Bayesian Optimization

solving for f(x) since 1974.

Bayesian Optimization

BY RILEY EDMUNDS AND RAHIL MATHUR

Introduction

A problem that comes up often in machine learning is trying to optimize a function that is extremely computationally expensive to evaluate. Imagine determining the best set of hyperparameters to minimize a loss function. In order to evaluate how good our hyperparameters are, we need to retrain our model and compute our loss function again. This process can be extremely time consuming and repeated trials to find a good set of hyperparameters can take an incredibly long time. In these situations, optimizing our loss function can be very expensive given the number of expensive evaluations necessary, so we need to find an alternative method. A different approach to solving this problem is to use Bayesian Optimization.

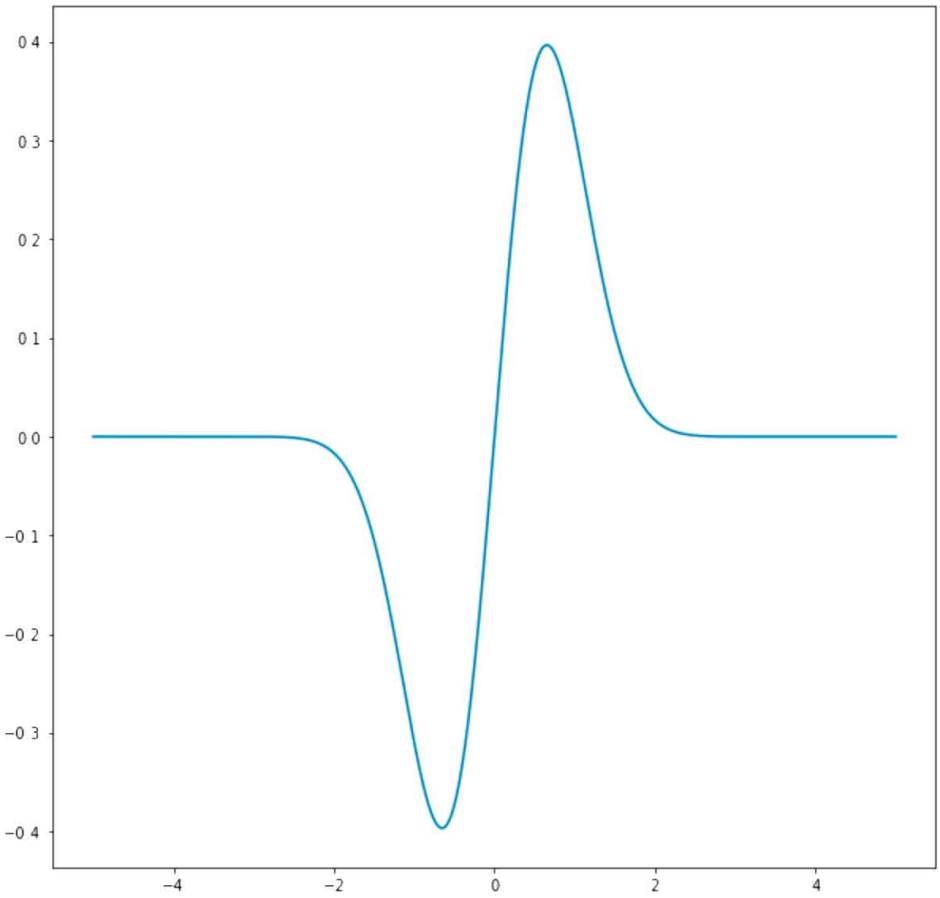
As the Bayesian part of the title may suggest, we use priors in order to make smarter decisions about sampling during optimizing our function in order to reach an optimum faster by minimizing the number of function evaluations we need to make (in the preceding example, number of times we retrain our model). By incorporating prior beliefs in calculating a posterior distribution over objective functions, we can construct an acquisition function that informs us where to next sample points from our domain in order to reach an optimum quickly.

In this article, we aim to discuss both some of the math and some of the applications of Bayesian optimization while providing as much intuition behind the underlying theory as possible. We hope to help demystify the concept of bayesian optimization, particularly in the context of deep learning.

