Speeding up the Quantification of Data Staleness in Dynamic Bayesian Optimization

Giovanni Ranieri
Lausanne, Switzerland
`giovanni.ranieri@epfl.ch`

School of Computer and Communication Sciences

Swiss Federal Institute of Technology in Lausanne

Bachelor Project

June 2024

Responsible
Prof. Patrick Thiran
EPFL / INDY

Supervisor
Anthony Bardou
EPFL / INDY

Abstract

Some algorithms have been proposed to extend Bayesian Optimization (BO) to time-varying functions. This adaptation is known as Dynamic Bayesian Optimization (DBO). To achieve great performance, W-DBO from [3] uses a criterion that quantifies how relevant an observation is for the future predictions of the Gaussian Process. By evolving in time, the optimum of the function changes, and observations of the function become less relevant due to their lack of information on its future values. In a long-period time optimization, keeping all of them will impact the performance of the algorithm. In fact, the sampling frequency will decrease due to the growth of the Gaussian Process' inference time. To remove them rapidly, the criterion should be calculated efficiently in a low-level programming language. By speeding up the computations using C++ to calculate the criterion, W-DBO shown great improvements over state of the art solutions. We present in this work (i) the C++ implementation, (ii) the performance of this implementation compared to a Python implementation, (iii) the performance of W-DBO compared to the state of the art. Additionally, we develop Python packages for W-DBO and the criterion.

1 Introduction

Black-box optimization algorithms aim to optimize an objective function $f : \mathcal{S} \to \mathbb{R}$ without any knowledge of its functional form. Challenges of this nature cannot be tackled using methods like gradient descent, as they rely on first-order derivatives. Such challenges arise for example in networking [2] and robotics [11], and can be addressed by a framework known as Bayesian Optimization (BO). The BO framework exploits a Gaussian Process (GP) as a surrogate model to encode beliefs about how the objective function should behave.

The extension of BO to time-varying functions is known as Dynamic Bayesian Optimization (DBO). By evolving in time, observations of f become less relevant in time to optimize it. If we denote n the number of observations at time t, the inference time of the GP is $\mathcal{O}(n^3)$. In the context of online optimization [6], we need to sample as fast as possible to keep track accurately of the optimum. Removing stale data becomes necessary to optimize f continuously and rapidly. Having too much observations will impact the performance of the algorithm because the sampling frequency will decrease (i.e. f will evolve faster than we can keep track of the optimum because of the inference time).

The contributions described here are exploited in [3] which aims at addressing data staleness in DBO. To do so, [3] uses a criterion that estimates how much an observation is relevant to optimize f in the future. It presents an algorithm (W-DBO) which outperforms state of the art (SOTA) solutions using this criterion implemented in C++. This criterion has the drawback of being time-consuming. Implementing the heavy computations in a low-level programming language becomes necessary for a DBO algorithm, since its performance heavily depends on its computation time. In fact, waiting too long before querying the next point would reduce the overall efficiency of the optimization as the maximum of f would change. To compute in short terms this criterion for a better optimization, our contributions include: (i) the C++ implementation of the criterion, (ii) the comparison of a C++ implementation for the criterion with a Python implementation, (iii) the implementation of the SOTA solutions to compare the new performance of W-DBO and (iv) the development of Python packages for W-DBO (`wdbo_algo`) and the criterion (`wdbo_criterion`) [1].

2 Background

2.1 Bayesian Optimization

BO, as introduced by [7], is an approach to solve black-box optimization problems. BO assumes that f is a GP $\mathcal{GP}(0, k)$, $\boldsymbol{x}$,$\boldsymbol{x}' \in \mathcal{S}$, where k is a kernel function that computes the covariance between function values. BO takes advantage of a GP because it provides a good best guess estimation with uncertainty for the next predictions [15]. By conditioning the GP with observations, we obtain a posterior distribution for future values of f. If we denote $\mathcal{D} = \{(\boldsymbol{x}_i, y_i)\}_{1 \leq i \leq n}$ the set of observations

[1] https://github.com/WDBO-ALGORITHM

where $y_i = f(\boldsymbol{x}_i) + \epsilon$, $\epsilon \sim \mathcal{N}(0, \sigma_0^2)$, then for any $\boldsymbol{x} \in \mathcal{S}$

$$f(\boldsymbol{x})|\mathcal{D} \sim \mathcal{GP}(\mu(\boldsymbol{x}), \sigma^2(\boldsymbol{x})) \tag{1}$$

where the posterior mean and variance are

$$\mu(\boldsymbol{x}) = \boldsymbol{k}^\top(\boldsymbol{x}, \boldsymbol{X})\boldsymbol{\Delta}^{-1}\boldsymbol{y}, \tag{2}$$
$$\sigma^2(\boldsymbol{x}) = k(\boldsymbol{x}, \boldsymbol{x}) - \boldsymbol{k}^\top(\boldsymbol{x}, \boldsymbol{X})\boldsymbol{\Delta}^{-1}\boldsymbol{k}(\boldsymbol{x}, \boldsymbol{X}) \tag{3}$$

with $\boldsymbol{X} = (\boldsymbol{x}_1, \cdots, \boldsymbol{x}_n)$, $\boldsymbol{y} = (y_1, \cdots, y_n)$, and

$$\boldsymbol{\Delta} = \boldsymbol{k}(\boldsymbol{X}, \boldsymbol{X}) + \sigma_0^2\boldsymbol{I}, \tag{4}$$
$$\boldsymbol{k}(\mathcal{X}, \mathcal{Y}) = (k(\boldsymbol{x}_i, \boldsymbol{x}_j))_{\substack{\boldsymbol{x}_i \in \mathcal{X} \\ \boldsymbol{x}_j \in \mathcal{Y}}}. \tag{5}$$

