Universidad Zaragoza

1542

Master Thesis

Development of context based Bayesian optimization techniques for the analysis of biometric data

Author

Gabriel Olteanu Morozan

Director

Rubén Martínez-Cantín

ESCUELA DE INGENIERÍA Y ARQUITECTURA

2025

Index

Resumen

Esta tesis presenta el desarrollo de una extensión contextual de Optimización Bayesiana sobre la biblioteca BayesOpt y la valida mediante el análisis y optimización de algoritmos de procesamiento de señales EEG para la estimulación del sueño profundo. La estimulación auditiva durante el sueño NREM requiere una predicción precisa de las dinámicas neurológicas, y el rendimiento de los detectores existentes depende de hiperparámetros ajustables que varían entre sujetos y noches. Para abordar esta variabilidad, extendemos los algoritmos de optimización bayesiana en BayesOpt para permitir la optimización de hiperparámetros con información contextual fija, como la identidad del sujeto, o subdivisión de la noche.

El trabajo introduce modificaciones en el backend C++ de BayesOpt, lo expone a una interfaz en Python mediante el uso de Cython y proporciona una serie de automatizaciones para facilitar experimentación a gran escala y garantizar su reproducibilidad. Utilizando registros EEG reales de múltiples sujetos a lo largo de diferentes noches, exploramos la optimización de tramos parciales de la noche, la acumulación incremental de contexto y la transferencia de conocimiento entre noches. Los experimentos realizados muestran que la modelización contextual acelera la convergencia, incluso con información contextual limitada, y permite ampliar el modelo de forma eficiente mediante el uso de datos de optimizaciones previas, sin necesidad de volver a evaluar funciones objetivo computacionalmente costosas.

La extensión contextual se empleó en aprendizaje por refuerzo para agarre robótico, demostrando que el enfoque contextual generaliza a problemas de optimización multi-tarea. Los resultados indican que el contexto actúa como mecanismo de reutilización de conocimiento, permitiendo adaptarse más rápido a tareas nuevas pero relacionadas, en este caso optimizar el agarre a distintos objetos. Las conclusiones extraídas de este caso se correlacionan directamente con nuestros hallazgos, mostrando que la construcción de contexto mejora las puntuaciones respecto al modelo base, con márgenes mayores a medida que aumenta la información contextual.

La tesis aporta un marco práctico, extensible y validado experimentalmente para la optimización bayesiana con contexto en escenarios multi-tarea y de optimización personalizada, demostrando beneficios reales para la regulación del sueño.

Abstract

This thesis presents the development of a contextual Bayesian optimization extension over the BayesOpt library and validates it through the analysis and optimization of EEG signal processing algorithms for deep sleep stimulation. Auditory stimulation during NREM sleep requires accurate prediction of neurological dynamics, and the performance of existing detectors depends on tunable hyper-parameters that vary across subjects and nights. To handle this variability we extend the Bayesian optimization algorithms in BayesOpt to enable hyper-parameter optimization with fixed contextual information like subject identity and night segmentation.

The work introduces modifications to the BayesOpt C++ backend, exposes it to a Python interface through the use of Cython and provides a series of automations for large scale experimentation facilitation and reproduce-ability. Using real EEG recordings from multiple subjects across different nights, we explored partial-night optimization, incremental context accumulation and multi-night knowledge transfer. The conducted experiments show that contextual modeling accelerates convergence, even for limited contextual information, and allows efficient expansion of the model through the use of previous optimization data without the need to re-evaluate computationally expensive objective functions.

Beyond sleep-related applications, the contextual expansion was utilized in robotic grasping reinforcement learning, showing that the contextual approach generalizes to multi-task optimization problems. The results indicate that context serves as a mechanism for knowledge reuse, enabling faster adaptation to new but related tasks, in this case optimizing for grasping different objects. The conclusions extracted from this use case directly correlated with our findings, showcasing the principle that context building improved the scores over the baseline with bigger margins as the contextual data increased.

Overall, the thesis contributes a practical, extensible, and experimentally validated context-based Bayesian optimization framework for multi-task and personalized optimization scenarios, which resulted in real world benefits in the field of deep sleep regulation and expands the horizon over the applications of this framework in real case scenarios.

Chapter 1

Introduction

1.1 Bayesian optimization for sleep disorders

Sleep disorders are a problem. They affect a significant portion of the population and can manifest in many forms, from insomnia and fragmented sleep to disorders that impair the length and quality of sleep. Poor sleep has short and long term consequences on cognitive functioning, mood, metabolic health and overall life quality. As research advances it has become increasingly clear that disrupted sleep architecture, particularly the degradation of deep sleep plays a central role in many of these issues.