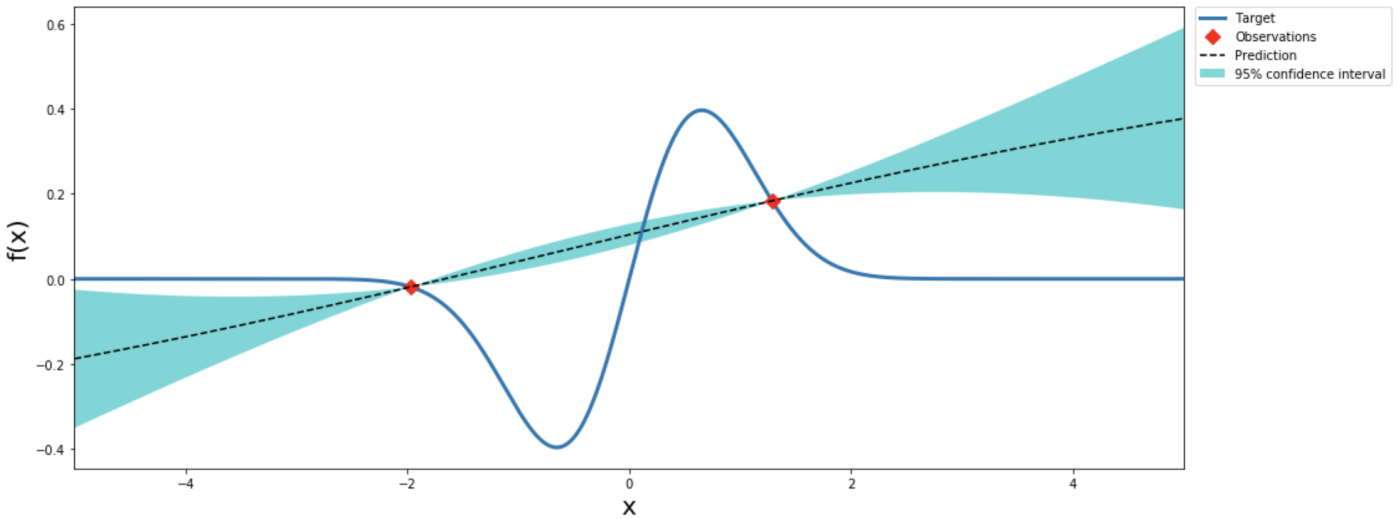
Gaussian Processes

To begin the discussion of the math behind Bayesian Optimization, we start with the topic of Gaussian Processes. Before we go into the formal math behind a Gaussian Process, let's start with a little bit of intuition. Imagine we have an target function $f(x)$ that we are trying to optimize. For the purposes of this example, let's take

$f(x) = \e^{-x^2} \cdot sin(x)$ as our target on the domain $[-5, 5]$ (acknowledging that in the context of Bayesian Optimization, we would likely have a much more expensive function, that we couldn't easily plot), which looks like

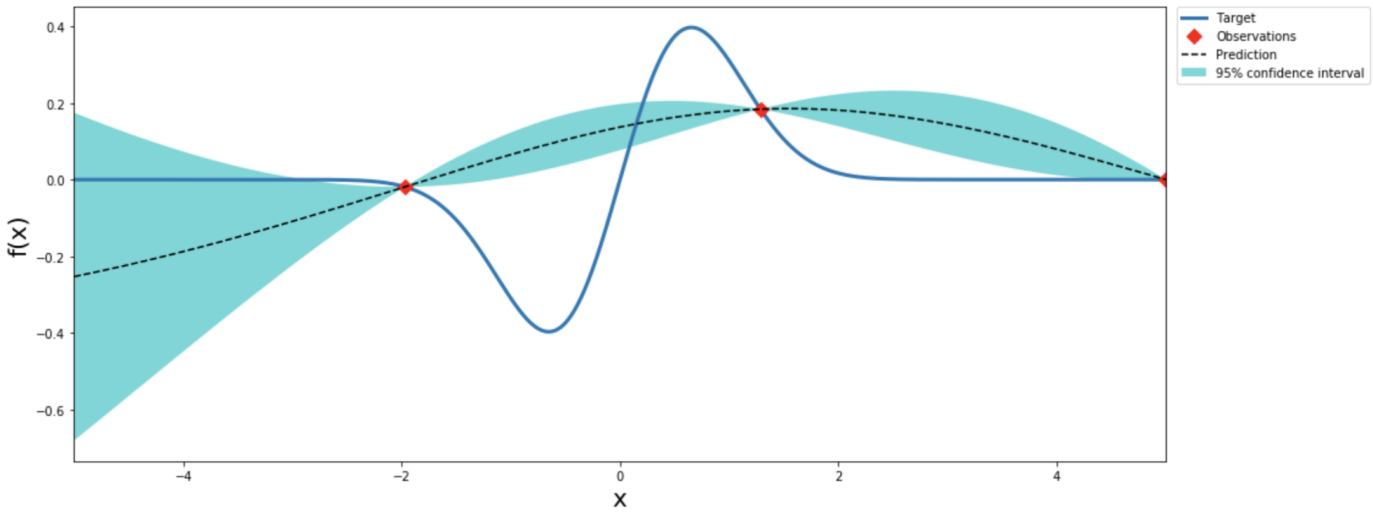


To start fitting a Gaussian Process, we need some initial points. Let's randomly sample two points from our domain and evaluate them, giving us,

