Technische Universität München

apsis

Automatic Optimization of Hyper Parameters for Machine Learning using Bayesian Optimization

Author	*Matrikel Nr.*
Frederik Diehl	03647595
Andreas Jauch	03647597

March 10, 2015

1 Introduction

Machine learning and the algorithms used for it have become more and more complex in the past years. Especially the growth of Deep Learning architectures has resulted in a large number of hyperparameters - such as the number of hidden layers or the transfer function in a neural network - which have to be tuned to achieve the best possible performance.
Since the result of a hyperparameter choice can only be evaluated by completely training the model, every single hyperparameter evaluation requires potentially huge compute costs. Depending on the task, such an evaluation can take from hours to weeks, for potentially hundreds of evaluations.
There exist several methods to select the hyperparameters. The one most commonly used is grid search, which evaluates all combinations of chosen values for each parameter. Clearly, this scales exponentially in the parameter dimensions. Additionally, this risks evaluating parameter combinations where the only change is varying an insignificant parameter multiple times.
In comparison, a continuous evaluation of randomly chosen parameter values as presented in [**?**] avoids the latter problem, and has the advantage of being easily scalable and easy to parallelize.
In general, most projects use grid search or random search in combination with a human expert who determines small areas of the hyperparameter space to be automatically evaluated. This method obviously depends on expert knowledge and is therefore very difficult to scale and not transferrable.
Bayesian Optimization, which constructs a surrogate function using Gaussian Processes, aims to rectify this and has been shown to deliver good results on the hyperparameter optimization problem, for example in [**?**].

The ***apsis*** **toolkit** presented in this paper provides a flexible framework for hyperparameter optimization and includes both random search and a bayesian optimizer. It is implemented in Python and its architecture features adaptability to any desired machine learning code. It can easily be used with common Python ML frameworks such as *scikit-learn* [**?**]. Published under the MIT License other researchers are heavily encouraged to check out the code, contribute or raise any suggestions.

In chapter 2 the concept of Bayesian Optimization is briefly introduced and some theoretical results *apsis* is based on are presented. The next chapter covers *apsis*' flexible architecture followed by a chapter on performance evaluation. Finally the conclusion lists possible further steps to the take the project to the next level.
We want to thank our supervisors Prof. Dr. Daniel Cremers and Dipl. Inf. Justin Bayer for their outstanding support and helpful contributions.

2 Bayesian Optimization for Optimizing Hyperparameters

The general objective in hyperparameter optimization is to minimize a loss function L or other performance measure of a machine learning algorithm A with respect to the hyperparameter vector λ on withheld data.

$$\hat{\lambda} = \operatorname*{argmin}_{\lambda \epsilon \Lambda} \left(\underbrace{L(X_{\text{test}}, A_{\lambda}(X_{\text{train}}))}_{\Psi(\lambda)} \right)$$

In the following the true objective function will be called Ψ and the space of hyperparameters will be called Λ.

Since every evaluation of Ψ with respect to a certain λ incurs high cost, one goal should be to minimize their number. In Bayesian Optimization, a surrogate model is built to approximate the objective function Ψ. Nearly all of the Bayesian Optimization literature uses Gaussian Processes as a surrogate function. This is constructed from already evaluated samples.
The next candidate hyperparameter values to be evaluated is obtained from the surrogate model using a utility or acquisition function u.
Figure 1 illustrates these concepts. Following this procedure the problem of hyperparameter optimization now turns into the problem of maximizing the acquisition function u.

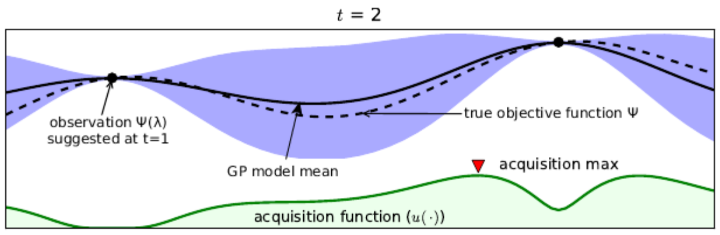


Fig. 1. Concept of sequential model-based optimization using a Gaussian Process model and an Expected Improvement acquisition function, here shown for a 1D toy maximization problem. Picture from [**?**], notation adapted.

In total, the Bayesian Optimization procedure is shown in algorithm 1, and is also called sequential model based optimization algorithm (SMBO) in the literature. In this algorithm, a model M_t is used to approximate the complex response function $\Psi(\lambda)$. The history of points where Ψ has been evaluated is stored in H. We run this algorithm for a fixed number of T step. In each of the steps we first search the extremum of the acquisition function u on the model M_{t-1}. Afterwards, we execute the expensive evaluation of Ψ at the found extremum λ^* and add this evaluation to our history. Finally we use the updated history information to update M_t. Figure 2 exemplarily shows this iterative approach for a one dimensional objective function.

