Time-Varying Bayesian Optimization and its Extension

時変ベイズ最適化とその拡張

付和文抄訳

A Thesis Presented to the Faculty of
International Christian University
for the Baccalaureate Degree

国際基督教大学教授会提出学士論文

by

Hanbie Lee
211748

June, 2021

Approved by
ISHIBASHI, Keisuke　石橋　圭介
Thesis Advisor　論文指導教授

Contents

1. Introduction

Global optimization is a problem to find the global maximum or minimum of a given function, called the objective function, under some constraints. When the objective function has "nice" properties, such as convexity and cheap evaluation, finding the global optimum is easy. However, in a practical setting, the objective function may not satisfy these properties; the concavity is unknown, the cost of evaluation is expensive, no derivatives are observed, and so on. Bayesian optimization is one of the methods to find the optimum under these conditions.

The most intuitive interpretation of Bayesian optimization is: "educated guess put to mathematical terms". Suppose one tries to find the optimal color for some designing. One does not have any prior information about where the optimum is, but knows only the value of the color in terms of hexadecimal color codes. One tries a certain color and receives some feedback, which updates their view on what the best color is. Continuing the search, one would search for different color, which would also give certain feedback, further updating their view. By repeating this process, one could guess what the best option is at that point according to the information they already know. Bayesian optimization does this process statistically, thus more rigorously.

Bayesian optimization is a sequential method for global optimization problem with black-box functions. The term "black-box" is used due to its lack of information gained when a point on the function is observed. The main difference between Bayesian optimization and other optimization methods is that in Bayesian optimization, one can only obtain the value of the function at some query point; in a "regular" optimization, one can obtain the general shape of the function via derivatives and concavity. Bayesian optimization mainly

has two parts when searching for the optimum of the function, the surrogate model and the loss function. Since the function to be optimized is a black-box, it is much easier to build a surrogate model to optimize, which uses the data that has already been obtained. In Bayesian optimization, the surrogate model is estimated statistically to calculate the prior distribution of the function, which is assumed to follow a Gaussian process. The next part is the loss function, which is used to find the next optimal query. Since most optimization borrows the methods used in bandit optimization, the loss function is represented as regret. Bandit optimization is analogous to Bayesian optimization but the possible choices are finite, discrete, and each possible choices do not necessarily correlate to each other. Given a finite set of choices, the algorithm, also referred to as the player, searches for the best choice to maximize the reward under a limited number of exploration. By minimizing the regret, either for each query or for the cumulative regret, the next query is selected. Bayesian optimization consists of the interaction of these two parts; the loss function finds the next best point using the current prior distribution and updates the posterior distribution which in turn becomes the prior distribution for the next query. Even though theoretically, very few information about the function is known in prior, in reality, much more properties are assumed For example, the function is assumed to be convex, continuous, and each similar input points results in a similar output, which comes from the characteristic of Gaussian process, which is explained in section 3.1.

Bayesian optimization does not have to consider the function to be time dependent, however, one can incorporate it. When the objective function is time dependent, it means that the function $f_t(x)$ will depend on the position and the time $t \in [0, \infty)$. From this point on, when we say at a time t, we will define it to be a positive integer i.e. $t \in \{1, 2, \cdots\}$; the function $f_t(x)$ can also be seen as a sequence of functions, each changing by some randomness. The general procedure for the time varying Bayesian optimization does not change from the time invariant setting; the largest difference is that the function to optimize changes with time, hence some adjustment must be made when building the surrogate model to incorporate a temporal aspect such that early-observed points are not treated equally with newer points. The specific instructions are given in section 3.3.

One of the problems about time-varying Bayesian optimization is that it assumes that the change of the function occurs very small. In order to resolve this problem, we proposed a method that extends on the time varying Bayesian optimization and samples from several points from each time step. The motivation for this idea comes from the fact that since time varying functions are irreversible, searching for several points at a time seems more beneficial than only one point. By searching for several points at a time, we hypothesize that the optimization would still work effectively on functions that change largely.

2. Related Works

There are various papers on time-invariant Bayesian optimization with extensions. Brochu et al. provides an overview on how the Bayesian optimization works, including some applications to problems on reinforcement learning, as well as possible directions for research in Bayesian optimization [2, 3]. Srinivas et al. introduces a new method that uses the technique from multi-armed bandit (MAB) problem, the upper confidence bound (UCB), as well as a regret bound for this algorithm.

In terms of time-varying Bayesian optimization, Bogunovic et al. gives the first analysis under the GP setting [1]. Their work extends from the work done by Srinivas et al. and incorporates a spacial aspect to the kernel, giving two types of time-varying extension algorithms as well as regret bounds for those extensions. The first proposed algorithm resets at every certain multiple of steps to the initial prior distribution. The second algorithm they proposed is to "forget" the past data in a smooth fashion such that a certain hierarchy rises from different points observed at different time step. Although there are very few studies done on time-varying GP settings, there are studies done within the MAB setting. Honda and Nakamura provides several examples, such as adversarial settings, where the agent competes with the adversary with fixed strategies, restless bandit, where each arm's state changes under the Markov chain, and rested bandits, where only the chosen arm's state change [7]. As a different extension from time-varying Bayesian optimization, there is a work by Imamura et al., where the function evaluation does not occur at every time step [4].

3. Method

At the heart of Bayesian optimization, time-varying or not, lies Gaussian process. Therefore, in order to understand how Bayesian optimization algorithm works, one has to know how Gaussian process comes into play.