2.2 Dynamic Bayesian Optimization

DBO is an extension of BO for time-evolving functions. We denote the new domain of f by $\mathcal{S} \times \mathcal{T}$ (where $\mathcal{T} \subset \mathbb{R}$ is the time dimension) and the dataset of observations $\mathcal{D} = \{(\boldsymbol{x}_i, t_i, y_i)\}_{1 \leq i \leq n}$, where t_i is the time at which $\boldsymbol{x}_i$ has been queried. In this framework, time must not be interpreted in the same way as yet another spatial dimension. By querying f at time t_0, the point queried is necessarily of the form $(\boldsymbol{x}, t_0) \in \mathcal{S} \times \mathcal{T}$. Additionally, since the function evolves in time (changing of optimum), points sampled in the past become less relevant to optimize it in the future (problem called *data staleness*). To optimize continuously f in an online setting, the sampling frequency needs also to be high. Removing stale data is necessary to avoid the BO inference ($\mathcal{O}(n^3)$, n being the size of the dataset of observations) to become asymptotically prohibitive, impacting the overall performance of the algorithms.

In the literature, DBO solutions fall into two categories. The first one removes stale data by resetting $\mathcal{D}$ periodically [4] or when an event is triggered [5]. The second approach (also in [4]) separates the kernel into two kernels, one for the spatial dimensions $k_S(\boldsymbol{x}_i, \boldsymbol{x}_j)$, $\boldsymbol{x}_i, \boldsymbol{x}_j \in \mathcal{S}$, and one for the time dimension $k_T(t_i, t_j)$, $t_i, t_j \in \mathcal{T}$, ideally to describe complex spatial and temporal dynamics. We discuss the cons of these solutions in Section (4).

2.3 W-DBO

The approach in [3] is different as it uses a criterion that addresses data staleness by **quantifying** the importance of an observation on the future predictions. We describe the criterion before diving into its C++ implementation. To model the function f, the kernel function takes two points $(\boldsymbol{x}, t), (\boldsymbol{x}', t') \in \mathcal{S} \times \mathcal{T}$ and has the form

$$k((\boldsymbol{x}, t), (\boldsymbol{x}', t')) = \lambda k_S(||\boldsymbol{x} - \boldsymbol{x}'||_2, l_S)k_T(|t - t'|, l_T) \tag{6}$$

where $\lambda > 0$, k_S and k_T are isotropic kernel functions for space and time with lengthscales $l_S > 0$ and $l_T > 0$, respectively (for more details on lengthscales and kernel functions, please refer to [15]).

Let us denote GPs conditioned on $\mathcal{D}$ and $\tilde{\mathcal{D}} = \mathcal{D} \setminus \{(\boldsymbol{x}_i, t_i, y_i)\}$, $(\boldsymbol{x}_i, t_i, y_i) \in \mathcal{D}$, with $\mathcal{GP}_\mathcal{D}$ and $\mathcal{GP}_{\tilde{\mathcal{D}}}$. If this removed observation was not relevant, both processes would be very similar. W-DBO proposes to quantify this relevancy by using the Wasserstein distance $W_2(\mathcal{GP}_\mathcal{D}, \mathcal{GP}_{\tilde{\mathcal{D}}})$ between the two GPs. The criterion is defined as follows

$$R(\mathcal{GP}_\mathcal{D}, \mathcal{GP}_{\tilde{\mathcal{D}}}) = \frac{W_2(\mathcal{GP}_\mathcal{D}, \mathcal{GP}_{\tilde{\mathcal{D}}})}{W_2(\mathcal{GP}_\mathcal{D}, \mathcal{GP}_\emptyset)} \tag{7}$$

where $\mathcal{GP}_\emptyset$ is the prior distribution. Compared to SOTA solutions, W-DBO do not naively remove all observations from $\mathcal{D}$, which could erase relevant observations. Instead, it computes upper bounds on $W_2(\mathcal{GP}_\mathcal{D}, \mathcal{GP}_{\tilde{\mathcal{D}}})$ and $W_2(\mathcal{GP}_\mathcal{D}, \mathcal{GP}_\emptyset)$ to approximate (7). This approximation is necessary because (i) no closed forms for the Wasserstein distance are provided by [3] and (ii) we need to compute the criterion rapidly, and numerical integration for high accuracy would be too long [12]. The observation is removed if the approximation of $R(\mathcal{GP}_\mathcal{D}, \mathcal{GP}_{\tilde{\mathcal{D}}})$ is small enough [3].