Slow waves (SW) play a central role in maintaining restorative sleep, as they reflect synchronized cortical down-states and up-states that support memory consolidation and neuronal recovery. When these oscillations weaken or fragment, the brain's capacity to engage in the normal processes of non-rapid eye movement (NREM) sleep diminishes. Aging is associated with marked reductions in SW activity, leading to lighter and more disrupted sleep[1]. This is even more pronounced in neuro-degenerative disorders like Parkinson's disease, with reduced SW amplitudes. These changes impair deep sleep and the naturally occurring SW enhancement[2, 3].

There are EEG SW tracking devices that deliver stimuli at the SW up-phase to elongate the slow waves and improve NREM sleep. Auditory stimulation timed to coincide with the up-phase of SW has been shown to amplify ongoing oscillations and enhance SW activity. Early work demonstrated that closed-loop stimulation can reliably strengthen slow oscillations and improve memory consolidation[4]. Extending this approach to older adults showed promise even in populations with reduced slow-wave amplitude (SWA) with variable effectiveness[5]. These wearable autonomous devices depend on continuous EEG monitoring to deliver sounds at a precise time throughout the night.

Because timing the stimulation to the SW up-phase determines whether the oscillation is enhanced or disrupted, the device must estimate the instantaneous EEG

phase before it happens with minimal delay. Traditional approaches relied on simple amplitude thresholds, which are limited by distortions from real-time filtering[6]. More advanced methods like the phase-locked loop (PLL)[7] improve prediction by attempting to track the oscillatory rhythm, but struggle when the SWAs are low or frequencies deviate from 1Hz. This motivated the development of more robust phase-estimation algorithms capable of adapting dynamically to individual EEG characteristics[8].

EEG SW are substantially variable across individuals and nights, driven from differences in sleep pressure, neurological status, age, or health conditions. Neuro-degenerative disorders introduce additional variability that complicates phase tracking, as well as the broad age-related changes in sleep structure[9, 1]. As a result, algorithms such as the amplitude threshold (AT) or PLL must rely on hyper-parameters that determine how sensitively they respond to incoming EEG dynamics. Tuning these parameters is essential to ensure accurate phase targeting across different populations, with degraded performance when the parameters are not adapted to the specific subject and recording[8]. Our goal is to optimize these hyper-parameters to maximize the proportion of stimulations occurring during the SW up-phase, where the oscillatory activity enhancement is most effective[4]. When evaluating performance, each configuration requires running the full detection pipeline on an EEG recording, making the objective function expensive to evaluate and without analytic gradients. Bayesian optimization is especially suitable for this scenario, as it efficiently explores high-cost, non-convex parameter spaces using a probabilistic surrogate model that performs well even with low samples.

To perform this optimization, we employ the BayesOpt library developed by Rubén Martínez Cantin[10], which provides a flexible and computationally efficient framework for Bayesian optimization methods. Furthermore, we are going to modify the library to make contextual information available through the optimization, which is going to allow us to optimize the hyper-parameters for specific EEG recordings, and then relate the knowledge between recording optimizations by labeling each recording with the contextual information of that recording. This exposes difficult to optimize but easy to aggregate contextual dimensions (subject and night specifics that could be further segmented into age, gender, room temperature, light in the room, etc. Which we had no access to.) to the problem and allows both the creation of a big problem information model and the use of that information for aiding specific EEG recording optimizations in smaller time, even real time.

1.2 Objectives and scope

The main objective of this work is the design and development of a context-based Bayesian optimization expansion based on the BayesOpt Bayesian optimization library, in order to evaluate its performance with a study of sleep patterns. The method is first assessed in a standard, non-contextual setting and subsequently with contextual information incorporated into the optimization loop. The objective of this feature is to improve the performance of sleep assistant devices, optimizing the configuration parameters adapted to a specific user, but using as prior knowledge the results obtained from the analysis of sleep patterns from multiple users, taking advantage of the similarities between users via the similarities in the context data.

Beyond implementing the contextual extension itself, this work includes the design of the software components required to integrate it cleanly into the existing BayesOpt framework, ensuring modularity, maintainability, and compatibility with future developments of the library.

A central part of the thesis is the experimental evaluation of the proposed approach. We investigate several hyperparameter-optimization strategies and context-construction methods in the use case of EEG signal processing and slow-wave (SW) up-phase prediction, making use of the Phase-Locked Loop (PLL) algorithm for real-time EEG analysis. The results of these experiments are compared with those from related work such as the contextual optimization framework applied to robot grasping[11], to demonstrate the generality and usability of the interface developed here.

Summarizing, the thesis aims to contribute a practical and extensible contextual Bayesian optimization framework and to demonstrate its potential impact in domains where personalization is essential, such as wearable devices designed to improve sleep quality in sleep-deprived populations.