3.1 Gaussian Process

3.1.1 Gaussian Process

A function f is said to follow a Gaussian process when given a certain input $x_1, x_2, \cdots, x_n \in \mathcal{X}$, where $\mathcal{X}$ is the input space, the vector $\boldsymbol{f} = (f(x_1), f(x_2), \cdots, f(x_n))$ follows a multivariate Gaussian distribution $\mathcal{N}(\boldsymbol{\mu}, K)$ where $\boldsymbol{\mu}$ is the mean vector given by the inputs and K is the covariance matrix where $K[n, n'] = k(x_n, x_{n'})$ for some kernel function k. This overall behavior is denoted as $f \sim \mathcal{GP}(\boldsymbol{\mu}(x), k(x, x'))$. Although the mean vector can be arbitrary, it is useful to normalize such that the mean vector is 0. The two largest difference from a Gaussian distribution is that Gaussian process uses a kernel instead of a variance, and that the "thing" that is sampled from is a function instead of a point. The kernel to be used has some freedom as long as it shows some kind of "closeness" similar to the variance. One of the simplest kernel is the squared exponential, which is in the form

$$SE(x, x') = \exp\left(\frac{||x - x'||^2}{2l^2}\right) \tag{3.1}$$

where l is a parameter and $||\cdot||$ is the euclidean distance. Another kernel which is highly used is the Matern Kernel, is in the given form:

$$Matern(x, x') = \frac{2^{1-\nu}}{\Gamma(\nu)} \left(\frac{\sqrt{2\nu}||x - x'||}{l} \right)^{\nu} \times B_{\nu} \left(\frac{\sqrt{2\nu}||x - x'||}{l} \right) \tag{3.2}$$

where $\nu > 0, l > 0$ are parameters, and $\Gamma(\cdot)$ is the Gamma function and $B_{\nu}(\cdot)$ is the modified Bessel function.

3.1.2 Gaussian Process Regression

Gaussian process regression (GPR) is a regression model which uses Gaussian process to calculate the predicted distribution of a function value, using the data that has already been observed as well as some new input point(s). In this scenario, every sample point has a Gaussian noise mixed into the true function value. GPR tries to calculate the mean and the variance of the posterior distribution with the obtained function in the form:

$$y = f(x) + \varepsilon \quad (\varepsilon \sim \mathcal{N}(0, \varsigma^2)) \tag{3.3}$$

The difference between the noiseless regression and noisy regression is in the covariance matrix. In the noisy regression, the diagonal element includes the noise as:

$$K' = K + \varsigma^2 I$$

where I is the identity matrix. The overall algorithm is described below.

Algorithm 1 GP Regression

Require: Training Data $\mathcal{D} = (x, y)$ $(x = (x_1, x_2, \cdots, x_N), y = (y_1, y_2, \cdots, y_N))$; Test Data $x^* = (x'_1, x'_2, \cdots, x'_M)$; kernel function $kern$; noise ς

1: Let $K[n, n'] := kern(x_n, x_{n'}) + \varsigma^2 \delta(x_n, x_{n'}) \quad (n, n' \in \{1, 2, \cdots, N\})$
2: Let $k_*[n, m] := kern(x_n, x'_m) \quad (n \in \{1, 2, \cdots, N\}, m \in \{1, 2, \cdots, M\})$
3: Let $k_{**}[m, m'] := kern(x'_m, x'_{m'}) \quad (m, m' \in \{1, 2, \cdots, M\})$
4: $\mu_* := k_*^T K^{-1} y$
5: $\sigma_* := k_{**} - k_*^T K^{-1} k_*$
6: **return** μ_*, σ_*

Since similar inputs results in a similar output on a Gaussian distribution, the output vector created by the matrix

$$\begin{pmatrix} K & k_* \\ k_*^T & k_{**} \end{pmatrix}$$

also follows a Gaussian distribution. When the known output vector is denoted as $y = (y_1, y_2 \cdots, y_N)^T$ and the new output vector is denoted as $y' = (y'_1, y'_2, \cdots, y'_M)^T$, the concatenated output vector $y_* = (y, y')^T$ is generated by the Gaussian distribution

$$\begin{pmatrix} y \\ y_* \end{pmatrix} \sim \mathcal{N}\left(\begin{pmatrix} 0 \\ \vdots \\ 0 \end{pmatrix}, \begin{pmatrix} K & k_* \\ k_*^T & k_{**} \end{pmatrix}\right) \tag{3.4}$$

where the mean and covariance matrix are both $n + m$ dimensional. From this, we know that the newly calculated output vector does not need to be only 1, but an arbitrary natural number. K, k_*, k_{**} are the symbols used in the algorithm above, and A^T is the transpose of a matrix A. In order to calculate the predicted distribution of the test outputs, the mean and variance, according to the training data, are calculated. These values are calculated by

$$\mu_* = k_* K^{-1} y \tag{3.5}$$

$$\sigma_* = k_{**} - k_*^T K^{-1} k_* \tag{3.6}$$

which both of them are the return values of the algorithm.

3.2 Time-invariant Bayesian Optimization

In both time-varying setting and time-invariant setting, Bayesian optimization uses GPR to update the posterior probability at each iteration. Since Bayesian optimization is a sequential method, the mean and variance in (3.5) and (3.6) will change to μ_t and σ_t, respectively, where the subscript refers to the value at time t. At each step, a certain mean μ_t and a certain variance σ_t is determined. Using these information, an acquisition function is defined, which is used to find the next "best" point. It is not the "the" best point per se,

as much as the global optimum, but is optimum according to the current information that one has obtained. There are several well known acquisition functions, but in this paper, an upper confidence bound(UCB) function is used. The idea of UCB comes from Multi-armed Bandit Problem(MAB). UCB chooses the next query point where the density of the data is the smallest while being in the range of the expected area. The function is written as follows:

$$UCB_t(x) = \underset{x \in D}{\operatorname{argmax}}\, \mu_t(x) + \alpha_t \sigma_t(x) \tag{3.7}$$

where α_t is a parameter that varies every step t. Depending on the value of this parameter, the next query point would be close to the mean or farther away from the mean. This choice in the parameter results in the trade-off between searching in the area where it is most likely have been searched and searching in the area where it has not been searched as much. This is called the exploration-exploitation trade-off. The overall algorithm for The Bayesian optimization algorithm is described below.

Algorithm 2 GP-UCB Algorithm

Require: Input space D; GP Priors ($\mu_0 = 0$, σ_0, k); Parameters α, ε, σ

1: **for** $t = 1, 2, \cdots$ **do**
2: Choose $x_t = \underset{x \in D}{\operatorname{argmax}}\, \mu_{t-1}(x_{t-1}) + \alpha_t \sigma_{t-1}(x_{t-1})$
3: Sample $y_t = f(x_t) + \varepsilon_t \quad (\varepsilon_t \sim \mathcal{N}(0, \sigma^2)$
4: Perform Bayesian update to obtain μ_t and σ_t
5: **end for**

Since every step is calculated probabilisically, it might not be close to the real value. While maximizing the point, it is also beneficial to minimize some loss. Since GP-UCB is derived from MAB setting, the loss function is derived from MAB, namely the regret. Regret is the difference between the actual maximum and the chosen point, indicating some sort of error in the choice made. In our algorithm a regret is calculated by the cumulative regret, which is written as:

$$R_T = \sum_{t=1}^{T} f_t(x_t^*) - f_t(x_t) \tag{3.8}$$

where x^* is the true maximum and x_t is the estimated position.