3 C++ Implementation

In this section, we describe the C++ implementation of the criterion in W-DBO (see lines 10-13 of Algorithm 1 in [3]). W-DBO was originally fully implemented in Python. To implement the heavy computations of the criterion in C++, we choose Eigen as linear algebra library [8] because it is a well-known and good documented library. The rest of W-DBO remains in Python and we use a Python package called Pybind11 [9] to call C++ methods in Python. The C++ incorporated in W-DBO is illustrated in Figure (1). After each query of W-DBO, for each observation $(\boldsymbol{x}_i, t_i, y_i) \in \mathcal{D}$, we compute the approximation of (7) denoted by

$$\hat{R}_i(\mathcal{GP}_{\mathcal{D}}, \mathcal{GP}_{\tilde{\mathcal{D}}}) \tag{8}$$

Please refer to [3] for a detailed close form. In the following subsections, we present the challenges we encountered and the solutions we proposed for them.

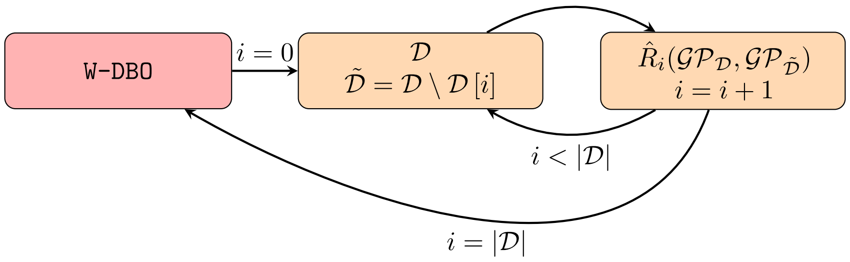
W-DBO
$i = 0$
$\mathcal{D}$
$\tilde{\mathcal{D}} = \mathcal{D} \setminus \mathcal{D}[i]$
$\hat{R}_i(\mathcal{GP}_{\mathcal{D}}, \mathcal{GP}_{\tilde{\mathcal{D}}})$
$i = i + 1$
$i < |\mathcal{D}|$
$i = |\mathcal{D}|$

Figure 1: Simplified schema of W-DBO. The Python part of W-DBO is represented by the red state. It queries a new point $(\boldsymbol{x}, t) \in \mathcal{S} \times \mathcal{T}$, observes it $y = f(\boldsymbol{x}) + \epsilon$, $\epsilon \sim \mathcal{N}(0, \sigma_0^2)$ and adds it into $\mathcal{D}$. The orange states show the C++ implementation. For all $1 \leq i \leq |\mathcal{D}|$, we compute $\hat{R}_i(\mathcal{GP}_{\mathcal{D}}, \mathcal{GP}_{\tilde{\mathcal{D}}})$. When it is done, the remainder of W-DBO takes place in the red state until the next query (and we remove possibly some observations from $\mathcal{D}$ due to their irrelevancy found by the criterion).

3.1 Matrix Inversion

Computing (8) requires many matrix inversions. Inverting a matrix $\boldsymbol{A} \in \mathbb{R}^{n \times n}$ has by default a time complexity of $\mathcal{O}(n^3)$. Achieving accuracy in numerical computation of the inverse is challenging. This accuracy, related to the condition number of $\boldsymbol{A}$ [1], can be improved using the right algorithm depending on the properties of $\boldsymbol{A}$. A better way to solve this problem is to not compute directly $\boldsymbol{A}^{-1}$. Instead, the idea is to solve linear systems involving $\boldsymbol{A}$ that can be easily solved, for example using the right matrix decomposition. Suppose you want to compute

$$\boldsymbol{x} = \boldsymbol{A}^{-1}\boldsymbol{b} \tag{9}$$

with $\boldsymbol{b} \in \mathbb{R}^n$ by inverting $\boldsymbol{A}$ and then multiply it by $\boldsymbol{b}$. Note that this equation is equivalent to solve a linear system of equations

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b} \tag{10}$$

and LU decomposition can be used to solve the system [1]. More specifically, if $\boldsymbol{A} = \boldsymbol{L}\boldsymbol{U}$, then

$$\boldsymbol{x} = \boldsymbol{U}^{-1}\boldsymbol{L}^{-1}\boldsymbol{b} \tag{11}$$

where the inverses of $\boldsymbol{U}$ and $\boldsymbol{L}$ can be computed efficiently and accurately[2][13]. The following paragraphs describe how we used matrix decomposition to compute quantities involving inverse of matrices.

[2]Note that having a lot of these systems to solve with the same $\boldsymbol{A}$ (which is not our case) then computing directly $\boldsymbol{A}^{-1}$ could be more efficient [1].

3.1.1 Cholesky Decomposition

Suppose you have a matrix $\boldsymbol{M} \in \mathbb{R}^{n \times n}$ that is positive definite (PD). Then, the Cholesky decomposition

$$\boldsymbol{M} = \boldsymbol{L}\boldsymbol{L}^T, \tag{12}$$

where $\boldsymbol{L}$ is a lower triangular matrix, exists and is unique [13]. It allows to compute the following form without computing explicitly $\boldsymbol{M}^{-1}$

$$\boldsymbol{x}_1 = \boldsymbol{b}^T \boldsymbol{M}^{-1} \boldsymbol{b} \tag{13}$$

with $\boldsymbol{b} \in \mathbb{R}^n$. For Equation (13), $\boldsymbol{L}$ from the Cholesky decomposition is used to solve the system

$$\boldsymbol{L}\boldsymbol{y} = \boldsymbol{b} \Rightarrow \boldsymbol{y} = \boldsymbol{L}^{-1}\boldsymbol{b}. \tag{14}$$

To compute $\boldsymbol{x}_1$, we do

$$\boldsymbol{x}_1 = \boldsymbol{y}^T \boldsymbol{y} = \boldsymbol{b}^T \underbrace{(\boldsymbol{L}^{-1})^T \boldsymbol{L}^{-1}}_{\boldsymbol{M}^{-1}} \boldsymbol{b}. \tag{15}$$

Equation (14), using backsubstitution [1], is easy to solve with $\boldsymbol{L}$ a triangular matrix. In our case, although $\boldsymbol{M}$ is PD, we use another decomposition called LDLT to avoid numerical instability when inverting $\boldsymbol{L}$.