Algorithm 1 Bayesian Optimization Algorithm [?]. Notation adapted.

```
1: function BAYOPT($\Psi$, $M_0$, $T$, $u$)
2:     $H \leftarrow \emptyset$;
3:     for all $t \in \{1..T\}$ do
4:         $\lambda^* \leftarrow argmin_\lambda\big(u(\lambda|M_{t-1})\big)$;
5:         Evaluate $\Psi(\lambda^*)$;
6:         $H \leftarrow H \cup (\lambda^*, \Psi(\lambda^*))$;
7:         $M_t \leftarrow$ refit surrogate model $M_{t-1}$ to updated $H$;
8:     end for
9:     return $H$;
10: end function
```

2.1 Acquisition Functions

One of the most important design decisions is the acquisition function used to interpret the model. Most related papers introduce the Probability of Improvement function as a starting point and historical reference, but in general, Expected Improvement works significantly better in practice. Having implemented both of these functions, *apsis* uses Expected Improvement as a default.

Probability of Improvement was first shown by **?** [?] and states the simple idea of maximizing the probability to achieve any improvement by choosing λ as a next point to sample from Ψ.

$$u_{\mathrm{PI}}(\lambda|M_t) = \Phi\left(\frac{\mu_{M_t}(\lambda) - f(\lambda^*)}{\sigma_{M_t}(\lambda;\theta)}\right) \tag{1}$$

That approach already indicates that it will favour exploitation of well-known areas over exploration of less-known regions of your optimization space Λ.

Expected Improvement A better criterion is the expected value of the actual improvement achieved when choosing to evaluate the objective for the next λ value.

$$u_{\mathrm{EI}}(\lambda|M_t) = \int_{-\infty}^{\infty} \underbrace{max(y^* - y, 0)}_{\text{value of improvement}} \cdot \underbrace{p_M(y|\lambda)}_{\text{probability of improvement}} \, dy \tag{2}$$

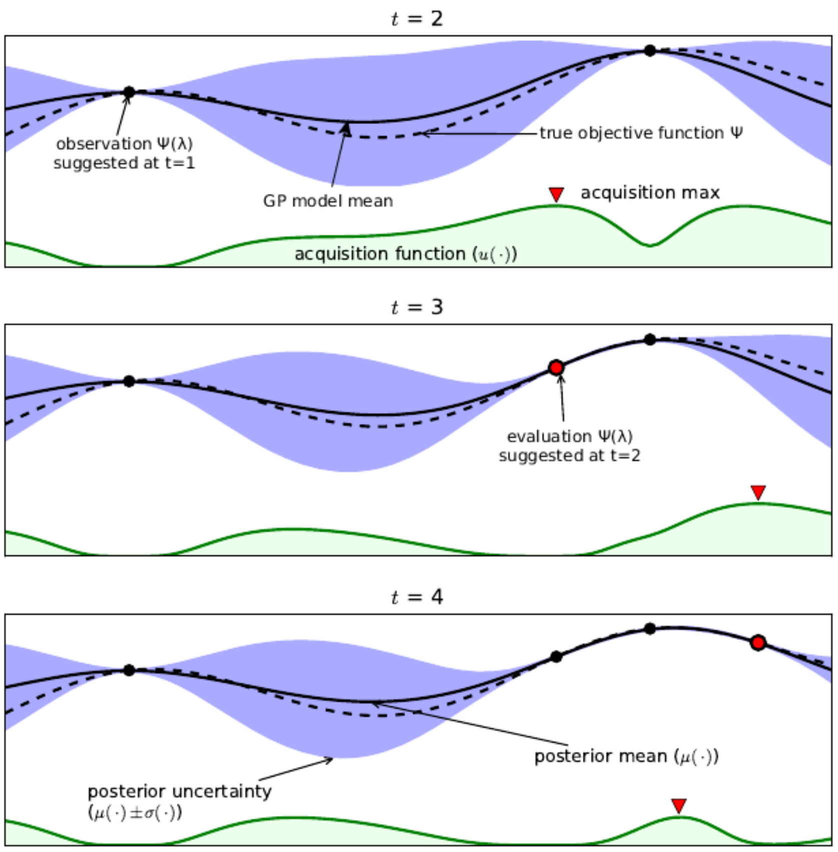


Fig. 2. Maximization of a one dimensional function with Bayesian Optimization with a GP and Expected Improvement acquisition function. At $t = 2$ only two observations are available and the uncertainty (in blue) between these two points is very high. Now the acquisition function balances the trade-off between exploration and exploitation having its maximum to be at the point λ_{t2}. $\Psi(\lambda_{t2})$ is evaluated and the GP posterior is updated at t=3. Picture from [**?**], notation in the picture adapted.