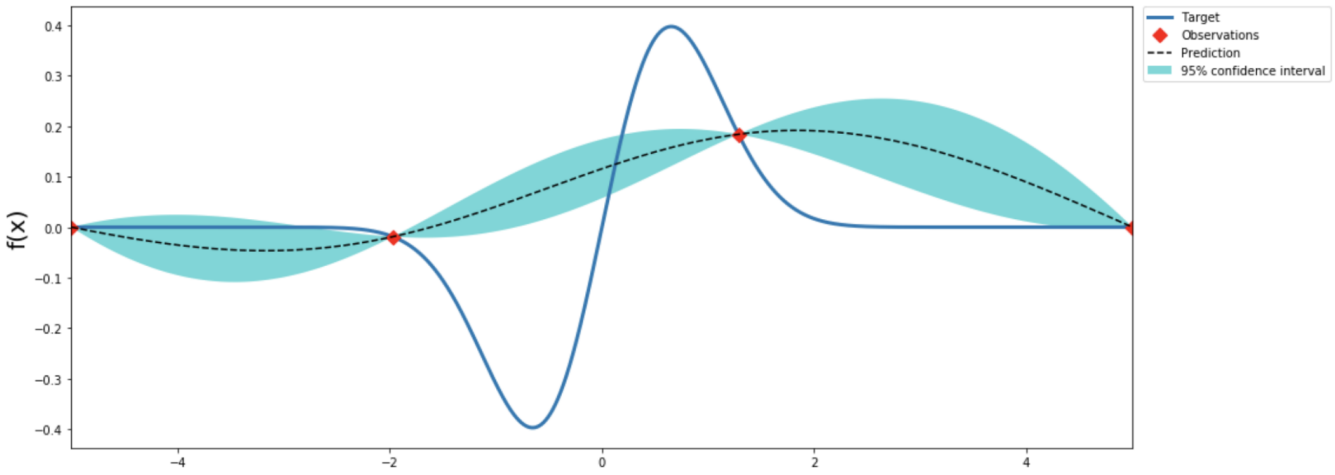


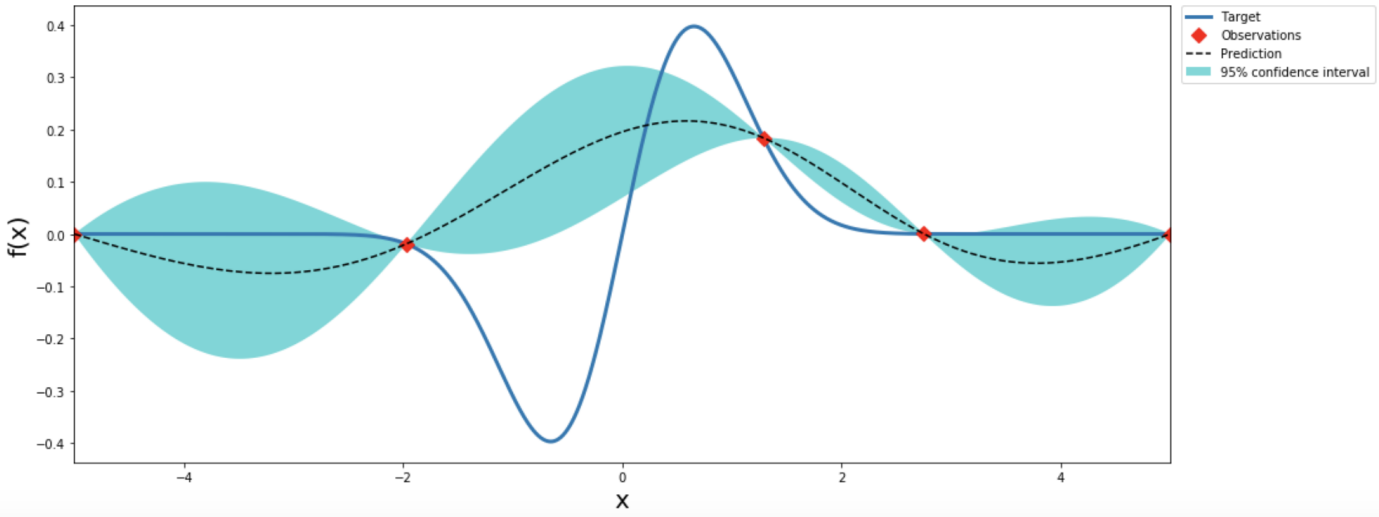
In the above plot, the blue curve represents our target and the black dashed line is our current prediction of the target. The light blue region represents our confidence interval (level of uncertainty about the predicted value of our

fitted function). The further we move away from our sample points, the wider our confidence interval gets (uncertainty increases). It's clear that we need more information, so let's sample another point.