3.1.2 QR decomposition

The QR factorization decomposes the matrix $\boldsymbol{M} \in \mathbb{R}^{n \times m}$ in

$$\boldsymbol{M} = \boldsymbol{Q}\boldsymbol{R} \tag{16}$$

where $\boldsymbol{Q}$ is an orthogonal matrix ($\boldsymbol{Q}\boldsymbol{Q}^T = \boldsymbol{I} \Rightarrow \boldsymbol{Q}^T = \boldsymbol{Q}^{-1}$) and $\boldsymbol{R}$ an upper triangular matrix [13]. Any matrix $\boldsymbol{M}$ admits a QR decomposition. It helps to solve the linear system

$$\boldsymbol{M}\boldsymbol{x} = \boldsymbol{b} \tag{17}$$

with $\boldsymbol{b} \in \mathbb{R}^n$. It implies that

$$\boldsymbol{Q}\boldsymbol{R}\boldsymbol{x} = \boldsymbol{b} \Rightarrow \boldsymbol{R}\boldsymbol{x} = \boldsymbol{Q}^T\boldsymbol{b}. \tag{18}$$

Similarly to (14), we can use backsubstitution to solve the system (18). Even if QR decomposition involves about twice as many operations as LU decomposition [13], Eigen's documentation recommends to use it for systems with n and m bigger than 10, which is our case.

3.2 Bindings

To retrieve the values of (8) for each observation in $\mathcal{D}$ at time t, we call a C++ method a single time. We provide $\mathcal{D}$ to Eigen by splitting it into a matrix of points $\boldsymbol{P} = (\boldsymbol{x}_1, \cdots, \boldsymbol{x}_n)$, a vector of function values $\boldsymbol{y} = (y_1, \cdots, y_n)$ and a vector $\boldsymbol{t} = (t_1, \cdots, t_n)$. The method returns an array where each element is (8), for all $1 \leq i \leq n$.

Since we don't know the size of $\mathcal{D}$ at compile time, we use dynamic-sized matrices, which are dynamic allocated arrays. To avoid copies of big data structures when calling the method, we rely on the documentation of Pybind11. When returning an ordinary vector to Numpy, Pybind11 saves the vector and returns a Numpy array that directly references the Eigen vector: no copy of the data is performed. Passing Numpy arrays or matrices to Eigen is more complex when considering the performance. Eigen provides a wrapper `Eigen::Ref<MatrixType>` to avoid the copy of data structures, but a problem of storage order comes. Numpy and Eigen use row-major (the data is stored row by row) and column-major (the data is stored column by column) storage, respectively. For matrices, we solve this problem by imposing row-major storage to $\mathcal{D}$ in Eigen. The main advantage of that is to avoid non-contiguous storage along the second dimension.

3.3 Auto-differentiation

The purpose of auto-differentiation (AD) is to differentiate a function g. This calculates the derivatives precisely. In contrast, numerical differentiation only approximates the derivatives [14]. Nowadays, AD is used in Machine Learning for first-order derivatives (e.g., to implement the backprogagation algorithm of neural networks). In W-DBO, we need to compute multiple times nth-order derivatives, $1 \leq n \leq 7$, of the function

$$P(t) = t^a (t+p)^b \tag{19}$$

with respect to t, where a, $b \in \mathbb{N}$ and $p \in \mathbb{R}$. AD frameworks for arbitrary functions exist but are very slow. Instead, we create a specific AD framework to focus on what is really needed. It provides accurate results with a simple and fast implementation.

Our AD framework is developed for products and sums of monomes $(t+p)^b$ with constants $c \in \mathbb{R}$ (note that t^a is a special monome with $p = 0$)[3]. We use the polymorphism of C++. We have a parent class `Derivative` with four child classes `Constant`, `ExpMonome`, `Sum`, `Product` that derive from `Derivative` and implement virtual (abstract) methods. They represent constants, monomes $(t+p)^b$, sums of two expressions and products of two expressions respectively. To prevent the exponential growth of the number of terms while increasing the order of the derivatives, we do test cases to know which of the four child types should be returned and avoid unnecessary objects. As an example, a product of a constant 1 and a monome is transformed into a simple monome, removing 2 objects from the 3 initial. Figure (2) illustrates this.