3.3 Time-varying Bayesian Optimization

The problem with GP-UCB is that it does not optimize the function if there is a temporal component. Time-varying GP-UCB (TV-GP-UCB) is an extension from GP-UCB such that it takes into account for temporal change in the objective function. The algorithm largely follows the GP-UCB algorithm, with the largest change in the covariance matrix and the change in the objective function for every time step. The behavior of the objective function is as follows:

$$f_1(x) = g_1(x) \tag{3.9}$$

$$f_{t+1}(x) = \sqrt{1-\varepsilon} f_t(x) + \sqrt{\varepsilon} g_{t+1}(x) \tag{3.10}$$

where each function $g_i(x)$ is a random function generated by the Gaussian process with mean 0 and a certain kernel function k i.e., $g_i \sim \mathcal{GP}(0, k)$. The objective function at time $t = 1$ is generated at random by the Gaussian process. For every update, the objective function incorporates the current function with the randomly generated function to create the time varying function. ε lies between the interval $[0, 1]$ which determines how "random" the function will move. If $\varepsilon = 0$, it will be the same as the time-invariant function. If $\varepsilon = 1$, it will be independent between each time step. Regardless of the value of ε, $f_t \sim GP(0, k)$.

The covariance matrix differs from GP-UCB in the sense that older points contribute less to the current shape of the objective function; the covariance matrix has to favor the newer points to the older ones. In order to do so, the calculation of the mean and variance has change as such:

$$\mu'_{t+1}(x) := k'_t(x)^T (K'_t + \varsigma^2 I)^{-1} y_t \tag{3.11}$$

$$\sigma'^2_{t+1}(x) := k(x,x) - k'_t(x)^T (K'_t + \varsigma^2 I)^{-1} k'_t(x) \tag{3.12}$$

$$K'_t := K_t \circ D_t, \quad D_t[i,j] = (1-\varepsilon)^{|i-j|/2} \quad (i, j \in \{1, 2, \cdots, T\})$$

$$k'_t(x) := k_t(x) \circ d_t, \quad d_t[i] = (1-\varepsilon)^{(T+1-i)/2} \quad (i \in \{1, 2, \cdots, T\})$$

where where T is the current time step and $\circ$ is the Hadamard product, where the matrix is multiplied element wise. The pseudocode for this algorithm is shown below.

Algorithm 3 TV-GP-UCB Algorithm

Require: Input space D; GP Priors ($\mu_0 = 0$, σ_0, k); Parameters α, ε; noise ς

1: **for** $t = 1, 2, \cdots$ **do**
2: Choose $x_t = \underset{x \in D}{\operatorname{argmax}}\, \mu_{t-1}(x_{t-1}) + \alpha_t \sigma_{t-1}(x_{t-1})$
3: Sample $y_t = f(x_t) + z_t \quad (z_t \sim \mathcal{N}(0, \varsigma^2)$
4: Perform Bayesian update to obtain μ'_t and σ'_t
5: Update objective function $f_{t+1}(x) = \sqrt{1-\varepsilon} f_t(x) + \sqrt{\varepsilon} g_{t+1}(x)$
6: **end for**

3.4 Multi-sampling Time-varying Bayesian Optimization

While the TV-GP-UCB optimizes the time-varying reward function, it only searches for one point at every time step. Since the change in reward function is irreversible, observing at a single point at each time step may lead to inconsistencies with the past data. In this paper, an algorithm for observing multiple points in each time step is devised. The algorithm is an extension from TV-GP-UCB such that the query point has multiple inputs. In addition to using the UCB function to find the largest point, this algorithm uses it to find the next likely point to check. The parameter λ chooses how much far away to look for. Since the query point already has a certain point to check, namely the point where the UCB function chose, it would be less efficient to check for point too close, but would be too ambitious to find points that are too far away. The pseudocode for this algorithm is shown below.

Algorithm 4 MP-TV-GP-UCB Algorithm

Require: Input space D; GP Priors ($\mu_0 = 0$, σ_0, k); Parameters α, ε, λ; noise ς; sample number p; kernel function $kern$

1: **for** $t = 1, 2, \cdots$ **do**
2: Choose $x_t[1] = \underset{x \in D}{\operatorname{argmax}}\, \mu_{t-1}(x_{t-1}) + \alpha_t \sigma_{t-1}(x_{t-1})$
3: Choose $x_t[i] = \underset{x_i \in D}{\operatorname{argmax}}\, \mu_{t-1}(x_{t-1}) + \alpha_t \sigma_{t-1}(x) - \sum_{j=1}^{i-1} \lambda(kern(x_t[j], x_i)) \quad (i \in \{2, \cdots, p\})$
4: Sample $y_t[i] = f(x_t[i]) + z_t \quad (z_t \sim \mathcal{N}(0, \varsigma^2) \quad (i \in \{1, \cdots, p\})$
5: Perform Bayesian update to obtain μ'_t and σ'_t
6: Update objective function $f_{t+1}(x) = \sqrt{1-\varepsilon} f_t(x) + \sqrt{\varepsilon} g_{t+1}(x)$
7: **end for**

4. Experiments and Results

The algorithm is tested using synthetic data, comparing the average cumulative regret, defined at (3.8), with different parameters. We will also consider four different settings in measuring the regret, which are:

1. Total points searched are same for all settings
2. Case 1 but only the top UCB values contribute to the regret
3. Total steps are same for all settings
4. Case 3 but only the top UCB values contribute to the regret

For cases 1 and 3, the cumulative regret R_T will be averaged over the number of points observed at each time step; 2-point optimization will divide by 2 and 3-point optimization will divide by 3. The settings are largely borrowed by Bogunovic's setting[3]. We consider the input space $D = [0, 1]$ which is discretized equally into 50 spaced points. The data is generated according to the time-varying model stated in section 3.3. The kernel used for the experiment is the squared exponential kernel, with the parameter set as $l = 0.2$. The sample noise is set as $\varsigma = 0.01$. We compared several sample points at each round, namely $p = 1, 2, 3$, each sampling a total of around 200 points. When $p = 1$, it is the same setting as the TV-GP-UCB algorithm. In this experiment, the parameters to test are ε and λ. ε has been included in order to test if it contributes to the optimization of the change of the objective function. Since the initial objective function is generated by the multivariate normal distribution with mean vector with all the elements being 0 and the covariance matrix as the identity matrix, the value of λ will be below 1. The experiment compares the change in

performance by changing two parameters, λ and ε. The values run over $\{0.5, 0.1, 0.01\}$ and $\{0.3, 0.1, 0.03, 0.01, 0.001\}$, respectively, testing with every combination.