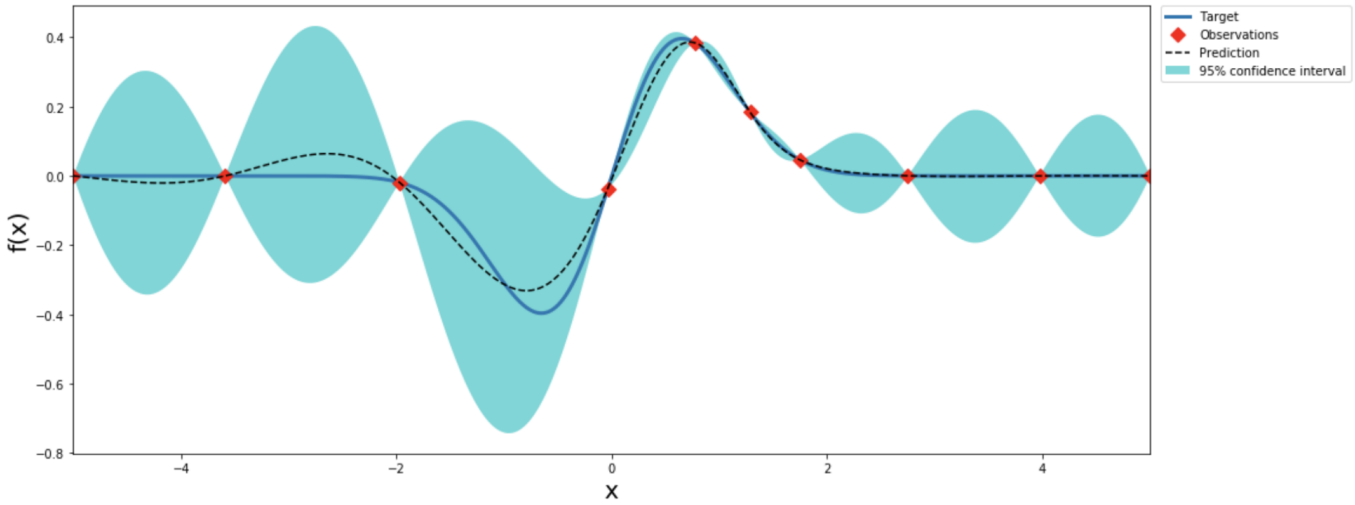


Let's sample two more times.





After three iterations from our initial state of two randomly sampled points, we can see that our confidence bound is starting to get closer to the true optimum of the function. And after five more iterations,



In just eight steps, we were able to get quite close to the optimum and our prediction is relatively close to the target. Compare this with sampling at every point in the function's domain, and the benefits of Bayesian Optimization quickly become apparent. It's important to note at this stage that when we perform Bayesian optimization, our goal is not to necessarily fit the underlying function but to just find the optimum. So, while in a simple example like this we are able to get a good fit to the target, it's more important to note that we were able to get close to the optimum.

In the above example, there are quite a few questions that naturally arise, such as how do Gaussian Processes actually work or how do we actually choose the point to sample next? We'll start with the first question.

We would like to know the value of the true function $f(x)$, such that we can find its optima! In the context of Bayesian Optimization, evaluation of $f(x)$ at every points x in the function's domain is prohibitively expensive. However, not all points are made equal – what if we could evaluate only at those points which are most promising? This problem is likely far more efficient. The next question that naturally comes to mind is: how do we evaluate whether a point is promising? This is where Gaussian Processes come in – they allow us to probabilistically model an unknown function (given observed samples of said function) such that, in the context of Bayesian Optimization, we can determine which point to evaluate next in search of an optima.