Since this integral cannot be directly computed, an analytical solution is helpful. For Gaussian Processes it has been shown [**? ?**] that the analytical solution of basic EI is obtained as

$$u_{\mathrm{EI}}(\lambda|M_t) = \sigma(\lambda) \cdot (z(\lambda) \cdot \Phi(\lambda) + \phi(\lambda)) \tag{3}$$

using $z(\lambda)$ defined as

$$z(\lambda) = \frac{f(\lambda^*) - \mu(\lambda)}{\sigma(\lambda)}$$

where $\mu(\lambda)$ and $\sigma(\lambda)$ are the mean and standard deviation as given from the Gaussian Process used. $\phi(\lambda)$ and $\Phi(\lambda)$ mark the standard normal distribution density and cumulative distribution function. Since *apsis* is supposed to handle both problems of maximization or minimization of the objective function both scenarios had to be incorporated into the acquisition function. Additionally, a trade-off parameter ζ to control the exploitation/exploration behaviour has been incorporated into u as suggested in [**?**]. This leads to the following modified version of $z(\lambda)$ that is used in *apsis*, where MAX is an integer being 1 when the objective function has to be maximized and 0 otherwise.

$$z(\lambda) = \frac{(-1)^{MAX} \cdot (f(\lambda^*) - \mu(\lambda) + \zeta)}{\sigma(\lambda)}$$

2.2 Expected Improvement Optimization

One possibility for optimizing the acquisition function is to use gradient based optimization methods as they usually feature a better convergence speed than methods not relying on first order derivative information.

EI Gradient Derivation Hence the gradient of EI had to be analytically derived. Applying the product rule to equation (3) one obtains

$$\nabla EI(\lambda) = \nabla\sigma(\lambda) \cdot \big(z(\lambda) \cdot \Phi(\lambda) + \phi(\lambda)\big) + \sigma(\lambda) \cdot \nabla\big(z(\lambda) \cdot \Phi(\lambda) + \phi(\lambda)\big) \tag{4}$$

$$\begin{aligned}\nabla EI(\lambda) = {} & \frac{\nabla\sigma(\lambda) \cdot EI(\lambda)}{\sigma(\lambda)} \\ & + \sigma(\lambda) \cdot \big(\nabla z(\lambda)\Phi(z) + \underbrace{z(\lambda)\phi(z) \cdot \nabla z(\lambda) - z(\lambda)\phi(z) \cdot \nabla z(\lambda)}_{=0}\big)\end{aligned} \tag{5}$$

In the above $\nabla\phi(x) = -x \cdot \phi(x)$ has been used and the last two terms of the sum cancel out leading to the compact result

$$\nabla EI(\lambda) = \frac{\nabla\sigma(\lambda) \cdot EI(\lambda)}{\sigma(\lambda)} + \sigma(\lambda) \cdot \big(\nabla z(\lambda)\Phi(z)\big). \tag{6}$$

Finally, we compute the derivative of $z(\lambda)$ using the product rule.

$$\nabla z(\lambda) = \frac{(-1)^{MAX} \cdot -\nabla\mu(\lambda)}{\sigma(\lambda)} - \frac{(-1)^{MAX}}{\sigma^2(\lambda)} \cdot (f(\lambda^*) - \mu(\lambda) + \zeta) \cdot \nabla\sigma(\lambda) \quad (7)$$

$$= \frac{(-1)^{MAX} \cdot -\nabla\mu(\lambda)}{\sigma(\lambda)} - \frac{z(\lambda) \cdot \nabla\sigma(\lambda)}{\sigma(\lambda)} \quad (8)$$

It is more convenient for the implementation to have $\nabla\sigma^2(\lambda)$ instead of $\nabla\sigma(\lambda)$, since the GP framework used in *apsis* returns both the mean and variance gradients. Fortunately, using the chain rule, one can derive

$$\nabla\sigma(\lambda) = \frac{\nabla\sigma^2(\lambda)}{2\sigma(\lambda)} \quad (9)$$

Using all of the above and inserting it in (6), the full EI gradient result is

$$\nabla EI(\lambda) = \frac{\nabla\sigma^2(\lambda)}{2\sigma(\lambda)} - (-1)^{MAX} \cdot \nabla\mu(\lambda) \cdot \Phi(z(\lambda)) - \nabla\sigma^2(\lambda) \cdot \Phi(z(\lambda)) \cdot \frac{z(\lambda)}{2\sigma(\lambda)}$$

which is the formula implemented in *apsis*.

Available Optimization Methods The following gradient and non-gradient based optimization methods are available in *apsis*.

- Quasi-Newton optimization using inverse BFGS [**?** , p. 72]
- Limited Memory BFGS with bounds algorithm (default) [**?**]
- Nelder-Mead method [**?**]
- Powell method [**?**]
- Conjugate Gradient method [**?**]
- inexact/truncated Newton method using Conjugate Gradient to approximately solve the Newton Equation [**?** , p. 62] [**?**]
- random search

Except for random search the implementations of these methods are taken from the SciPy project [**?**]. In order to provide these methods with a promising starting point λ_0 a random search maximization is always executed first and the best resulting λ will be used as λ_0 for any of the further optimization methods.