4.1 Results

All cases consists of 15 graphs, each having different λ and ε values listed above. Each cell has 1, 2, and 3-point multi-point optimization, which the graph will be represented as dotted, solid, and dashed line, respectively. In each cell, the horizontal axis represents the time step, and the vertical axis represents the average cumulative regret. As a whole, along the horizontal direction, λ changes in a descending order from the left and along the vertical direction, ε changes in a descending order from the top.

4.1.1 Case 1

This case treats each point equally thus for 2-point and 3-point algorithms, to step number is decreased to 100 and 67, respectively, such that the total number of points searched would be around 200.

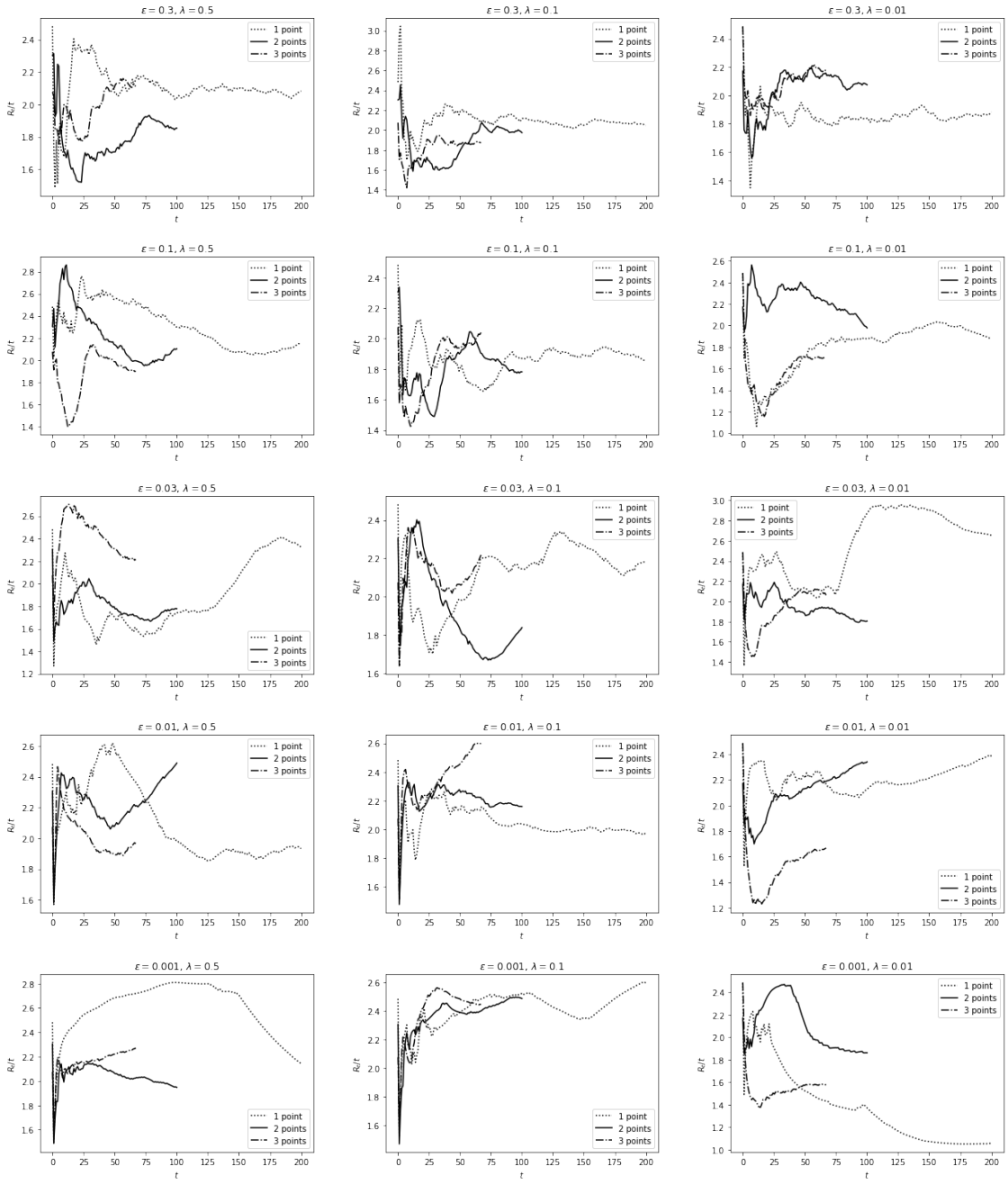

Figure 4.1: Performance of the MPTVGPUCB algorithm case 1 setting

When $\lambda = 0.5$, which is the leftmost column, the multi-point optimization performed the best on most of the experiments, and when $\lambda = 0.01$, the single-point optimization performed the best on most of the experiments. The middle column have mixed results, some experiments resulting in the single-point to perform better, while others resulting in multi-point to perform better, and some ending in similar regrets. For ε, the correlation between ε and the regret seems to not exist however, there is a general trend that can be said about the setting. As ε decreases, the algorithm seems to prefer the multi-point optimization

since they perform better as the value decreases. From this experiment, we can conclude that λ has a stronger effect on the result than ε since each column has a similar behavior but varies largely among each row. For the 4th row, the result is peculiar to the overall trend of the experiment. Although multi-point optimization performs better as ε decreases, when $\varepsilon = 0.01$, the single-point optimization seemed to perform slightly better.

4.1.2 Case 2

The algorithm is the same as case 1, but the only the best value is added to the regret; only one point contributes to the regret at each time step. The graph for one-point optimization should not change from case 1.