Formally, a Gaussian Process (GP) is akin to a gaussian distribution over function space. A GP defines a prior over function space, and then uses observed data to update the posterior. Suppose we are trying to model the form of some function f, given some set of points over the function's domain $x = x_1, \dots, x_n$. Gaussian Processes assume the distribution over outputs $p(f(x_1) \dots f(x_n))$ is jointly gaussian, with some mean μ_x and some covariance $\Sigma(x)$, where $\Sigma_{ij} = k(x_i, x_j)$ and k is a positive definite kernel function, chosen such that if two points are deemed similar by the kernel, they are also deemed similar by the function itself. To reiterate, we would like to learn a distribution of output values $f(x_*)$, conditioned on known inputs output pairs $(x, f(x))$, and new inputs x_*, and assuming that $f(x)$ and $f(x_*)$ are jointly gaussian. From the joint distribution $p(f(x), f(x_*))$, we can derive the posterior $p(f(x_*)|f(x))$, along with its defining mean μ_*, and covariance matrix Σ_*. That is all.

We have defined have a distribution over values of $f(x_*)$, for any input x_*. We can now query the true function based on which point x_* on this distribution our acquisition function tells us is most favorable.

When defining the Gaussian Process, we must make some choices. We choose our prior over the GP functions by selecting our:

- Positive definite covariance function $C : X \times X \rightarrow R$
- Mean function $m : X \rightarrow R$

It turns out that Gaussian Processes are super sensitive to these choices of priors. Intuitively, we need to make assumptions about the nature of the true function in order to define a predictive positive definite kernel (if two points are deemed similar by the kernel, they are also deemed similar by the function itself). We elaborate further on these choices in the Kernel Selection section below.

Acquisition Functions

How do we choose where to observe the function next? Bayesian Optimization uses an acquisition function $a(x)$ to tell us how promising an observation will be. The acquisition function is then used to choose where to evaluate next is our "proxy optimization".

When choosing where to evaluate next, we have two opposing goals. We would like to incentivize exploration of neighborhoods which have not been explored, and also incentivize exploitation of areas we have high confidence will have low value. More rigorously, we can think of exploration as seeking places with high variance, and exploitation as seeking places with low mean.

An ideal acquisition function would:

- Have *high* value for points we expect to be optimal
- Have *high* value for points we have not explored
- Have *low* values for points we have already visited

We would like to choose an acquisition function which has all these qualities.

Probability of Improvement

Likely the first acquisition function used in Bayesian Optimization was *Probability of Improvement*. This acquisition function rewards evaluation of f at any point which is likely to improve upon the current minimum known value of f.

$$PI = P(f(\theta) < \lambda)$$

(where λ is the best value so far, and $f(\theta)$ is the value at the point we will explore)

The point on the function landscape with the highest probability of improvement is chosen and evaluated next. This is the point with maximal expected utility under the PI objective.

Unfortunately, the *probability of improvement* reward of observation does not depend on the expected difference between the current best known value of f, and the value at the observation. Thus it encourages evaluation in places

we are highly confident will improve upon the current minimum, independent of the size of improvement. This acquisition function thus tends to have low exploration, and empirically evaluate points near those we have already explored. This quasi-exploitative behavior can lead to getting stuck in local optima.

Expected Improvement

A more desirable (and consequently more popular) acquisition function is *Expected Improvement*. This acquisition function rewards evaluation of f relative to the magnitude of its expected improvement over the current best, if the evaluation beats the current best, and otherwise, it gives to reward.

$$EI = E[max(\lambda - f(\theta), 0)]$$

(where λ is the best value so far, $f(\theta)$ is the value at the point we will explore, and E is expectation)

The point on the function landscape with the highest expected improvement is chosen and evaluated next. This is the point with maximal expected utility under the EI objective.

Expected improvement allows for both exploration of risky unexplored areas (as the acquisition function is not lower if we explore a point which is much worse than what we already have – notice we're taking max of 0 and improvement over current best λ) and exploitation of areas we are highly confident will give us improvement.

In practice, *Expected Improvement* is by far the most common acquisition function used in Bayesian Optimization.

Often, choise of acquisition function is the most important determinant of success when using Bayesian Optimization, and its choice should depend on careful evaluation of the nature of the dependence between the hyperparameters and the optimization itself. Sometimes, the default acquisition function is not the best choice. To this end, we include two additional less popular acquisition functions: Entropy Search, and Upper Confidence Bound.

Entropy Search

The *Entropy Search* acquisition function acknowledges that we are seeking the location of the optimal value x^* of f:

$$x^* = argmin_{x \in X} f(x)$$

It seeks to reduce the entropy of the distribution over x^*, $p(x^*|\mathcal{D})$. Unfortunately, given the complicated nature of this distribution, we must make approximations.