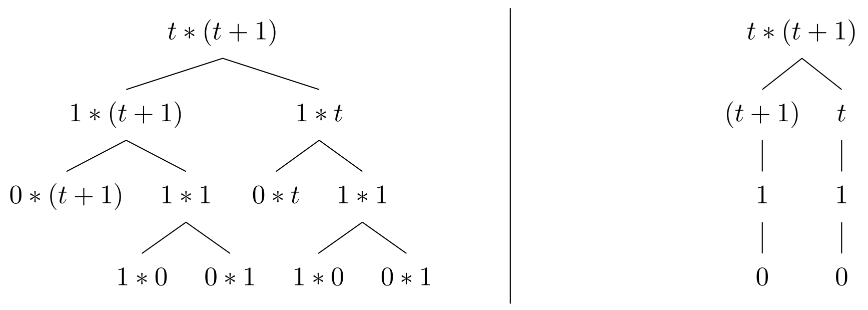

Figure 2: (Left) The root is the function that we need the first to third order derivatives. Children of depth $i \geq 1$ represent the i-th derivatives. Without doing any check on the derivatives, we see that 22 objects have been created. (Right) By doing type checks, we remove unnecessary objects and cut a branch if some derivatives are zero. In this example, only 8 objects are created.

3.4 Optimization with Vectorization

Vectorization comes from applying the same operation simultaneously to multiple and independent pieces of data [10]. It is a process available on computer architectures today and increases the speed of calculations. Compilers are built to recognize operations that could be vectorized, e.g. vector/matrix manipulations. Vectorization brings an advantage when $\boldsymbol{k}(\mathcal{D}, \mathcal{D})$ is computed following (5) and (6). We can compute its elements sequentially. A better approach is to group intermediate results in matrices and apply k_S and k_T on them, letting the compiler optimize the calculations for a faster computation. Let $\boldsymbol{t} = (t_1, \cdots, t_n)$. To compute $k_T(t_i - t_j)$ for each pair $\{(\boldsymbol{x}_i, t_i, y_i), (\boldsymbol{x}_j, t_j, y_j)\} \in \mathcal{D}$ we do

$$\begin{pmatrix} t_1 \\ \vdots \\ t_n \end{pmatrix} \to \boldsymbol{Z}_1 = \begin{pmatrix} t_1 & t_1 & \cdots & t_1 \\ \vdots & \vdots & \ddots & \vdots \\ t_n & t_n & \cdots & t_n \end{pmatrix}, \quad \boldsymbol{Z}_1 - \boldsymbol{Z}_1^T = \begin{pmatrix} t_1 - t_1 & t_1 - t_2 & \cdots & t_1 - t_n \\ t_2 - t_1 & t_2 - t_2 & \cdots & t_2 - t_n \\ \vdots & \vdots & \ddots & \vdots \\ t_n - t_1 & t_n - t_2 & \cdots & t_n - t_n \end{pmatrix}$$

[3]One could say that writing the closed forms for derivatives is faster. However (i) it is not modular and (ii) the number of terms for the closed form grows exponentially with the order of the derivatives.

$$\boldsymbol{Z}_1 - \boldsymbol{Z}_1^T \xrightarrow{k_T} \boldsymbol{Z}_2 = \begin{pmatrix} k_T(t_1-t_1) & k_T(t_1-t_2) & \cdots & k_T(t_1-t_n) \\ k_T(t_2-t_1) & k_T(t_2-t_2) & \cdots & k_T(t_2-t_n) \\ \vdots & \vdots & \ddots & \vdots \\ k_T(t_n-t_1) & k_T(t_n-t_2) & \cdots & k_T(t_n-t_n) \end{pmatrix} \tag{20}$$

Then for $k_S(||\boldsymbol{x}_i - \boldsymbol{x}_j||_2)$ for each pair $\{(\boldsymbol{x}_i, t_i, y_i), (\boldsymbol{x}_j, t_j, y_j)\} \in \mathcal{D}$, we use the following relation: for any pair of real vectors $\boldsymbol{x}, \boldsymbol{y}$:

$$||\boldsymbol{x} - \boldsymbol{y}||_2^2 = ||\boldsymbol{x}||_2^2 + ||\boldsymbol{y}||_2^2 - 2\langle \boldsymbol{x}, \boldsymbol{y} \rangle .$$

By writing $\boldsymbol{x}_i = (x_{i,1}, \cdots, x_{i,d})$ of $(\boldsymbol{x}_i, t_i, y_i) \in \mathcal{D}$, $1 \leq i \leq n$, we compute

$$\begin{pmatrix} x_{1,1} & \cdots & x_{1,d} \\ \vdots & \ddots & \vdots \\ x_{n,1} & \cdots & x_{n,d} \end{pmatrix} \to \begin{pmatrix} ||\boldsymbol{x}_1||_2^2 & \cdots & ||\boldsymbol{x}_n||_2^2 \end{pmatrix} \to \boldsymbol{W}_1 = \begin{pmatrix} ||\boldsymbol{x}_1||_2^2 & \cdots & ||\boldsymbol{x}_n||_2^2 \\ \vdots & \ddots & \vdots \\ ||\boldsymbol{x}_1||_2^2 & \cdots & ||\boldsymbol{x}_n||_2^2 \end{pmatrix}$$