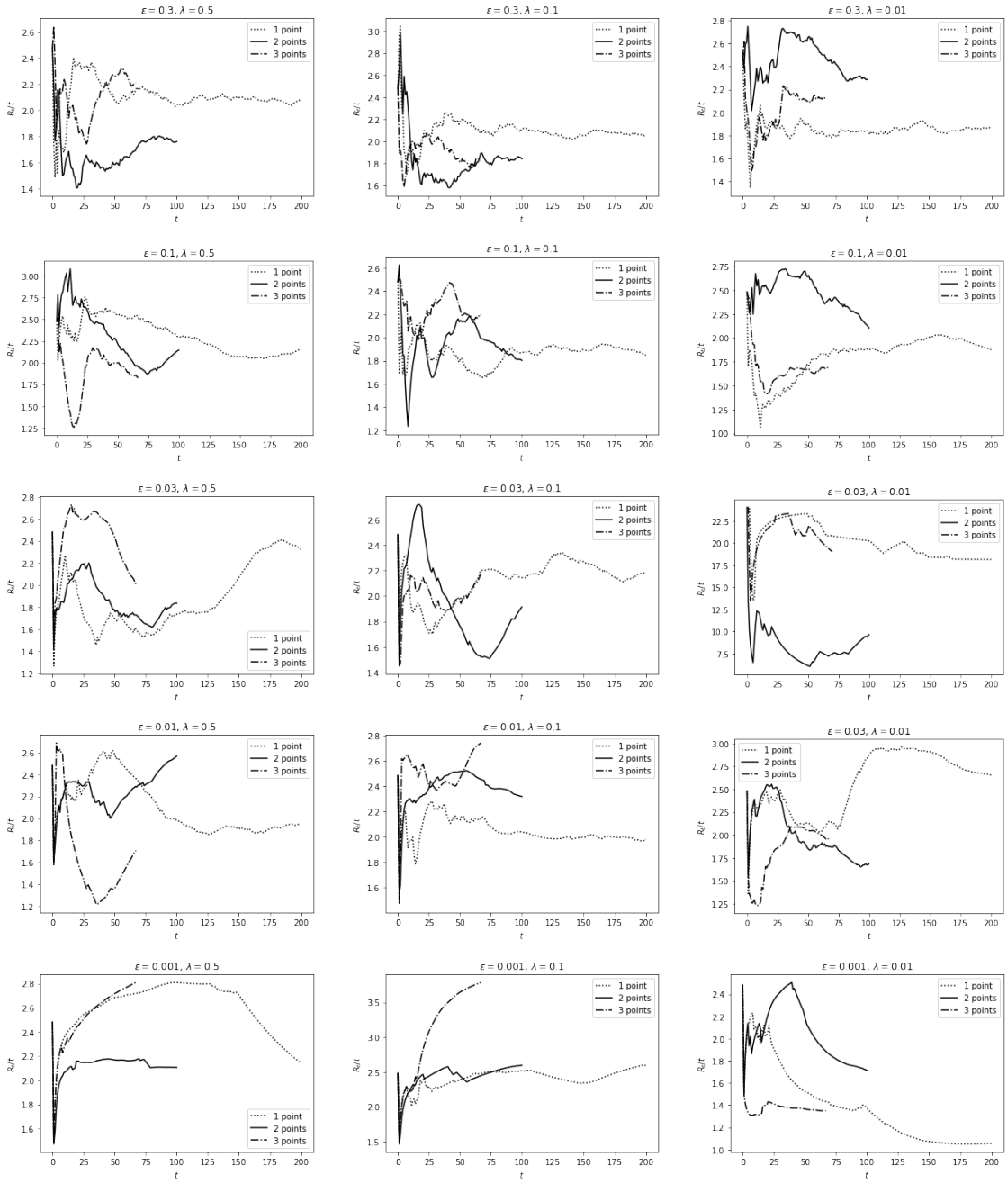

Figure 4.2: Performance of the MPTVGPUCB algorithm with case 2 setting

The overall behavior of case 2 is similar to that of case 1; the leftmost column performed the best on multi-point optimization and when ε has a low value, it favors multi-point optimization. One of the changes from case 1 is that some of the regret increased; for example, 3-point optimization at $\lambda = 0.5, \varepsilon = 0.001$ i.e. the bottom left cell has an increased regret compared to case 1.

4.1.3 Case 3

In this case, the number of time steps are treated equally, and all multi-point scenario takes 200 points. In other words, case 3 is a continuation from case 1 until all settings reach 200 time steps.