1.3 Methodology

To carry out all our objectives through completion we will need to rely on a series of tools:

First, on the software side, we will need to rely on **BayesOpt**. This library is publicly hosted on **Github**, and so we will carry out our development work on a

separate **Git** branch to be able to merge it back into production once we are done with its development. The library works best in **Linux**, with **Ubuntu** being the main target of most of their developers. In my case, I will develop from Windows, so I will be using **Ubuntu 22.04** as my target through **WSL2**. The core of BayesOpt is written in **C** and **C++**, with a **Python** interface that matches the BitBrain provided code, so those will be the programming languages I will use as well as some **Cython** to expose the core to Python. I will use **Visual Studio Code** as my integrated development environment (IDE), as well as the occasional **Visual Studio** for its improved C++ debugging capabilities. Additionally, since there will be multiple instances where we need to perform long, repeatable and parametrize-able optimizations, we will use **Bash** as our shell and scripting language, with all of the results being saved both in NumPy format as well as through Matplotlib for generating graphs. To be able to write the thesis in **LaTeX**, make it accessible through the web and allow collaborative insights to be added by any member at any time, we will use **Overleaf**.

Chapter 2

EEG-Based Sleep Optimization

2.1 Overview of EEG-Based Sleep Enhancement

Electroencephalography (EEG) provides a non-invasive, real-time window into brain activity and has become a foundational tool for sleep research and neural stimulation. Among various features observed in EEG during sleep, slow waves (SW) are of particular interest. These oscillations, typically in the frequency range of 0.5–4.5 Hz, occur prominently during non-REM (NREM) sleep and have been strongly linked to memory consolidation and restorative brain processes[8]. Enhancing slow wave activity (SWA) in subjects with low-amplitude SW has shown potential cognitive and physiological benefits, which is prominent in populations with impaired sleep quality, such as older adults or patients with neuro-degenerative conditions.

To influence slow wave dynamics, a growing body of researchers explores in-phase auditory stimulation, delivering sound cues precisely at specific phases of ongoing SWs. Specifically, triggering sounds during the SW up-phase (0°–90°) has been shown to increase the amplitude and duration of slow waves, whereas stimulation during the down-phase has minimal or even negative effects [8, 4].

Our work relies on optimizing the parameters for the PLL algorithm from Ferster et al.[8] which is why it's going to be the main source of information when talking about EEG based signal processing and auditory stimulation.

2.2 Sleep stages based on EEG signal processing

The work from Ferster et al.[8] divided the recordings on different stages based on whether it was NREM sleep (N2 and N3 sleep) or non-NREM sleep (awake, N1 and REM). This was decided based on EEG signal spectral power crossing certain frequency bands (0.5-2Hz, 2-4Hz, 20-30Hz), which was divided into 4s chunks based on the last 80s of EEG.

The decision on whether to generate auditory stimuli relied on a decision logic tree based on NREM sleep, SWA, EEG beta power and EEG phase target conditions. If the spectral power crossed predefined thresholds, the EEG signal was classsified as NREM sleep. High delta power is indicative of awake, light sleep, artifacts and arousal, so when crossing a threshold (17-22Hz) the algorithm would consider the last 4s to be non-NREM.

2.3 Algorithms for SW tracking

Some previous works to the one we base our optimization on suffer from poor phase accuracy at these low EEG amplitudes and with fluctuating SW frequencies. [8] introduces an algorithm based on a phased-locked-loop (PLL) and the phase vocoder (PV). They also compare their performance relative to a simpler amplitude threshold approach. We are only going to go into explaining the AT and PLL algorithms as the vocoder was deemed to have too much of an overhead and a mixed AT and PLL algorithm was used for our optimizations, which was passed down as a requirement from BitBrain. One advantage of optimizing using our current setup is the ability to change the SW tracking algorithm if necessary, as Bayesian optimization treats the objective function as a black box and doesn't rely on any of the parameters to work.

2.3.1 Baseline amplitude threshold algorithm

The Ferster et al.[8] approach is based on a previous threshold-based approach from Fattinger et al.[6] where a pink noise was played when a band-passed filtered EEG signal crossed a 50μV EEG amplitude threshold (AT) as well as an electromyography (EMG) threshold. Their simplification was to relax the pre-processing filtering condition of the EEG signal to reduce the non-linear phase delays and replacing the EMG condition with EEG based sleep sub-stages as to only need to optimize the EEG AT.

2.3.2 The PLL Algorithm for Real-Time SW Detection

This algorithm consists on a phase detector and a number-controlled oscillator (NCO) as seen in Figure 2.1. The phase detector multiplied the preprocessed EEG signal s_n with the PLL output s^p_{n-1}, which were sinusoidal and cosine functions. The phase detector output resulted in a signal error term s^e_n modulated by the phase detector gain κ_{pd}. The error signal was used as the NCO control signal to generate a proportional periodic signal to minimize the phase error s^e_n relative to s_n. The rate of change in

the NCO was given by its gain κ_{NCO}, represented as the sensitivity. The NCO output was constructed as a cosine signal with angular frequency 1Hz and a phase given by $\phi_n^p = \phi_{n-1}^p - s_n^e * K_{NCO}$. The stimulation was given when ϕ_n^p crossed a target phase ϕ_T. The PLL gains κ_{pd} and κ_{NCO} where multiplied and considered as one unique κ_{PLL}, leaving the optimization with only K_{PLL} and ϕ_T as tunable parameters.

