdocument (C). It constructs a control object via which the evaluation function can communicate with mongo and other processes (R). Finally it executes a sub-process that calls $f(C, R)$. When f returns, the subprocess marks h as being complete (or failed with an error). The `mongo-worker` script is meant to be launched using existing cluster dispatching software such as Torque, PBS, or SGE. A single experiment can draw on `mongo-worker` scripts running on multiple networks and clusters, the bandwidth requirement for communication between `mongo-worker` and `mongo` can be very low.

The `mongo-search` script requires command-line arguments specifying (a) the optimization algorithm and (b) the evaluation function. The script polls the mongo server and when there are zero new candidates in $\mathcal{H}$ it appeals to a plug-in (indicated by the command-line arguments) to suggest a new candidate. The plug-in makes a suggestion based on the state, the configuration, and the evaluation result of each $h \in \mathcal{H}$. The state of each h is either "new", "running", "finished successfully", or "finished in error". The `mongo-search` script inserts that candidate into $\mathcal{H}$, where it will be picked up by an idle `mongo-worker`. The `mongo-worker` and `mongo-search` script are independent clients from the operating system's perspective: either one can be killed and restarted without disrupting the other. `mongo-search`saves its state to the MongoDB if it is killed with CTRL-C, so that it can be resumed later.

2.3 Using hyperopt to optimize a new function

To optimize a new function with `hyperopt`, implement the evaluation function f as a `Bandit` subclass. In `hyperopt`, a Bandit carries two pieces of information:

- a callable function (corresponding to f in Figure 2)
- a prior over the configuration argument to f (see Figure 3).

The callable function is simply a class method or static method of your bandit subclass. The prior over configuration arguments can be specified with the stochastic expression language defined within

```
rdict(
    "preprocessing", one_of(
        rdict(
            "kind", "raw"),
        rdict(
            "kind", "zca",
            "energy", uniform(low=0.5, high=1.0))),
    "dataset_name", "MNIST",
    "seed", one_of(5, 6, 7, 8),
    "batchsize", one_of(1, 20, 100),
    "lr", lognormal(mu=log(.01), sigma=3),
    "lr_anneal_start", ceil_lognormal(mu=log(1000), sigma=2),
    "l2_penalty", one_of(0, lognormal(mu=log(1.0e-6), sigma=3)),
    "n_hid", ceil_lognormal(mu=log(512), sigma=3, round=16))
```

Figure 3: Specification of a search space using random dictionaries (`rdict`), random choices (`one_of`), and random numbers (`uniform`, `lognormal`, `ceil_lognormal`). Optimization algorithms in `hyperopt` inspect this data structure to draw random samples, infer posteriors, and define kernel functions in the configuration space.

`hyperopt` (see Figure 3 for an example). This language is makes it easy to express broad, simple priors. it is possible to write a more complicated prior distributions over the configuration space by writing a stochastic Theano program.[2] Both of these methods provide a graphical description of the configuration space to an optimization algorithm.

The structure of the configuration description can have an impact on search algorithms. For example, Figure 3 uses a nested random variable to define the energy of the ZCA preprocessing option. We could express exactly the same prior if preprocessing were defined as `one_of("raw", "zca")` and `"energy"` were promoted to a top-level configuration field. However, these two expressions of the prior would lead to different kernels in the GP optimization algorithm. In the former case energy would not enter into the distance between trials with "raw" preprocessing. In the latter case, it would. It is our hope that a single description of a configuration space suffices to get the best performance from all optimization algorithms.

2.4 Optimization Algorithms

`hyperopt` provides three optimization algorithms: random search, the GM algorithm from [1], and the GP algorithm from [1]. More algorithms are planned by the authors, and contributions from other researchers are welcome.

Optimization algorithms should inherit from the `BanditAlgo` (or `TheanoBanditAlgo`) base classes. Those classes have a virtual method `suggest` (or `theano_suggest` that derived classes must implement. The details of how those functions should be implemented are provided in the code and documentation online. Essentially those functions accept the current state of $\mathcal{H}$ as an argument and return some number of promising candidates x^* for `mongo-search` to insert in the database.

2.5 Other notable resources

`hyperopt` also includes:

- `mongo-show` - a script for rapid visualization of the state of an experiment. It can display a scatterplot of scores against time, or against the various hyper-parameters of the configuration space. It serves additionally as a starting point for custom visualization code.
- `search` - a script for serial-mode evaluation that works without MongoDB.
- Simple inexpensive bandits for testing optimization algorithms.

[2] Stochastic Theano programs implemented with MonteTheano: http://www.github.com/jaberg/MonteTheano

3 Conclusion

Bayesian optimization and sequential model-based optimization represent promising approaches to hyper-parameter optimization. This paper has introduced `hyperopt`, an open source package for distributed hyper-parameter optimization. It implements the algorithms of [1], and provides extensible platform for future work. For more information, please follow the development of `hyperopt` online at `http://www.github.com/jaberg/hyperopt`.

References

[1] J. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl. Algorithms for hyper-parameter optimization. In *NIPS 24*. 2011.

[2] G. E. Hinton, S. Osindero, and Y. Teh. A fast learning algorithm for deep belief nets. *Neural Computation*, 18:1527–1554, 2006.

[3] P. Vincent, H. Larochelle, I. Lajoie, Y. Bengio, and P. A. Manzagol. Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion. *Machine Learning Research*, 11:3371–3408, 2010.

[4] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, November 1998.

[5] M. Ranzato and G. E. Hinton. Modeling pixel means and covariance using factorized third-order Boltzmann machines. In *CVPR 2010*. IEEE Press, 2010.

[6] H. Larochelle, D. Erhan, A. Courville, J. Bergstra, and Y. Bengio. An empirical evaluation of deep architectures on problems with many factors of variation. In *ICML 2007*, pages 473–480, 2007.

[7] F. Hutter. *Automated Configuration of Algorithms for Solving Hard Computational Problems*. PhD thesis, University of British Columbia, 2009.

[8] F. Hutter, H. Hoos, and K. Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *LION-5*, 2011. Extended version as UBC Tech report TR-2010-10.

[9] D. Ginsbourger, R. Le Riche, and L. Carraro. Kriging is well-suited to parallelize optimization. In Y. Tenne, C. Goh, L. Hiot, and Y. Ong, editors, *Computational Intelligence in Expensive Optimization Problems*, volume 2 of *Adaptation, Learning, and Optimization*, pages 131–162. Springer Berlin Heidelberg.