$$\boldsymbol{W}_2 = 2 \begin{pmatrix} x_{1,1} & \cdots & x_{1,d} \\ \vdots & \ddots & \vdots \\ x_{n,1} & \cdots & x_{n,d} \end{pmatrix} \begin{pmatrix} x_{1,1} & \cdots & x_{n,1} \\ \vdots & \ddots & \vdots \\ x_{1,d} & \cdots & x_{n,d} \end{pmatrix} = 2 \begin{pmatrix} \langle \boldsymbol{x}_1, \boldsymbol{x}_1 \rangle & \cdots & \langle \boldsymbol{x}_1, \boldsymbol{x}_n \rangle \\ \vdots & \ddots & \vdots \\ \langle \boldsymbol{x}_n, \boldsymbol{x}_1 \rangle & \cdots & \langle \boldsymbol{x}_n, \boldsymbol{x}_n \rangle \end{pmatrix}$$

$$\boldsymbol{W}_1 + \boldsymbol{W}_1^T - \boldsymbol{W}_2 \xrightarrow{k_S} \boldsymbol{W}_3 = \begin{pmatrix} k_S(||\boldsymbol{x}_1 - \boldsymbol{x}_1||_2^2) & \cdots & k_S(||\boldsymbol{x}_1 - \boldsymbol{x}_n||_2^2) \\ \vdots & \ddots & \vdots \\ k_S(||\boldsymbol{x}_n - \boldsymbol{x}_1||_2^2) & \cdots & k_S(||\boldsymbol{x}_n - \boldsymbol{x}_n||_2^2) \end{pmatrix} \tag{21}$$

To compute $\boldsymbol{k}(\mathcal{D}, \mathcal{D})$, we multiple element-wise (21) with (20) (by using the Hadamard product) and at the end we multiply by λ. More specifically

$$\boldsymbol{k}(\mathcal{D}, \mathcal{D}) = \lambda \boldsymbol{W}_3 \odot \boldsymbol{Z}_2. \tag{22}$$

To highlight the speed advantage of vectorization, we compare in Section (6) two versions of the C++ implementation: one with vectorization and one without (i.e. computing sequentially each element of $\boldsymbol{k}(\mathcal{D}, \mathcal{D})$).

4 Implementation of State of the Art

In this section, we present some DBO solutions in more details for (i) an overview on the SOTA (R-GP-UCB and TV-GP-UCB in [4], and ET-GP-UCB in [5]) and what changes with W-DBO, (ii) implementing them to compare their performance with W-DBO. Even if we implement these three DBO solutions using only Numpy (and therefore Python), comparing their performance with W-DBO is fair because Numpy is also built on top of C/C++ code. Additionally, compared to W-DBO, the DBO solutions do not have time-consuming calculations.

4.1 R-GP-UCB and ET-GP-UCB

We start with R-GP-UCB of [4] and ET-GP-UCB of [5]. To remove irrelevant observations, R-GP-UCB resets $\mathcal{D}$ periodically, while ET-GP-UCB when an event is triggered. The objective function f is changing and removing all observations after some time allows the model to adapt to new data without being influenced by outdated information. More details on the resetting period of R-GP-UCB can be found in [4]. The event triggered in ET-GP-UCB assumes that if an observation is far away from what the mean and variance of the GP would imply in the next prediction, then f has changed significantly and we should reset $\mathcal{D}$ for the same reason as R-GP-UCB.

In summary, R-GP-UCB ensures that the learning process remains responsive to recent changes. Similarly, ET-GP-UCB leverages an event trigger to update the dataset based on significant changes in the observed data, maintaining an up-to-date model. Both methods aim to balance the trade-off between exploration and exploitation, ensuring that the GP remains accurate and relevant as new data is observed and as the function evolves. In contrast, W-DBO proposes to quantify how relevant an observation is before removing it. Resetting $\mathcal{D}$ completely would probably remove observations with valuable information for the optimization.

4.2 TV-GP-UCB

TV-GP-UCB in [4] models the dynamics of f with two kernels, one for the spatial dimensions and one for the time dimension. The older the observation is, the less it should contribute for the next predictions. The time kernel has the form

$$(1-\epsilon)^{\frac{|i-j|}{2}} \tag{23}$$

where i and j are indexes of $\{(\boldsymbol{x}_i, t_i, y_i), (\boldsymbol{x}_j, t_j, y_j)\} \in \mathcal{D}$ and $\epsilon \in [0,1]$ (more details are provided in [4]). Note that the sampling frequency of this algorithm (and also for R-GP-UCB and ET-GP-UCB) is discrete, and this one never removes stale data. Again in long-time optimization, the optimum of f is not tracked accurately. By keeping all observations, the inference time of the GP grows asymptotically, and the sampling frequency decreases (which induces less queries). Instead, W-DBO (i) removes stale data to optimize f rapidly on a long-period time and (ii) is more flexible as the time kernel can be any kernel.

5 Building Python Packages

This section details how the Python packages for the criterion and W-DBO have been built. We used Pybind11 to create the Python wheels required to install `wdbo_criterion`. Wheels are a standard Python distribution format. They are used to install a package and without any C/C++ extensions, these packages work on any platform (and can be generated on any computer). Because it has C/C++ extensions, `wdbo_criterion` is called platform specific which means that the wheel created on the computer needs to be installed on the same platform of the computer, including the same Python version. To create wheels for each platform and Python version, we use `cibuildwheels`[4]. This framework, using any continuous integration platform, creates virtual environments (Linux, Windows, MacOS) to create wheels on these platforms.