Upper Confidence Bound

The *Upper Confidence Bound* acquisition function (commonly known as GP-UCB) contains explicit exploration and exploitation terms, and has strong theoretical backing for its convergence to global optima, in the long term. It takes the form:

$$a_{UCB}(x; \beta) = 1[f(x) \leq f']](x) - \beta\sigma(x)$$

Where $\sigma = \sqrt{K(x, y)}$ is the marginal standard deviation of $f(x)$.

Quick Aside: Epsilon-Greedy Search

A big question that arises when sampling in order to determine an optimum is whether to be sampling points according to where you currently think the optimum is or to be sampling points from regions where you have little to no information currently. This question is often phrased as balancing 'exploiting' and 'exploring'. A very common approach to balancing this tradeoff is to use a method called epsilon-greedy search. We define a parameter $\epsilon \in [0, 1]$ such that we sample points to exploit with $\Pr(1 - \epsilon)$ and sample points to explore with $\Pr(\epsilon)$. While one can keep ϵ constant throughout the sampling process, it makes sense to instead use a decay on the parameter such that we explore most at the beginning when we have no information and exploit later on when we have information from previous samples.

The pseudocode is as follows:

```
def epsilon_greedy_select(eps):
        '''choose to explore with P(eps) and to exploit with P(1-eps)'''
        rand = random(low=0,high=1)
        is (rand > eps):
                exploit()
        else:
                explore()
```

Part of the purpose of the acquisition functions mentioned above is to balance this tradeoff between exploring the domain in order to reduce the size of the confidence interval around our mean prediction and exploiting the mean prediction we have to move towards the highest predicted mean region. While traditional methods like Epsilon-Greedy Search balance exploring and exploiting through explicitly generating a random number and making a decision, acquisition functions achieve similar functionality implicitly through the construction of the function. For example, the upper confidence bound acquisition function previously mentioned has a β parameter which can be adjusted to encourage exploration or exploitation.

Kernel Selection / Covariance

An interesting fact about Gaussian processes is that we can define them solely in terms of their second order statistics, the mean and the covariance function. Often, we find the mean is 0 so that the covariance function completely defines Gaussian process. This indicates that the choice of covariance function is a pivotal one in ensuring we model our problem correctly. The covariance function that defines a Gaussian process does so by inducing a level of smoothness on the prior. If we think that nearby points should have nearby objective values, we should aim to choose a covariance function that induces a smooth prior. If we think there should be more variance in the objective values of nearby points, we should choose a different covariance function.

Two families of covariance functions that are of note are the Matérn covariance and rational quadratic covariance functional families.

Matérn Covariance Function Family

The Matérn covariance function family is a parametric family that describes the statistical covariance between points with respect to the distance between the points. Because this covariance depends solely on the distance between points, we say that this family is stationary. Matérn covariance functions take the following form:

$$C_\nu(d) = \sigma^2 \frac{2^{1-\nu}}{\Gamma(\nu)}(\sqrt{2\nu}\frac{d}{\rho})^\nu K_\nu(\sqrt{2\nu}\frac{d}{\rho})$$

The above function is parametrized by the non-negative parameters ρ and ν. Furthermore, Γ is the standard gamma function and K_ν is the modified Bessel function of the second kind. Functions that are generated from this covariance function through Gaussian processes are $\lfloor \nu - 1 \rfloor$ differentiable, so through controlling ν, you can control the smoothness of the sampled functions.

Included in this family is the exponential covariance function:

$$C(d) = e^{-\frac{d}{V}}$$

and the squared exponential covariance function:

$$C(d) = e^{-(\frac{d}{V})^2}$$

The exponential covariance function leads to sample functions that are not smooth whereas the squared exponential covariance function leads to smooth sample functions.

Rational Quadratic Covariance Function Family

Like the Matérn covariance family, the Rational Quadratic family is also stationary as it describes the statistical covariance as a function of the distance between two points. Rational quadratic covariance functions are given by:

$$C(d) = (1 + \frac{d^2}{2\alpha k^2})^{-\alpha}$$

These functions are parametrized by the non-negative parameters α and k. Sampled functions using this covariance function tend to be quite smooth. An interesting note about this covariance family is that these functions can be represented as infinite sums of squared exponential covariance functions, hence the induced smoothness.

An Example of Bayesian Optimization

Now that we've described the different components of Bayesian Optimization, let's go into an interactive example. Here is a webapp where you can step through the optimization process using the sample example as above. You can select which acquisition function you would like to use (either Expected Improvement or Upper Confidence Bound) and press the 'update' button to sample and fit our Gaussian process. By changing the acquisition function, you can see how the utility landscape changes according to the induced goals of the acquisition function.

(The app will sit here once we fix webhosting, but instructions to run locally are attached).

