Bayesian Optimization Implementation Report

Background

Bayesian Optimization is a sequential global optimization strategy for black-box functions. There are two main advantages of this method: it does not require derivatives of target function, and it takes relatively small amount of evaluations of the target function. These features make Bayesian Optimization a popular choice to optimize hyperparameters of neural networks.

Spearmint is a popular opensource Bayesian optimizer. It's used to evaluate the performance of my codes.

Methodology

Since the objective function is unknown, the Bayesian strategy is to treat it as a random function (i.e. sampled from Gaussian Process) and place a prior over it. The prior captures our beliefs about the behavior of the function. After gathering the function evaluations, which are treated as data, the prior is updated to form the posterior distribution over the objective function. The posterior distribution, in turn, is used to construct an acquisition function (often also referred to as infill sampling criteria) that determines what the next query point should be.

1. Gaussian Process

   a Gaussian process is a stochastic process (a collection of random variables indexed by time or space), such that every finite collection of those random variables has a multivariate normal distribution.

   A Gaussian process $f(x)$ is completely specified by its mean function $m(x)$ and covariance function $k(x, x')$:

   $$f(x) \sim GP(m(x), k(x, x')).$$

   Generally, we consider the noisy version, where observation y is affected by noise:

   $$y = f(x) + \varepsilon ,$$
   $$\varepsilon \sim N(0, {\delta_n}^2).$$

   Based on inputs X and observations Y, we can predict the Gaussian field f_* for new test point x_*:

   $$\bar{f_*} = K(X, x_*)^T [K(X, X) + {\delta_n}^2 I]^{-1} Y,$$
   $$cov(f_*) = K(x_*, x_*) - K(X, x_*)^T [K(X, X) + {\delta_n}^2 I]^{-1} K(X, x_*).$$

   $K(X, x_*)$ is the matrix of $k(x, x')$ given by each pair of x in X and x' in x_*.

2. Covariance Function

   The choice of covariance depends on the beliefs or assumptions about the target function. A common choice is the Matérn class of covariance functions:

$$k_{3/2}(r) = a(1+\frac{\sqrt{3}r}{l})\exp(-\frac{\sqrt{3}r}{l}),$$
$$k_{5/2}(r) = a(1+\frac{\sqrt{5}r}{l}+\frac{5r^2}{3l^2})\exp(-\frac{\sqrt{5}r}{l}).$$

These covariance functions are stationary (invariant to translations). They assume the target function to be differentiable but not necessarily smooth.

3. Bayesian Treatment for Hyperparameters

This Gaussian process regression itself have several hyperparameters: δ_n from noise, and a and l from covariance function. We can use θ to represent these hyperparameters. To obtain the most likely θ, we optimize the marginal likelihood:

$$\ln\big(p(Y|X,\theta)\big) = -\frac{1}{2}Y^T\big[K(X,X)+{\delta_n}^2 I\big]^{-1}Y - \frac{1}{2}\ln\big(\big|K(X,X)+{\delta_n}^2 I\big|\big) - \frac{n}{2}\ln(2\pi).$$

4. Acquisition Function

One of the most popular acquisition function is the Expected Improvement (EI). Given a test point x_* and the current best result y_{min},

$$EI(x_*) = E(\max(y_{min} - f_*, 0)).$$

It can be expressed in closed form:

$$EI(x_*) = (y_{min} - \bar{f}_*)\Phi(\frac{y_{min}-\bar{f}_*}{s}) + s\phi(\frac{y_{min}-\bar{f}_*}{s}),$$
$$s = \sqrt{cov(f_*)}.$$

The point x_* that maximizes EI will be chosen for the next function evaluation.

Implementation

1. Platform

The algorithm is first implemented on Matlab 2017b, for the simplicity of matrix calculation. At the final stage of evaluation, a python (3.5.2) version is also implemented for better communication with the neural network in python.

2. Design Choices

1. Initial Points

At the start of the Bayesian Optimization, first we need some initial points. There are different ways to start. The number of initial points is usually 10 times the dimension of the function inputs based on experience. However, it's flexible. Spearmint seems to start with only two points with no problem. How to choose the initial points is another question. I chose to use just random points at the beginning, but switched to Latin hypercube later. Latin hypercube sampling tries to sample points in a way that all the points are almost uniformly distributed in each dimension of the high

dimensional space. The good spreading feature can help building a more demonstrative initial model.