The BFGS method marks the strongest of these methods and offers the best convergence speed since it uses gradient information and approximates the second order information. Similar to the original Newton method it computes the next optimization step s_k by solving the Newton equation

$$\nabla^2 f(\lambda_k) \cdot s_k = -\nabla f(\lambda_k) \quad (10)$$

but as a so called Quasi Newton method it tries to approximate the Hessian ∇^2. Therefore it maintains the Quasi Newton equation (11) satisfied in every step to make sure H provides a well enough approximation for the Hessian.

$$H_{k+1} \cdot (\lambda^{k+1} - \lambda^k) = \nabla f(\lambda^{k+1}) - \nabla f(\lambda^k) \quad (11)$$

Since equation (10) can be solved for s_k by inverting the Hessian, the inverse BFGS method used in *apsis* directly approximates the inverse of the Hessian using an iterative algorithm to speed up optimization.
It will however not always converge since the algorithm involves a division by something that can become 0 [**?** , p. 72]. In that case the algorithm stops gracefully with an error and *apsis* relies on the random search result. Since the optimization space is bounded in real settings *apsis* by default uses an adopted version of BFGS, the L-BFGS-B [**?**] algorithm. In contrast to ordinary BFGS it does not store the full $n \times n$ Hessian but only stores a few vectors to implicitly represent the Hessian approximation H. Furthermore it has been extended by **?** [**?**] to respect simple bounds constraints for each variable definition which is sufficient for the bounds used in *apsis*.

3 Architecture

3.1 General Architectural Overview

The architecture of *apsis* is designed to be interoperable with any Python machine learning framework or self-implemented algorithm. *apsis* features an abstraction layer for the underlying optimization framework as depicted in figure 3. Every optimizer adheres to the abstract base class `Optimizer`. The optimizer uses three important model classes to control optimization. `Candidate` is used to represent a specific hyperparameter vector λ. `Experiments` are used to define a hyperparameter optimization experiment. An `Experiment` holds a list of `ParamDef` objects that define the nature of each specific hyperparameter. The external program interacts with *apsis* through a set of *Assistants*. They help to initialize *apsis*' internal structure and models and provide convenient access to optimization results, and can optionally store and plot them or compare several experiments. Additionally, the external program has to provide some integration code between the external program's machine learning algorithm and the *apsis* interface. Since the external program could be multi-threaded or clustered *apsis* refers to the processes running the machine learning algorithms as *Workers*.
On the implementation level *apsis* is made up by six packages reflecting this architecture. Table 1 lists all of them and briefly describes their purpose.

3.2 Model Objects

Candidate represents a specific hyperparameter vector λ and optionally stores the result of the objective function $\Psi(\lambda)$ achieved under this λ if already evaluated. Additionally it can store the cost occurred for evaluation and any other meta information used by the worker, e.g. the worker could keep track of the classifier's weights in each step and store them there. The actual vector λ is stored as a dictionary such that every parameter dimension $\lambda^{(i)}$ is named.

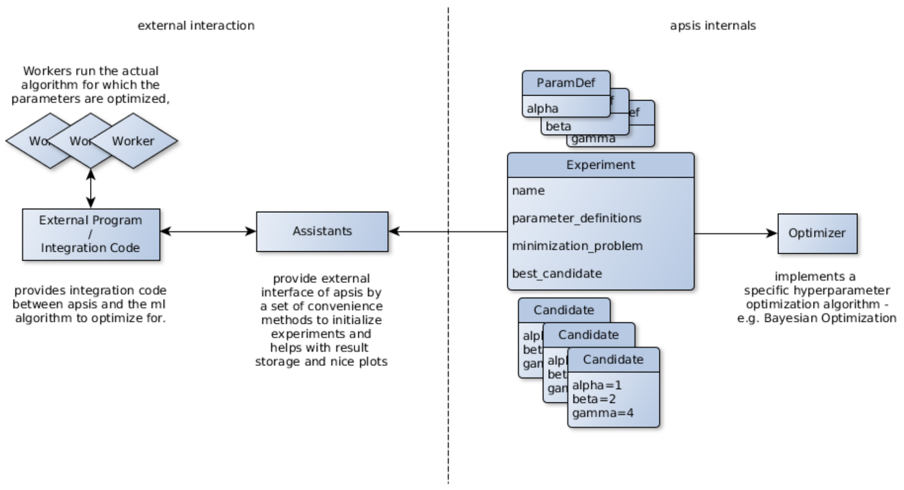

external interaction
apsis internals
Workers run the actual algorithm for which the parameters are optimized,
Worker
External Program / Integration Code
provides integration code between apsis and the ml algorithm to optimize for.
Assistants
provide external interface of apsis by a set of convenience methods to initialize experiments and helps with result storage and nice plots
ParamDef
alpha
beta
gamma
Experiment
name
parameter_definitions
minimization_problem
best_candidate
Candidate
alpha=1
beta=2
gamma=4
Optimizer
implements a specific hyperparameter optimization algorithm - e.g. Bayesian Optimization

Fig. 3. *apsis* general architectural overview

Experiment stores information about the nature of the parameters to be optimized, defines if the problem is for minimization and maximization. It keeps track of successfully evaluated, currently evaluated and to be evaluated `Candidate`s. It stores the best `Candidate` and optionally names the experiment.
Additionally, it provides methods for semantic check if candidates are valid for this experiment, and can convert itself to a CSV format for result storage.