Modifications on Bayesian Optimization

We've now gone through a pretty standard example of what Bayesian optimization looks like. However, it's important to mention some of the recent developments in this field.

While bayesian optimization can be used to "black box" hyperparameter tuning, it shouldn't always be used as a black box. Oftentimes we are working with hardware that has limited resources, or optimizing under a deadline!

Fortunately, there exist improvements upon vanilla Bayesian Optimization to account for these scenarios.

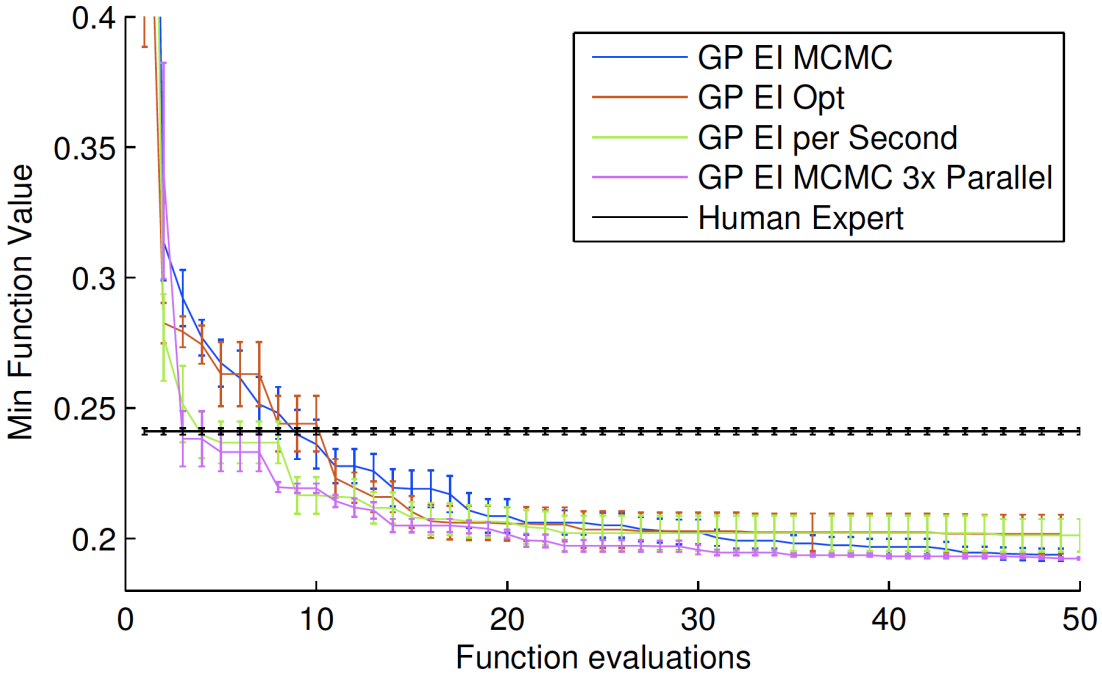
Expected improvement per second

Resource requirements for evaluation of the objective function can scale depending on the region in which we are evaluating. In this vein, we may want to perform cheap evaluations first, even if they are less optimal than their expensive counterparts. Bayesian optimization can thus yield better improvements in the objective function per second by using time-weighing in its acquisition function, especially for highly nonuniform distributions of time-to-evaluate. For example, for optimization in hardware systems with limited resources, it may take longer to evaluate the objective for particularly high values of parameters (which consume these limited resources). Alternatively, imagine optimization for deep learning: one parameter to Bayesian Optimization may be network depth – in this case training will likely take longer to converge for very deep networks. Using expected improvement per second may prioritize evaluation of a shallower network before training a very expensive deep network to convergence, and consequently reduce the overall time required to converge to the global optima.

Parallelizing Bayesian Optimization

Classical Bayesian Optimization is sequential. We choose a point to evaluate, wait for the evaluation to finish, and repeat. We can improve upon this traditional approach in a couple ways. First, we can evaluate the objective function at multiple next possible locations. To efficiently choose multiple points of evaluation, we *fantasize outcomes* from the gaussian process. Concretely, we use predictions of possible objective values for the next evaluation to choose promising two-step-ahead points to evaluate (points which look promising under the *average future*). We then evaluate all these points in parallel, and repeat. To fully parallelize, we can take some number of fantasies, and integrate their posteriors to yield an overall expected improvement. Working on that distribution for point selection empirically outperforms other parallelization methods.