2. Optimizers
   There are two sub optimization problems in the framework, one is the optimization of marginal likelihood for hyperparameter estimations. The other one is the optimization of expected improvement for choosing next evaluation position. Thus, we need additional optimizers.

   For the Matlab code, I have tried two optimizers, one is **fmincon** using quasi-newton algorithm. The other one is **Covariance Matrix Adaptation Evolution Strategy** (CMAES). Both the two optimizers can work without gradient. The advantage for the first one is that it is a built-in function of Matlab, thus more trustable. However, it does not naturally provide global minimum, we must try multiple initial points for that. The CMAES code is not built-in function, but it is posted on Yarpiz. It requires a little work for the correct boundary constraint though. It does not require initial points, but the number of iterations must be specified. By a few tests, it was found that CMAES can find better minima in similar time in more than 90% cases. Thus, I choose the CMAES optimizer.

   For the python code, I found a CMAES package but the feature it provides is different from the Matlab version. It requires initial points and the constraints cannot be applied. I tried several optimizers in python and chose to use another evolution algorithm, **differential evolution**. It provides similar features as CMAES in Matlab, But it's slower.

3. Hyperparameter Estimation
   In the standard scenario, the hyperparameters should be re-estimated once a new data point is added. However, due to the computational constraints, the value of marginal likelihood may go to negative infinity sometimes, thus the optimization result may be unreliable. To avoid that, we can do the hyperparameter estimation only once based on only initial points.

   For certain problems, if the function itself is definite, we can assign a small number to the noise term in hyperparameters. That can guarantee a good regression result when the hyperparameter tends to assign a large noise. This treatment could be understood as providing prior knowledge.

3. Technical Issues and Solutions

1. Singularity of Matrix
   The covariance matrix $K(X, X)$ should always be positive definite in theory. However, in some cases, when two points in X are too close, the two rows or columns

might be almost the same, and the matrix will be singular, giving zero or negative determinant and cause a lot of trouble. This happens a lot when I used squared exponential covariance function. It happened less after I switched to Matérn class covariance functions because they are better-conditioned.

Another solution could be adding a small number to the diagonal of the covariance matrix. It's easy to see that this number behaves the same as the noise term ${\delta_n}^2$, thus can be just combined. We can assign a lower bound to the ${\delta_n}^2$ when we really need that to make the matrix positive definite.

2. Negative Variance
   This is another computational problem, when evaluating $cov(f_*)$, in very rare cases we get negative numbers. That cause problem because we need the standard deviation to calculate EI. It was solved by simple taking the max of $cov(f_*)$ and a small positive number.

Test Problems and Results

1. Branin Function
   Branin function is a 2-variable smooth function usually used for testing of optimization algorithms.

$$f(x) = (x_2 - \frac{5.1{x_1}^2}{4\pi^2} + \frac{5x_1}{\pi} - 6)^2 + 10\left(1 - \frac{1}{8\pi}\right)\cos(x_1) + 10$$

   The minimum is 0.397887.
   The Matlab code was run several times, and it got below 0.3980 within 100 evaluations (20 initial points + 80 iterations). On this task, it had very similar performance compared to Spearmint.

2. 9D Function 2D version
   The 9D function is simplified to 2D in this task. See fig 1 for comparison. Due to some parallel features, Spearmint didn't wait for the result in the first few iterations, so there's to output. As can be seen, the performance of my Matlab code is comparable to Spearmint in this case.

3. 9D Function
   This is the complete and hard task. My code was run several times to evaluate the performance. See fig 2 for comparison. It seems Spearmint reached better minimum much faster, but the first few evaluations are pretty random. And in the long term, two of my runs outperformed Spearmint. Although there was once the optimizer couldn't find the correct region of the minimum.

One thing interesting to mention is that only one run of my code got slightly better minimum than the 2D case after so many evaluations. It's possible that the true minimum just lies close to the domain of the 2D case.