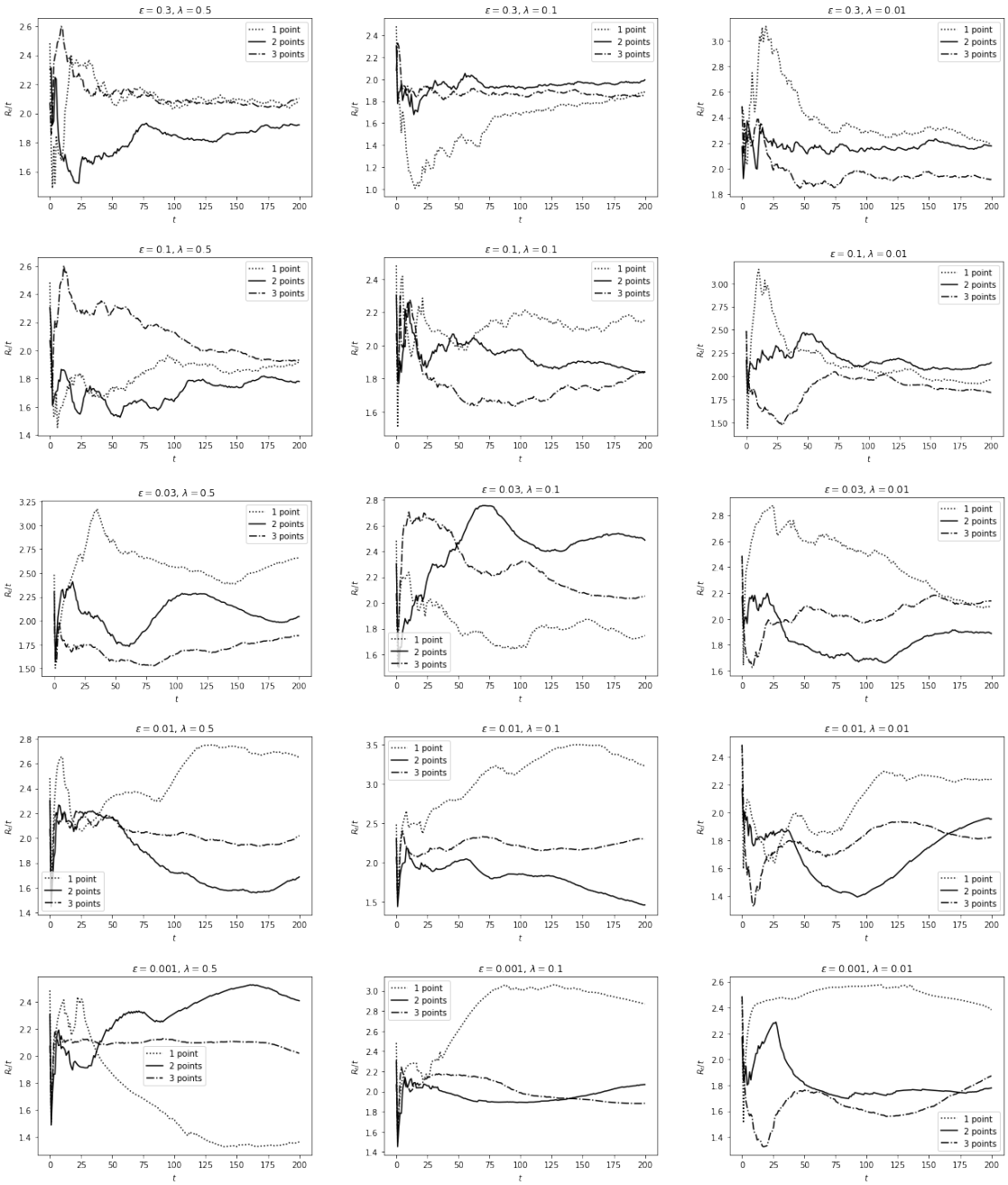

Figure 4.3: Performance of the MPTVGPUCB algorithm with case 3 setting

In this setting, most of the experiment have the multi-point optimization performed better than single-point optimization. When λ is small, the change in regret is gradual whereas

when λ is large, the regret can change suddenly. Regardless of the smoothness of the regret, it is shown that multi-point optimization performs better than single-point optimization. Although multi-point optimization outperforms single-point optimization, their values seem to close when $\varepsilon >= 0.01$; below this point, there seems to be a large gap between multi-point and single-point.

4.1.4 Case 4

In this case, all scenarios take 200 time steps and only the top position contributes to the regret; all other points are treated as auxiliary points.

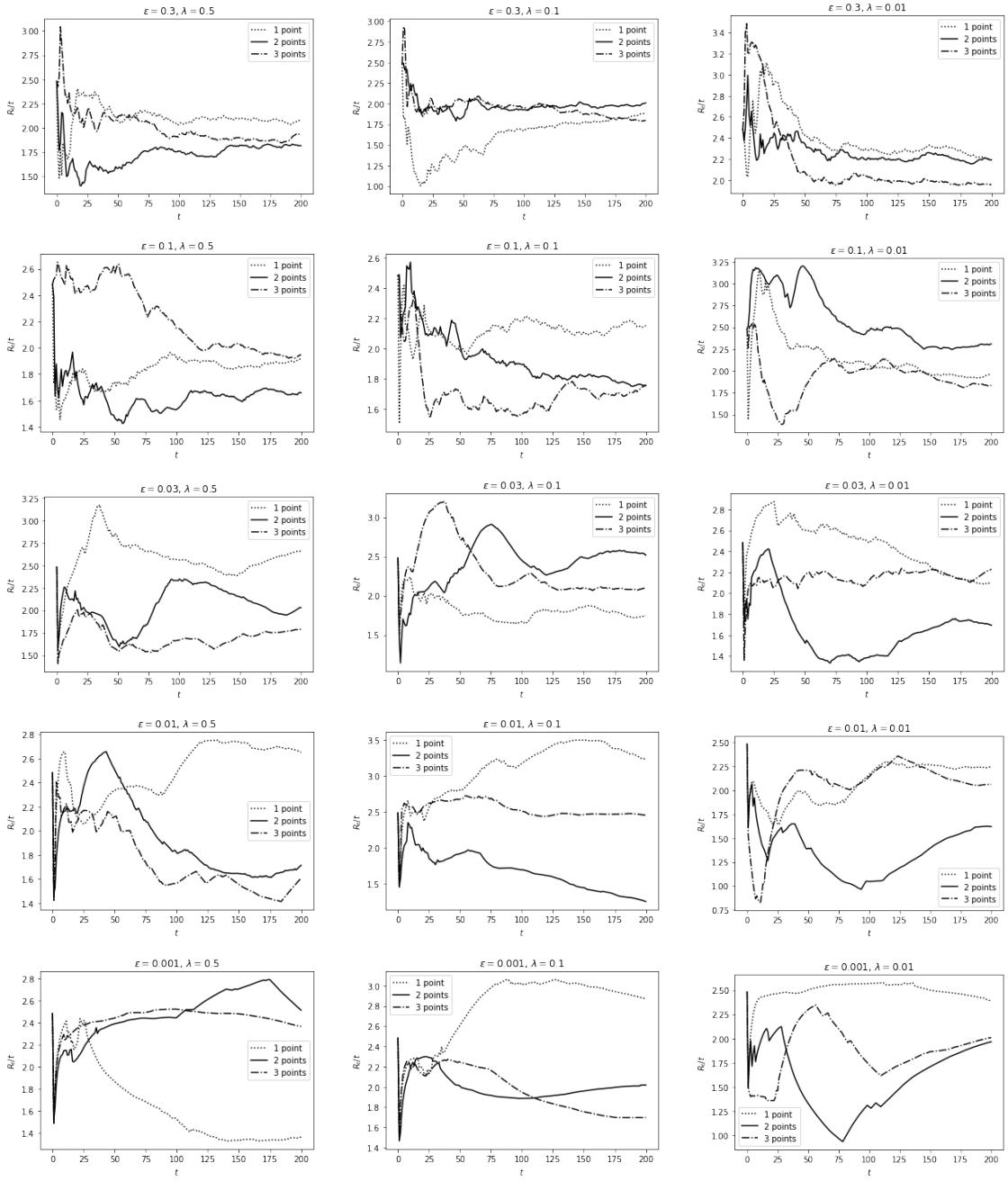

Figure 4.4: Performance of the MPTVGPUCB algorithm with case 4 setting

The overall shape of the graph are similar to case 3. One of the differences from case 3 is that most of the graph has a bumpy curve instead of them being gradual and smooth. Another difference is that the value of the regrets has changed, some changing slightly yet some others changing drastically. Most of the drastic changes comes from 3-point optimization.