Hyperparameter Tuning of Deep Networks using Bayesian Optimization



(a)

Snoek et al. (2012) explore variants of Bayesian Optimization for tuning the hyperparameters of a convolutional neural network on CIFAR-10 (source: https://arxiv.org/pdf/1206.2944.pdf)

Snoek et al. found that they could surpass human-expert level validation accuracy when tuning the network's hyperparameters using Bayesian Optimization. Hyperparameters included: parameters to each layer (weight priors, pooling sizes), learning rate, number of epochs to train the model for, and others. They achieved using a 3% improvement on CIFAR-10's state-of-the-art using *GP EI MCMC* (Gaussian Process with an acquisition function computed by marginalizing out all hyperparameters and sampling the posterior over GP hyperparameters to yield a Monte Carlo estimate of the integrated Expected Improvement).

Applications Beyond Hyperparameter Tuning

While Bayesian Optimization is not often used in practice beyond its uses for hyperparameter tuning (due to its computationally demanding nature), there is power in black-box optimization. Since it's a black-box function optimizer, it can be used for any optimization task parametrized by some θ. Thus, it can be used with very exploratory, novel approaches, even when other gradient-based optimizers cannot be used.

Benefits & Drawbacks

Unfortunately, while Bayesian Optimization can yield improvements in the efficiency with which we search through parameter space, it does not reduce the dimensionality of said space. Consequently, it is plagued (just like the true function) by the curse of dimensionality. Lastly, it is ironic that the algorithm we employ to automate tuning of hyperparameters has its own hyperparameters. Thus Bayesian Optimization fails in turning this problem into a complete black-box, as we still need to choose a kernel, define a covariance function, and tune the parameters to the gaussian process. Additionally, the algorithm's performance is often highly sensitive to choice of these parameters.

However, for particularly intricate models, such as CNNs, the hyperparameters to the Bayesian Optimization meta-optimizer may be far more intuitive to hand-tune, or may have a far smaller search space over which we would need to perform random or grid search.

Feel free to send any questions to edmunds@ml.berkeley.edu and rahilmathur@berkeley.edu.

Cheers!

References

Adams, Ryan P. A Tutorial on Bayesian Optimization for Machine Learning , 2014.

Dewancker, I., M. McCourt, and S. Clark. “Bayesian Optimization for Machine Learning : A Practical Guidebook.” ArXiv e-prints , December 2016. arXiv: 1612.04858 [cs.LG].

Duvenaud, D., J. R. Lloyd, R. Grosse, J. B. Tenenbaum, and Z. Ghahra- mani. “Structure Discovery in Nonparametric Regression through Compo- sitional Kernel Search.” ArXiv e-prints , February 2013. arXiv:1302. [stat.ML].

Gonzalez, J., Z. Dai, A. Damianou, and N. D. Lawrence. “Preferential Bayesian Optimization.” ArXiv e-prints , April 2017. arXiv:1704.03651 [stat.ML].

Grosse, Roger. CSC321 Lecture 21: A Tutorial on Bayesian Optimization for Machine Learning.

Hannestad, S., and T. Tram. “Optimal prior for Bayesian inference in a con- strained parameter space.” ArXiv e-prints , October 2017. arXiv: .08899.

Jasper Snoek, Hugo Larochelle, and Ryan P. Adams. Spearmint .https:// github.com/JasperSnoek/spearmint, August 2013.

Martinez-Cantin, R. “BayesOpt: A Bayesian Optimization Library for Nonlin- ear Optimization, Experimental Design and Bandits.” ArXiv e-prints, May 2014, arXiv:1405.7430 [cs.LG].

Murphy, Kevin P. Machine Learning: A Probabilistic Perspective. The MIT Press, 2012. isbn: 0262018020, 9780262018029.

Noguiera, Fernando (Python Implementation of Global Optimization with Gaussian Processes) https://github.com/fmfn/BayesianOptimization, 2011.

Pham, Vu. *Bayesian Optimization for Hyperparameter Tuning* .https://arimo.com/data-science/2016/bayesianoptimization-hyperparameter-tuning/. Blog, 2016.

Rainforth, T., T. A. Le, J.-W. van de Meent, M. A. Osborne, and F. Wood. “Bayesian Optimization for Probabilistic Programs.” *ArXiv e-prints* , July 2017. arXiv:1707.04314 [stat.ML].

Snoek, J., H. Larochelle, and R. P. Adams. “Practical Bayesian Optimization of Machine Learning Algorithms.” *ArXiv e-prints* , June 2012. arXiv:1206. 2944 [stat.ML].

Wang, Z., C. Gehring, P. Kohli, and S. Jegelka. “Ensemble Bayesian Optimiza- tion.” *ArXiv e-prints* , June 2017. arXiv:1706.01445 [stat.ML].

Previous