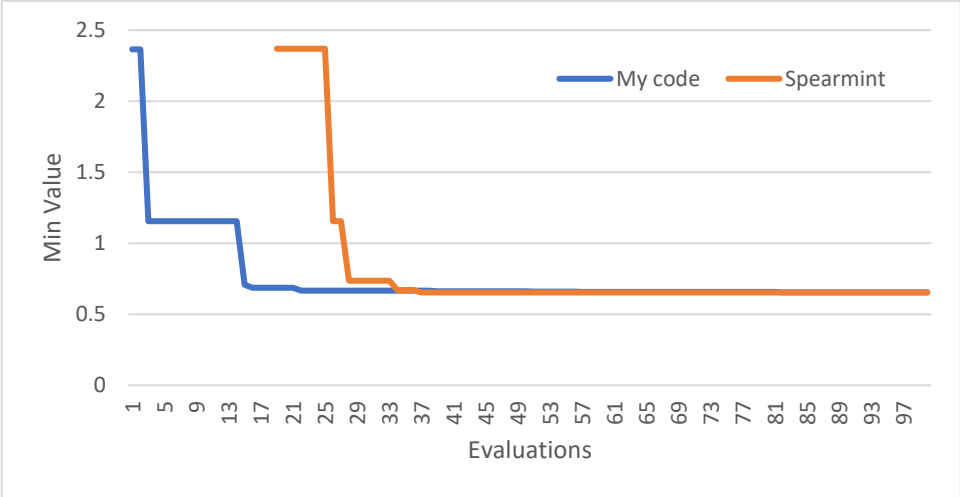

Figure 1 Trace Comparison for 2D Case

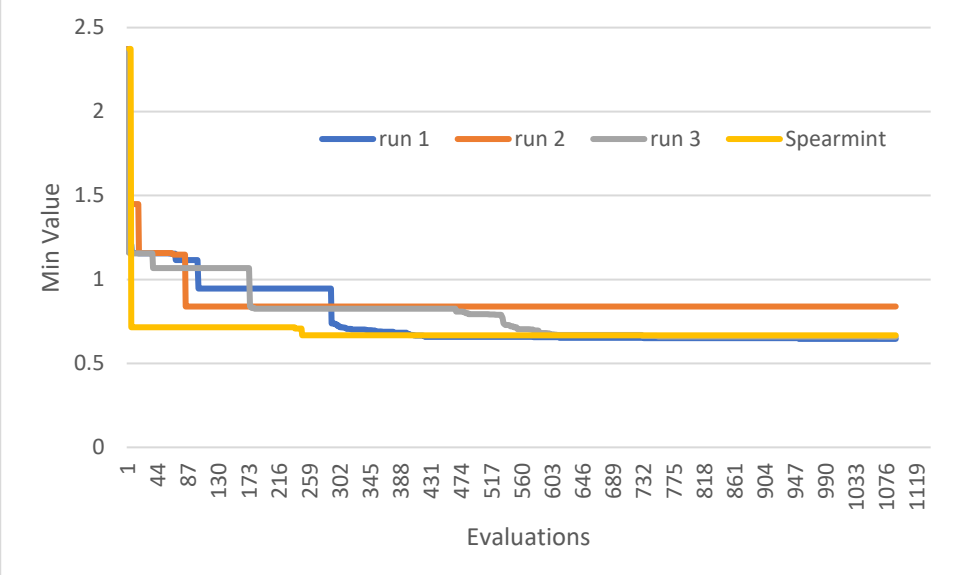

Figure 2 Trace Comparison for 9D Case

4. Neural Network

This neural network is from my final project of Deep Learning. It's a LSTM based 5-layer neural network. It tries to identify the author of given text from 4 candidates. For the evaluation of the Bayesian Optimization algorithm, the training dataset is reduced. The task is to tune the network size for the first 4 layers. See fig 3 for comparison. Since the result didn't seem so good, I also added Latin Hypercube sampling and random sampling.