5. Conclusion

We proposed an extension to the TV-GP-UCB algorithm which queries multiple points at each time step. Although multi-point optimization did perform better than single-point optimization, our hypothesis, which was that experiments with higher ε can also perform better, was false. As more time steps were taken at 2-point and 3-point optimization, some regrets went below single-point optimization and performed better. As a result, our hypothesis were shown false; the algorithm performs better but at larger ε, there is little advantage to it and the algorithm performs better when λ is set at larger points. Regardless of taking the average each point or to exclude lower confidence bounds, it had a positive effect on the regret. As mutli-point optimization performs better, 2-point optimization seemed to perform better than 3-point optimization. Since $\lambda = 0.5$ had performed the best within all columns, it shows that more exploitation could cause better performance for larger ε.

An immediate future work from this paper is the theoretical analysis of the regret bounds and complexity of the algorithm. Another further research can be done on the environment of the experiment, such as different kernels such as Matern kernel, input space with higher dimensions, testing the robustness of the algorithm, as well as testing with real life data. Expanding on the other algorithm proposed in Bogunovic's paper is a considerable candidate as well.

Bibliography

[1] I. Bogunovic, J. Scarlett, and V. Cevher, “Time-Varying Gaussian Process Bandit Optimization”, Preprint 1601.06650, arXiv, 2016

[2] E. Brochu, V. M. Cora and N. de Freitas, “A Tutorial on Bayesian Optimization of Expensive Cost Functions, with Application to Active User Modeling and Hierarchical Reinforcement Learning”, Preprint 1012.2599, arXiv, 2010

[3] P. I. Frazier, “A Tutorial on Bayesian Optimization”, Preprint 1807.02811, arXiv, 2018

[4] H. Imamura, N. Charoenphakdee, F. Futami, I. Sato, J. Honda, M. Sugiyama, “Time-varying Gaussian Process Bandit Optimization with Non-constant Evaluation Time”, Preprint 2003.04691, arXiv, 2020

[5] C. E. Rasmussen and C. K. I. Williams, “Gaussian Process for Machine Learning”, MIT Press, 2006

[6] N. Srinivas, A. Krause, S. M. Kakade, and M. Seeger, “Gaussian Process Optimization in the Bandit Setting: No Regret and Experimental Design”, Preprint 0912.3995v4, arXiv, 2010

[7] 本多淳也，中村篤洋．バンディット問題の理論とアルゴリズム．機械学習プロフェッショナルシリーズ．講談社．2019

[8] 持橋大地, 大羽成征．ガウス過程と機械学習．機械学習プロフェッショナルシリーズ．講談社．2020

A. Appendix

Here, the source code for the algorithm is given.

```
#!/usr/bin/env python
# coding: utf-8

import numpy as np
import math
import matplotlib.pyplot as plt

np.random.seed(0)

# GP Regression
def GPR(xtest, xtrain, ytrain, kernel, TV=False, *param):
    n0 = len(ytrain)
    N = len(xtest)
    noise, epsilon = param

    K = np.zeros([n0, n0])
    for j, y in enumerate(xtrain):
        for i, x in enumerate(xtrain):
            K[i][j] = kernel(x, y)
    K = K + noise * noise * np.eye(n0)

    k_star = np.zeros([n0, N])
    for j, y in enumerate(xtest):
        for i, x in enumerate(xtrain):
            k_star[i][j] = kernel(x, y)

    k_starstar = np.zeros([N, N])
    for j, y in enumerate(xtest):
        for i, x in enumerate(xtest):
            k_starstar[i][j] = kernel(x, y)

    # Calculation of time-varying kernel
    if TV:
```

```
            D = np.zeros([n0, n0])
            d_star = np.zeros([n0, N])

            for i in range(n0):
                for j in range(n0):
                    D[i][j] = (1 - epsilon) ** (np.abs((i - j) / 2))
            K = K * D # element-wise multiplication

            for i in range(n0):
                for j in range(N):
                    d_star[i][j] = (1 - epsilon) ** ((n0 - i) / 2)
            k_star = k_star * d_star

        Kinv = np.linalg.inv(K)
        mu = np.matmul(k_star.T, np.matmul(Kinv, ytrain))
        var = k_starstar - np.matmul(k_star.T, np.matmul(Kinv, k_star))

        return np.array(mu), np.array(var)


# calculating p-1 points to query
def ucblist(grid, kernel, ucb, weight, n):
    bestidxs = []
    for i in range(n):
        sumgrid = []
        for z in grid:
            sum = 0
            for j in range(i):
                sum += kernel(z, grid[bestidxs[j]])
            sumgrid.append(sum)
        sumgrid = [weight * s for s in sumgrid]

        bestidx = np.argmax([u - s for u, s in zip(ucb, sumgrid)])
        ucb[bestidx] = -100
        bestidxs.append(bestidx)

    return bestidxs

# TV-GP-UCB algorithm
def TVGPUCB(grid, kernel, f, N, samplepts, noise, lamda, epsilon, singret=False):
    gridsize = len(grid)
    regrets = []
    regret = 0
```

```
    f_star = f
    sol = []

    mu = np.zeros(grid.shape[0])
    var = np.eye(grid.shape[0])

    xtrain = []
    ytrain = []
    for iter in range(1, N+1):
        alpha = np.sqrt(0.8 * np.log(4 * iter))
        ucb = mu + alpha * np.sqrt(np.diag(var))
        bestidxs = ucblist(grid, kernel, ucb, lamda, samplepts)
        bestpos = [grid[i] for i in bestidxs]

        xtrain.extend(bestpos)
        ytrain.extend([f_star[i] + np.random.normal(0, noise) for i in bestidxs])
        mu, var = GPR(grid, np.array(xtrain), np.array(ytrain), kernel, True, noise,
            epsilon)

        sol.append(f_star.tolist())

        # branch for single regret
        if singret:
            regret += np.max(f_star) - f_star[math.floor(bestpos[0] * gridsize) - 1]
        else:
            regret += (len(bestpos) * np.max(f_star) - np.sum([f_star[math.floor(i *
                gridsize) - 1] for i in bestpos])) / len(bestpos)
        regrets.append(regret / iter)

        f_star = np.sqrt(1 - epsilon) * f_star + np.sqrt(epsilon) * np.random.
            multivariate_normal(np.zeros(gridsize), np.eye(gridsize))

    return regrets


# f_1
def f_random(gridsize):
    return np.random.multivariate_normal(np.zeros(gridsize), np.eye(gridsize))


# SE kernel
def gauss_kern(x, y, gamma: float = 800):
    return np.exp(-gamma * np.linalg.norm(x - y)**2)
```

```
# MPTVGPUCB env setup
def TVGPUCBtest(case):
    gridsize:int = 50
    noise: float = 0.01
    epsilon: float = 0.1
    X_star = np.linspace(0, 1, num=gridsize)
    f_init = f_random(gridsize)

    def plot(l, e):
        if case == 1 or case == 2:
            tot1, pt1 = (200, 1)
            tot2, pt2 = (100, 2)
            tot3, pt3 = (67, 3)
        elif case == 3 or case == 4:
            tot1, pt1 = (200, 1)
            tot2, pt2 = (200, 2)
            tot3, pt3 = (200, 3)

        if case == 1 or case == 3:
            regret1 = TVGPUCB(X_star, gauss_kern, f_init, tot1, pt1, noise, l, e)
            regret2 = TVGPUCB(X_star, gauss_kern, f_init, tot2, pt2, noise, l, e)
            regret3 = TVGPUCB(X_star, gauss_kern, f_init, tot3, pt3, noise, l, e)
        elif case == 2 or case == 4:
            regret1 = TVGPUCB(X_star, gauss_kern, f_init, tot1, pt1, noise, l, e, True)
            regret2 = TVGPUCB(X_star, gauss_kern, f_init, tot2, pt2, noise, l, e, True)
            regret3 = TVGPUCB(X_star, gauss_kern, f_init, tot3, pt3, noise, l, e, True)

        print(f"plot␣l={l},␣e={e}")
        plt.xlabel(r"$t$")
        plt.ylabel(r"$R_t␣/␣t$")
        plt.title(r"$\varepsilon␣=␣$" + f"{e},␣" + r"$\lambda␣=␣$" + f"{l}")
        plt.plot(np.linspace(0, tot1, num=tot1), regret1, color="black", linestyle="dotted"
            , label="1␣point")
        plt.plot(np.linspace(0, tot2, num=tot2), regret2, color="black", linestyle="solid",
             label="2␣points")
        plt.plot(np.linspace(0, tot3, num=tot3), regret3, color="black", linestyle="dashdot
            ", label="3␣points")
        plt.legend()
        plt.show()

    for l in [0.5, 0.1, 0.01]:
        for e in [0.3, 0.1, 0.03, 0.01, 0.001]:
            plot(l, e)
```

```
def main():
    TVGPUCBtest(1) # case 1
    TVGPUCBtest(2) # case 2
    TVGPUCBtest(3) # case 3
    TVGPUCBtest(4) # case 4

if __name__ == "__main__":
    main()
```