ParamDef is the most general superclass used to define the nature of one hyperparameter dimension. It makes no assumption on the nature of the stored parameter. `NominalParamDef` defines non-comparable, unordered parameters and stores all available values in a list. `OrdinalParamDef` represents ordinal parameters where the order is maintained by the position in the list of values. As ordinal parameters can be compared they adhere to the `ComparableParamDef` interface and provide a `compare_values` function that provides similar semantic to the Python integrated `__cmp__` function. Furthermore, *apsis* comes with special support for numeric parameters represented by the `NumericParamDef` class. Numeric param defs are comparable and are represented internally by a warping into the $[0, 1]$ domain. Hence they need to be given an inwards and outwards warping function. The warped parameters are now compared and treated according to the rules of treating ordinary floats between 0 and 1. To ease initialization of numeric parameters `MinMaxNumericParamDef` automatically provides a warping assuming an equal distribution of the parameter between the given minimal and maximal value. `AsymptoticNumericParamDef` provides a parameter definition

Package	Description	Contents
assistants	Contains lab and experiment assistants for *apsis*' external interface.	experiment_assistant.py lab_assistant.py
optimizers	Contains all available optimizers and the abstract base class. Contains a subpackage for Bayesian Optimization.	optimizers.py random_search.py bayesian_optimization.py bayesian/acquisition_functions.py
models	Contains the model classes.	candidate.py experiment.py parameter_definition.py
tests	Contains unit tests for complete *apsis* code.	test_assistants/... test_models/... test_optimizers/... test_utilities/...
utilities	Utility functions used accross all packages.	benchmark_functions.py file_utils.py import_utils.py logging_utils.py optimizer_utils.py plot_utils.py randomization.py
demos	Some demos intended for learning how to use *apsis* by examples.	demo_MNIST.py demo_MNIST_MCMC_py

Table 1. list of *apsis* packages with their purpose and contents

for a parameter which is expected to be close to one value. For example, learning rates can be estimated as being close to 0. This leads to significantly better results during optimization, as long as some expert knowledge is available. Figure 4 depicts the structure of available parameter types and there inheritance relationships.

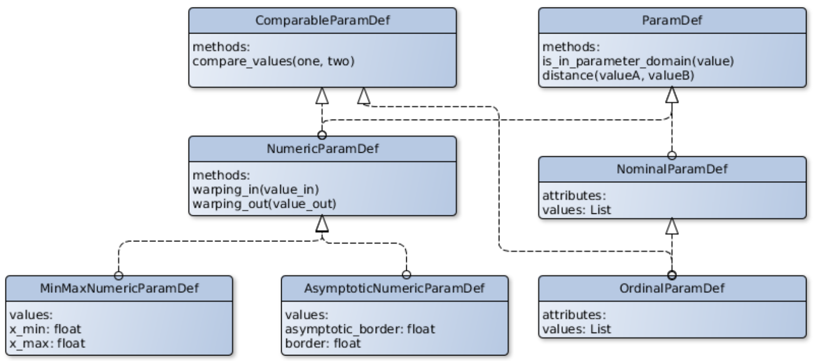


Fig. 4. Overview of ParamDef model classes and their relations.

3.3 Optimization Cores

The abstract base class `Optimizer` defines the common interface of any optimization algorithm provided in *apsis*. Central is the method `get_next_candidates`. Based upon information stored in the given experiment instance the method provides its user with one or several promising candidates to evaluate next. On construction the optimizer can receive a dictionary of optimizer specific parameters to define optimization related hyperparameters. Note that these parameters are not the ones that are subject of optimization but are parameters to govern the optimization behaviour such as which acquisition function or kernel is used in Bayesian Optimization. *apsis* provides two different optimization cores: a simple random search based optimizer called `RandomSearch` and the `SimpleBayesianOptimizer`.

RandomSearch Implements a very simple random search optimizer. For parameters of type `NumericParamDef` a uniform random varibale between 0 and 1 is generated to select a value in warped space for each parameter. For parameters of type `NominalParamDef` a value is drawn uniformly at random from the list of allowed values. All random numbers are generated using the numpy [**?**] random package.

SimpleBayesianOptimizer The `SimpleBayesianOptimizer` works according to the theory described in chapter 2. It is called simple since it currently works with one concurrent worker at a time only (though a multi worker variant is planned). It uses Gaussian Processes and their kernels provided by the GPy framework [**?**]. For the acquisition functions implemented and the methods in use for their optimization it shall be referred to chapter 2.1 and 2.2.

3.4 Experiment Assistants

The `BasicExperimentAssistant` provides the simplest interface between *apsis* and the outside world. It administers at most one experiment at a time. An experiment needs to be initialized by at least specifying a name to identify it, the `Optimizer` to be used and a list containing one `ParamDef` object for each hyperparameter to be optimized. It holds and manages the `Optimizer` and `Experiment` instances and provides an abstraction layer to their interface.