Since `wdbo_algo` has no C/C++ extensions, we just need to create a single wheel for any platform and Python version. These packages are uploaded on the Python repository PyPI. They can be simply installed using the Python package manager `pip`.

6 Numerical Results

In this section, we show the numerical results of the C++ implementation and the overall performance of W-DBO compared to the SOTA solutions. We call a run the calculations of (8) for each observation in $\mathcal{D}$. To analyse the performance of the C++ implementation, we compare the execution time of multiple runs of two C++ implementation (one with and one without vectorization techniques) with a Python implementation. The final results are shown on Figure (3) and show great speed improvements with both C++ versions. We achieve between one and two orders of magnitude faster compared to the Python version. The plot shows also the effectiveness of vectorization techniques as it is this version that runs faster.

The overall performance of W-DBO compared to SOTA solutions is shown in Figure (4). It shows the normalized average regret of many DBO solutions, including W-DBO (see the plot in [3] for more details).

[4]https://cibuildwheel.pypa.io/en/stable/

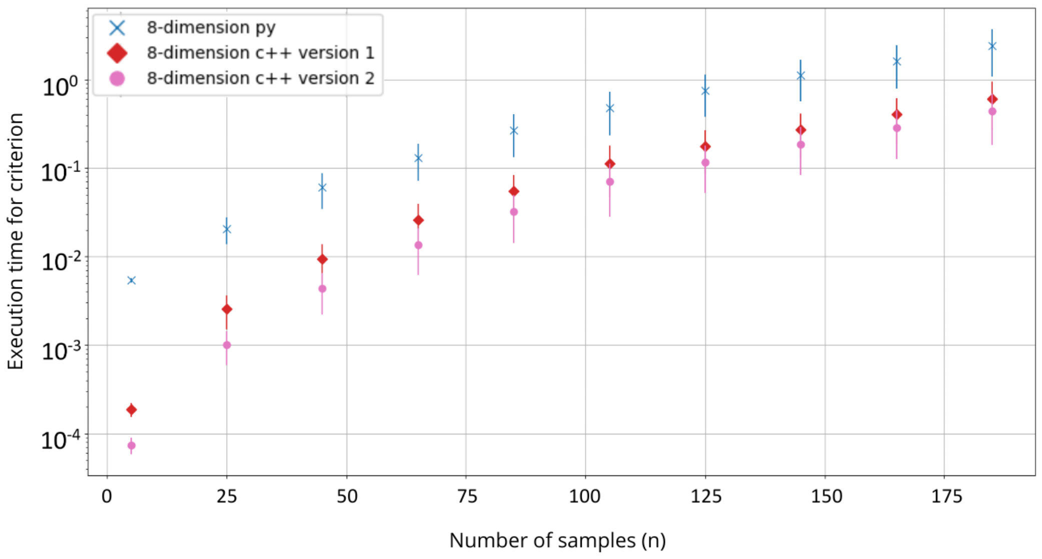

Figure 3: Average execution time of five runs for many sizes of dataset (i.e. $|\mathcal{D}|$) with different implementations. The blue dots are the Python implementation, the red are the C++ implementation without vectorization and the pink are with vectorization. The plot is in logscale. We achieve between one and two orders of magnitude faster with the second C++ version compared to the Python version.

7 Discussion

7.1 Application of DBO

Consider a WLAN (Wireless Local Area Network) created by many Access Points (APs) and we need to adapt their transmitting power. The power would depend on many factors, for example the data rate that people, connected to the WLAN, would like to use (which is a dynamic problem). In the context of DBO, we would like to find the optimal configuration for the transmitting power of each AP.

For this problem, W-DBO can be implemented on the embedded devices that are on each AP. Even if memory management has been taken into account during the implementation, great improvements should be made to reduce the resources needed. A non-exhaustive list would be: (i) since many matrices are symmetric, we could consider only the upper triangular part of the matrices and report the values to the other half (reducing by 2 many operations), (ii) improving the AD framework by reducing the number of pointers and matrices used, (iii) working with inplace's Eigen methods to avoid copies of data structures during algebra manipulations.

7.2 Profiling of the C++ code

A great way of seeing the performance of a piece of code is by profiling it. Profilers analyses how much time the code takes to run the different methods, how many memory accesses have been done, etc. During the C++ implementation, we ended up with a matrix that could not take advantage of matrix decomposition and linear solvers. This left us no choice than inverting this matrix alone. To be sure that this inversion was time-consuming, we profiled 300 runs of our vectorized C++ version with exactly the same inputs. The profiling was done on a computer equipped with an Intel Core i7 9th generation, RTX2060 GPU with 32GB of RAM. The results are shown in Table (1). In fact, the methods used to invert this matrix (`inverse_matrix_cholsky` and `F_val`) consume 50% of the total running time. The column showing the total time took in minutes is just for completeness in the sense that profiling adds time in the computations (ie. it is not the real time taken by 300 runs). Some work could be done to find a way of removing these two methods, giving `wdbo_criterion` better performance.