和文抄訳

大域的最適化とは目的関数と呼ばれる関数全範囲における最大値もしくは最小値を求める問題である。理想的な状況では関数の凸性や勾配なども知ることができるが、実際の実験などでは凸性や勾配はおろか、関数値の獲得にコストがかかる場合もある。ベイズ最適化はそのような制約がある目的関数において大域的最適解を求める手法の一つである。

ベイズ最適化はブラックボックスと呼ばれる関数を逐次的に最適化する手法である。すなわち、アルゴリズムの各ステップにおいて関数の点を探り、それをもとにより最適解に近づく手法である。関数がブラックボックスであるとは関数値を観測する際に得られる情報が限られていることを指す。一般的な最適化では観測した際、関数値だけでなく、勾配や凸性も得られるが、ブラックボックス関数は関数値のみが帰ってくる。したがって、ベイズ最適化は目的関数を直接最適化するのではなく、得られた情報をもとに構築したモデルを用いて最適解を求める。

ベイズ最適化の手順は大まかに 2 つに分けられている。一つは代理モデルの構築である。この代理モデルは統計的手法を用いて目的関数の事前分布を計算する。この事前分布はガウス分布に従うと仮定する。もう一つは損失関数の構築である。損失関数を用いて次のステップにおいて最適なクエリを求める。多くのベイズ最適化はバンディット問題の手法を用いており、損失関数はリグレットで表されることもある。バンディットとはベイズ最適化の離散バージョンと捉えることができ、限られた選択数で与えられた選択肢とそれに対応する報酬を最大化する決定問題である。代理モデルと損失関数は各ステップにおいて相互に働きかける。損失関数を用いて点と関数値を準備し、それと既に得られた情報で事前分布を構築する、ということを繰り返すことで目的関数の最大値を求めている。理論的には関数値のみの情報しか得られないとし

ているが、実際は凸性や連続性などいくつか目的関数の仮定をしアルゴリズムを構築している。

目的関数は位置についてだけではなく時間について依存させることもできる。時間依存の目的関数というのは $f_t(x), t \in \{1, 2, \cdots, \}$ という形で表せる。これは時間 t、位置 x におけるそれぞれの時間ステップごとに変化する関数列とみなすこともできる。時間変化ありのアルゴリズムは時間変化なしのアルゴリズムとあまり変わらないが、関数が時間に依存するため、代理モデルの構築において古い点の重要度を下げるような工夫が必要である。詳しいアルゴリズムは 3.3 節にある。既存の時間依存のベイズ最適化の問題点として、時間変化率が小さいことが条件にある。本論文ではそこを解消できることを試みてアルゴリズムを拡張した。

本論文は各時間において 1 点のみ探索するのではなく複数点探索することによって関数の急な変化に対応できると予想した。実験は四種類の設定があり、それぞれに一点、二点、三点探索を 2 つのパラメータを変化させて実験したパラメータの一つは本論文で導入した新しいパラメータでもう一つのパラメータに依存しない想定である。設定は全点がリグレットに影響する場合、最良点のみが影響する場合の二種類と探索点をを一定にする場合と時間ステップを一定にする場合の二種類の組み合わせで行った。実験データはガウス過程に従う分布の合成データを用いた。実験の結果、複数点観察した実験のほうがリグレットが少なく済んだが、予想とは違い時間変化が大きい関数では単一点のみの場合と大差がなかった。しかし、大きい変化でも複数点探索がよりよい結果だったため、広い探索を行うことも可能であることが示された。