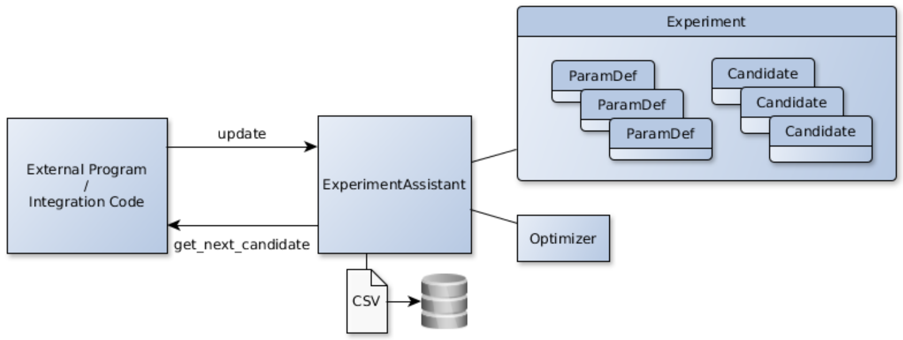


Fig. 5. The role of experiment assistants in *apsis*.

The experiment assistants provide the two methods `get_next_candidate` and `update` that should be called by the external program when it is ready to evaluate a new candidate or wants to notify *apsis* about work on a `Candidate`. The `Candidate` doesn't necessarily need to be one which was provided by *apsis* but can be any Candidate which adheres to the parameter space. Furthermore `BasicExperimentAssistant` cares for storing the results to CSV files when running to be sure to have all information available after termination or abortion of an experiment run. The behaviour of the CSV writing can be controlled upon initialization of the experiment assistant. As an extension *apsis* provides `PrettyExperimentAssistant` that can additionally create nice plots on the experiment. Chapter 4 shows some of these plots.

3.5 Lab Assistants

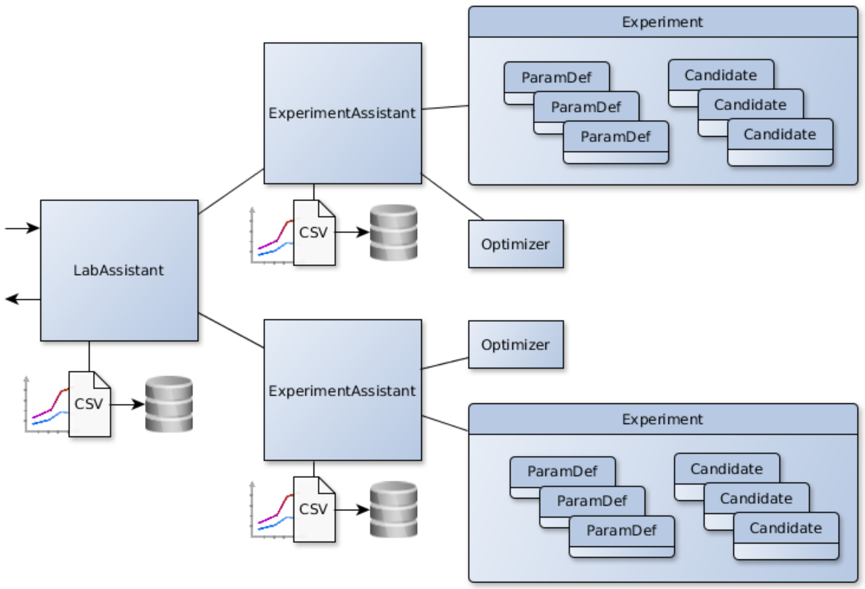


Fig. 6. Using lab assistants to manage and compare several experiments at once.

`BasicLabAssistant` deals with the requirement to run and control several experiments at once. It holds a dictionary of experiment assistants and provides a common interface to all of them following the same semantic as the experiment assistants themselves. The advantages of using lab assistants in contrast to controlling several experiments at the same time manually is that the results can now be stored together and compared. `BasicLabAssistant` provides only that function, while `PrettyLabAssistant` adds comparative plots for the comparison of multiple optimizers.

4 Evaluation

This section evaluates *apsis* on several benchmark and one real world example. All experiments are evaluated using cross validation and 10 initial random samples that are shared between all optimizers in an experiment to ensure comparability.

4.1 Evaluation on Branin Hoo Benchmark Function

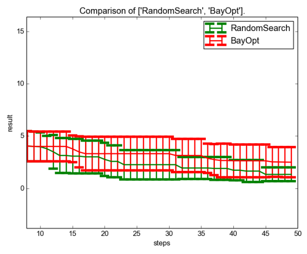
Some recent papers on Bayesian Optimization for machine learning publish evaluations on the Branin Hoo optimization function [**?**]. The Branin Hoo function

$$f_{\text{Branin}}(x) = a \cdot (y - b \cdot x^2 + c \cdot x - r)^2 + s \cdot (1 - t) \cdot cos(x) + s$$

using values proposed in [**?**] is defined as

$$f_{\text{Branin}}(x) = (y - \frac{5.1}{4\pi^2} \cdot x^2 + \frac{5}{\pi} \cdot x - 6)^2 + 10 \cdot (1 - \frac{1}{8\pi}) \cdot cos(x) + 10.$$

In contrast to our expectations Bayesian Optimization was not able to outperform random search on Branin Hoo. Still the result is much more stable and the bayesian optimizer samples only closely at the optimium.