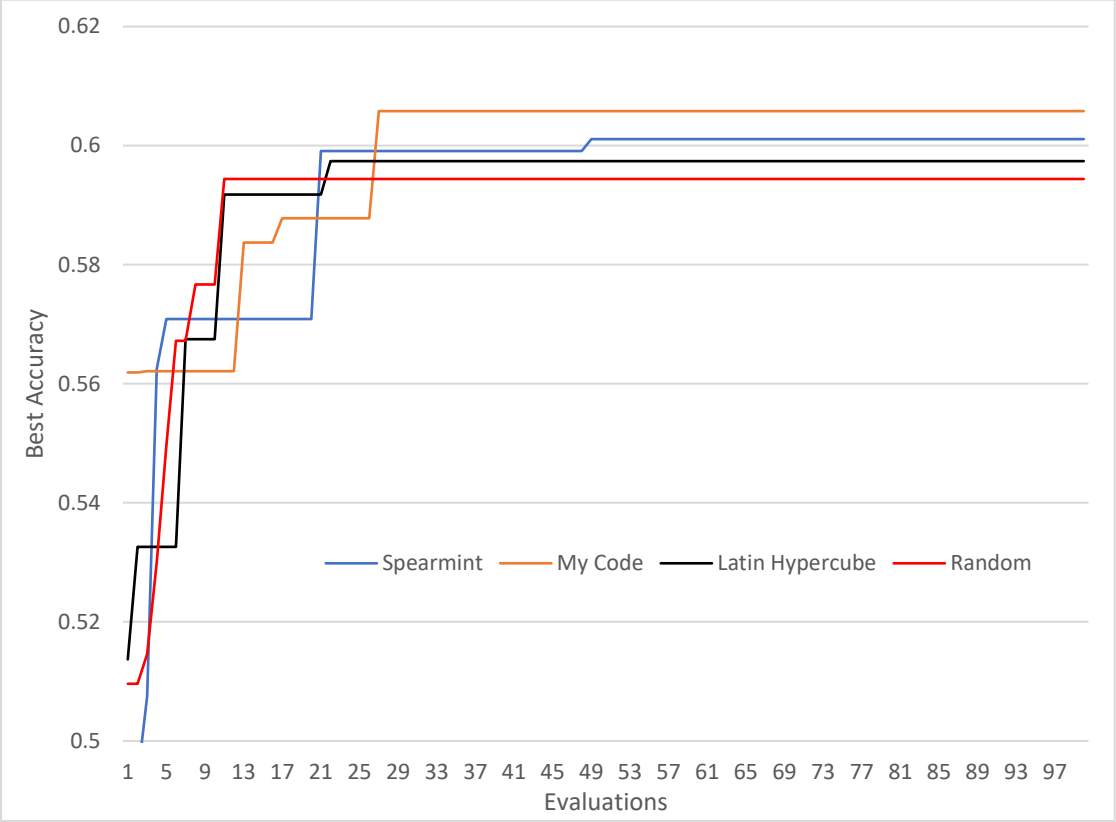

Figure 3 Comparison for Neural Network

It can be seen that the performances of the 4 sampling are relatively close. Bayesian Optimization doesn't provide significantly better result for this case.

Discussion

From the comparison, we can conclude that the performance of my code is close to that of Spearmint. Maybe it is not stable enough because something might have gone wrong in the second run of 9D case.

The failure of both Bayesian Optimization method on the neural network case is not so surprising. The network gives an accuracy between 0.5 to 0.6 in a large region. And this function has true noise: randomness from the initialization of weights and optimizer, which are hidden and handled automatically for elegance. Sometimes given the same inputs, the network may output an accuracy from 0.52 to 0.57. It's really a hard problem.

What's more surprising is that the optimized hyperparameters of the network give an even worse accuracy when I try the full training dataset. It seems that this problem shouldn't be modeled this simple. If we really want to optimize the training of full

dataset, maybe we have to try the trick to also model the relative size of training dataset as a variable.

Possible future work:
Add cross-validation at the hyperparameter estimation stage to make sure the hyperparameters fit the model well.
Find better optimizers for EI and marginal likelihood.
Seek better solutions for singularity.
Follow recent improvements on Bayesian Optimization.
Investigate for what kind of neural networks Bayesian Optimization can truly be helpful.