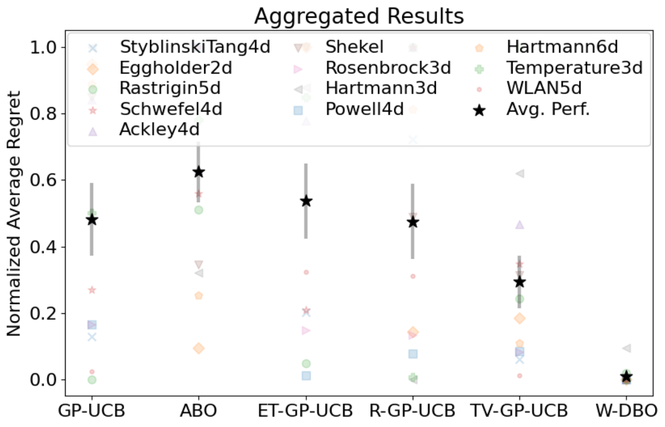

Figure 4: Normalized average regrets for many DBO algorithms. The average performance of the DBO solutions are given in black. W-DBO beats the others with a good margin.

Table 1: Results of profiling the vectorized C++ implementation. The table summarizes 300 runs. Each run calls the criterion for $|\mathcal{D}| = 25$, dimension of each point $d = 5$, $\lambda = 0.72971242$ with RBF space and time kernels of lengthscales $l_s = 0.2908291$ and $l_t = 0.18401412$. The same dataset $\mathcal{D}$ is provided as inputs for `wdbo_criterion`. The main method called is `wasserstein_criterion` (as the number of call is 300). The second and third methods, used to invert directly this particular matrix without linear solvers, take 50% of the total running time.

Method's name	Number of call	Time inside (%)	Time inside
`wasserstein_criterion`	300	100	`1h 05m`
`inverse_matrix_cholsky`	$7'800$	36.71	`23m 53s`
`F_val`	$7'500$	24.82	`16m 8s`
`Eigen's internal method`	$193'200$	23.24	`15m 6s`
`isPsd`	$15'600$	17.06	`11m 6s`

8 Conclusion

DBO appears in many fields of engineering such as robotics and wireless networks. It takes advantage of a GP to optimize a time-varying function f, and needs to manage the number of observations kept during the optimization. In [3], W-DBO proposes to remove observations that are irrelevant for the next predictions depending on the value of a criterion based on the Wasserstein distance. This irrelevancy, known as data staleness, is inherent to DBO. As the time goes on, the optimum of f changes and past observations bring less information on the future function values. Removing them is necessary and it decreases the inference time of the GP. To continuously optimize f and keep track of the optimum accurately, the criterion is implemented in C++, as it involves heavy computations. This work explains how this implementation has been conducted, presents the numerical and speed improvement challenges (e.g., inverse of matrices, AD and vectorization), and shows how the implementation is efficient. Additionally, it provides Python packages for W-DBO and the criterion. Minimize the resources needed to compute the criterion and find a way to remove `inverse_matrix_cholsky` from the implementation are works that could be done to improve the overall performance of W-DBO.

References

[1] HM Antia. *Numerical methods for scientists and engineers.* Vol. 2. Springer, 2012.

[2] Anthony Bardou, Patrick Thiran, and Thomas Begin. “Relaxing the Additivity Constraints in Decentralized No-Regret High-Dimensional Bayesian Optimization”. In: *arXiv preprint arXiv:2305.19838* (2023).

[3] Anthony Bardou, Patrick Thiran, and Giovanni Ranieri. *This Too Shall Pass: Removing Stale Observations in Dynamic Bayesian Optimization*. 2024. arXiv: `2405.14540 [stat.ML]`.

[4] Ilija Bogunovic, Jonathan Scarlett, and Volkan Cevher. "Time-varying Gaussian process bandit optimization". In: *Artificial Intelligence and Statistics*. PMLR. 2016, pp. 314–323.

[5] Paul Brunzema et al. "Event-triggered time-varying bayesian optimization". In: *arXiv preprint arXiv:2208.10790* (2022).

[6] Sébastien Bubeck. "Introduction to online optimization". In: *Lecture notes* 2 (2011), pp. 1–86.

[7] Peter I Frazier. "Bayesian optimization". In: *Recent advances in optimization and modeling of contemporary problems*. Informs, 2018, pp. 255–278.

[8] Gaël Guennebaud, Benoit Jacob, et al. "Eigen". In: *URl: http://eigen.tuxfamily.org* 3.1 (2010).

[9] Wenzel Jakob, Jason Rhinelander, and Dean Moldovan. "pybind11–Seamless operability between C++ 11 and Python". In: *URL: https://github.com/pybind/pybind11* (2017).

[10] Ken Kennedy and John R Allen. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., 2001.

[11] Daniel J Lizotte et al. "Automatic Gait Optimization With Gaussian Process Regression." In: *IJCAI*. Vol. 7. 2007, pp. 944–949.

[12] Anton Mallasto and Aasa Feragen. "Learning from uncertain curves: The 2-Wasserstein metric for Gaussian processes". In: *Advances in Neural Information Processing Systems* 30 (2017).

[13] William H Press. *Numerical recipes 3rd edition: The art of scientific computing*. Cambridge university press, 2007.

[14] Maximilian E Schüle et al. "LLVM code optimisation for automatic differentiation: when forward and reverse mode lead in the same direction". In: *Proceedings of the Sixth Workshop on Data Management for End-To-End Machine Learning*. 2022, pp. 1–4.

[15] Matthias Seeger. "Gaussian processes for machine learning". In: *International journal of neural systems* 14.02 (2004), pp. 69–106.