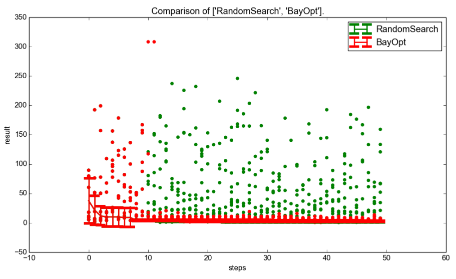


Fig. 7. Comparison of Bayesian Optimization vs. random search optimization on Branin Hoo function. The left picture shows the best-result in every step where random search clearly outperforms Bayesian Optimization. The right plot additionally plots each function evaluation as a dot so one can see that Bayesian Optimization works a lot more stable and does not evaluate so many non-promising candidates as random search.

4.2 Evaluation on Multidimensional Artificial Noise Function

Compared to random search an intelligent optimizer should be better on less noisy function than on very noisy functions in theory. A very noisy function has a tremendous amount of local extrema making it hard to impossible for Bayesian Optimization methods to outperform random search. To investigate this proposition an artificial multidimensional noise function has been implemented in *apsis* as shown in figure 8.

Using this noise function, one can generate multi-dimensional noises with varying smoothness. The construction process first constructs an n-dimensional grid of random points, which remains constant under varying smoothness. Evaluating a point is done by averaging the randomly generated points, weighted by a gaussian with zero mean and varying variance[1]. This variance influences the final smoothness. A one-dimensional example of generated functions for differing

[1] Actually, only the closest few points are evaluated to increase performance.

variances is seen in figure 8.
The result can be seen in figure 9. As expected, Bayesian Optimization outperforms random search for smoother functions, while achieving a rough parity on rough functions.

4.3 Evaluation on Neural Network on MNIST

To evaluate the hyperparameter optimization on a real world problem, we used it to optimize a neural network on the MNIST dataset [**?**]. We used Breze[**?**] as a neural network library[2] in Python.
The network is a simple feed-forward neural network with 784 input neurons, 800 hidden neurons and 10 output neurons. It uses sigmoid units in the hidden layers, and softmax as output. We learn over 100 epochs. These parameters stay fixed throughout the optimization.
For assigning the neural network weights, we use a backpropagation algorithm. Its parameters - step_rate, momentum and decay - are optimized over, as is c_{wd}, a weight penalty term. Hence, resulting in a four dimensional hyperparameter optimization.
We ran all neural network experiments with a five-fold cross validation. Even so, total evaluation time ran close to 24 hours on an Nvidia Quadro K2100M.
Figure 10 shows the performance of the optimizers for each step. As can be seen, Bayesian Optimization - after the first ten, shared steps, rapidly improves the performance of the neural network by a huge amount. This is significantly more stable than the random search optimizer it is compared with.

However, the optimization above uses no previous knowledge of the problem. In an attempt to investigate the influence of such previous knowledge, we then set the parameter definition for the step_rate to assume it to be close to 0, and the decay to be close to 1. This is assumed to be knowledge easily obtainable from any neural network tutorial.
The effects of this can be seen in figure 11, and are dramatic. First of all, even random search performs significantly better than before, reaching a similar value as the uninformed Bayesian Optimization. Bayesian optimization profits, too, and decreases the mean error by about half.

[2] Breze uses theano [**? ?**] in the background.

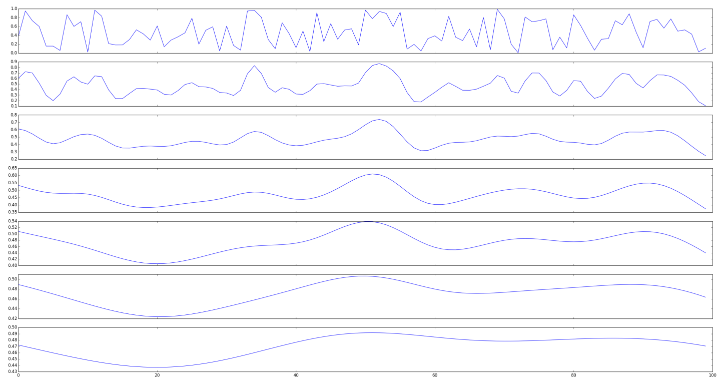


Fig. 8. Plot of artificial noise function used as an optimization benchmark in *apsis*. This is generated using a grid of random values smoothed over by a gaussian of varying variance.

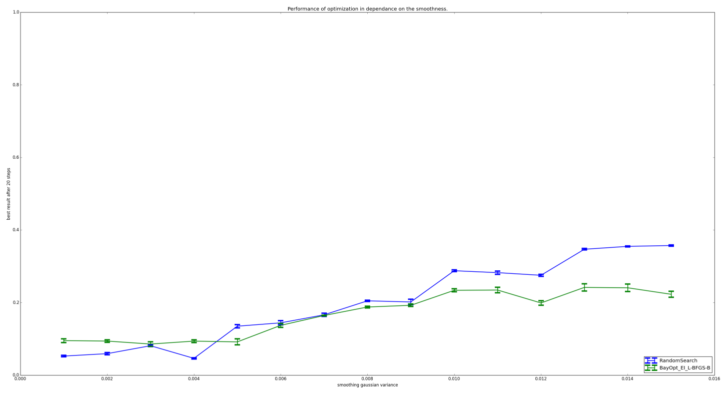


Fig. 9. Plot of the end result after 20 optimization steps on a 3D artificial noise problem depending on the smoothing used. Values to the right are for smoother functions. A lower result is better.

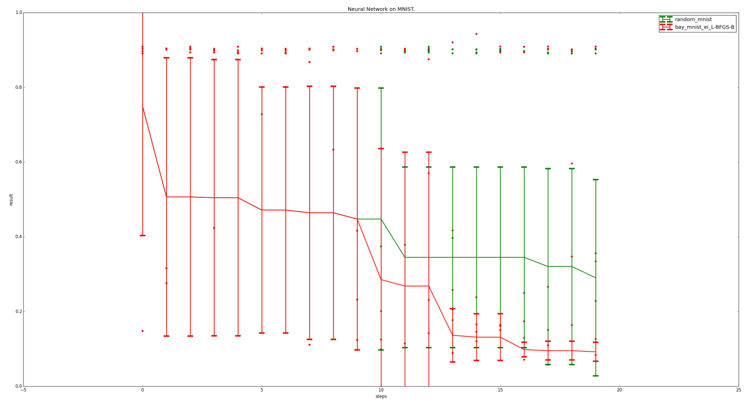


Fig. 10. Comparison of random search and Bayesian Optimization in the context of a neural network. Each point represents one parameter evaluation of the respective algorithm. The line represents the mean result of the algorithm at the corresponding step including the boundaries of the 75% confidence interval.

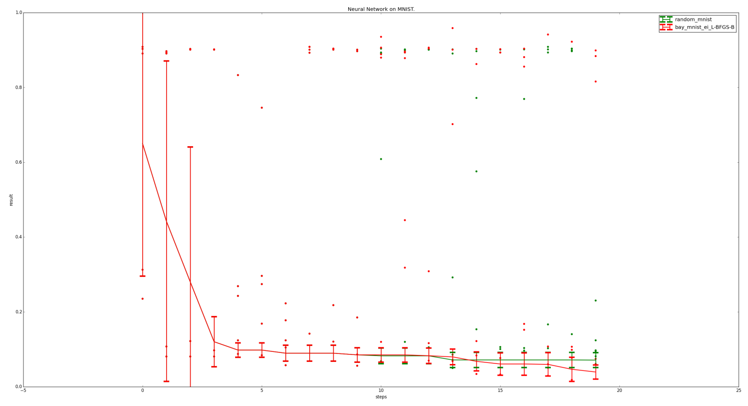


Fig. 11. Comparison of random search and Bayesian Optimization in the context of a neural network. This optimization uses additional knowledge in that step_rate is assumed to be close to 0 and decay to be close to 1.

5 Conclusion

With *apsis* we presented a flexible open source framework for optimization of hyperparameters in machine learning algorithms. It implements most of the state of the art on hyperparameter optimization of current research and is open for further extension.
It is our hope that *apsis* will continue to be expanded, and will be used extensively in academia.

The performance evaluation justifies that Bayesian Optimization is significantly better than random search on a real world machine learning problem. Furthermore the need for an efficient parameter optimization arises already on an experiment with only fifteen minutes of computation time per evaluation. These settings will certainly be met by any practical machine learning problem. Including very simple prior knowledge of the algorithm leads to another significant performance improvement.

There are several areas in which *apsis* can be further improved. We would like to implement support for multiple concurrent workers, allowing us to optimize on clusters. Adding a REST web-interface allows an easy integration of *apsis* to arbitrary languages and environments.
There also remain possible improvements on the implemented bayesian optimizer itself. [**?**] propose using early validation to abort the evaluation of unpromising hyperparameters. [**?**] propose learning input warpings, which would further reduce the reliance on previous knowledge. [**?**] use a cost function to minimize total computing time instead of the number of evaluations. The problem generally known as a tree-structured configuration space as pointed out by [**?**] could be tackled, allowing us to mark parameters as unused for a certain evaluation. Student-t processes might be investigated as a replacement for gaussian processes. Lastly, adding efficient support for nominal parameters in Bayesian Optimization is important for a good performance, though since no publication on that topic exists so far this might be one of the biggest barriers to tackle.