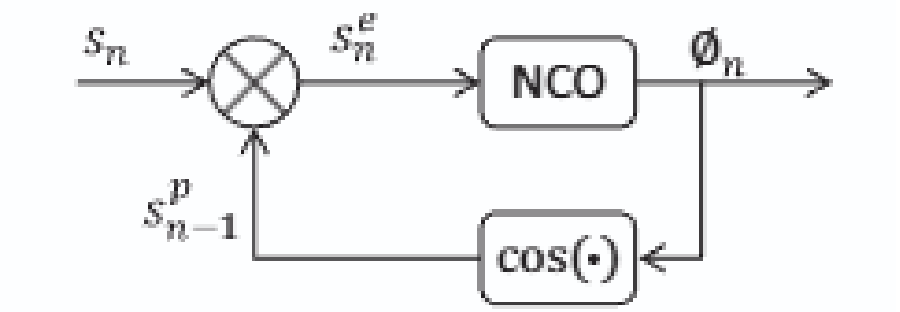


Figure 2.1: Phase-locked loop (PLL) block diagram. The EEG input signal s_n is multiplied by the PLL output signal s_{n1}^p generating an error signal s_n^e that serves as a control signal for the number-controlled oscillator (NCO). Taken from[8] Fig 1.

2.4 Optimization Metrics and Loss Integration

To calibrate the auditory stimulation system, we optimize several performance metrics derived from real-time EEG interactions: CMAE_{45} $\text{PAS}_{\in \text{UP}}$ $\text{PAS}_{\notin \text{UP}}$.

Previous works highlighted the importance of targeting of the SW up-phase to induce a SWA enhancement effect, while targeting the down-phase would have a negative effect. The first optimization objective was minimizing the circular mean absolute error (CMAE) with a target angle 45º to account for pre-processing filtering and processing delays, such as

$$\text{CMAE}_{45} = \frac{|\text{circularMean}(\phi) - 45°|}{180°} \tag{2.1}$$

Furthermore, there is a high interest in maximizing the amount of stimulation triggers delivered within a night. This is night dependent, favoring longer nights, so to make it night and subject independent a new metric measuring the percentage of active stimulations (PAS) in the SW up-phase $\text{PAS}_{\in \text{UP}}$ during NREM sleep was introduced, such as

$$\text{PAS}_{\in \text{UP}} = \frac{\#\text{ of Stimulations} \in (0 - 90°)}{(\#\text{ of NREM windows}) * (\text{MaxStim})} \tag{2.2}$$

To minimize the possibility of stimuli occuring at undesired phases of the night such as the SW down-phase a new metric $\text{PAS}_{\notin \text{UP}}$ is introduced, calculated as

$$\text{PAS}_{\notin \text{UP}} = \text{PAS}_{ALL} - \text{PAS}_{\in \text{UP}} \tag{2.3}$$

In Ferster et al.[8], these metrics are combined using a multi-objective optimization strategy that minimizes an Euclidean distance from an ideal point (0,0,1) representing (CMAE_{45°, $P_{\neg \mathrm{up}}$, P_{up}), defined as:

$$\mathrm{ED} = \sqrt{\mathrm{CMAE}_{45}^2 + \mathrm{PAS}_{\notin \mathrm{UP}}^2 + (1 - \mathrm{PAS}_{\in \mathrm{UP}})^2} \tag{2.4}$$

In our own optimization procedure, we adapt this idea by defining a weighted mean loss function that integrates all four metrics with defined weights α_i, allowing fine-grained regularization:

$$\mathrm{Score} = \alpha_1 * \mathrm{CMAE}_{45} + \alpha_2 * \mathrm{PAS}_{\in \mathrm{UP}} + \alpha_3 * \mathrm{PAS}_{\notin \mathrm{UP}} \tag{2.5}$$

The weights determine how much importance we give to each metric when optimizing for the score. Since each metric is calculated using different units and the score function is unit-less, we need to regularize the values from each metric. We do this by sampling the objective function with good default hyper-parameter values and saving each of the returned metric values. Then, we give each of their alphas the value of the inverse of their respective collected metric. Apart from using the alphas as a regularizing factor, not all metrics are equally important for maximizing correct stimulations. Through a preliminary study it was deemed that the primary objective was optimizing for $\mathrm{PAS}_{\in \mathrm{UP}}$ and $\mathrm{PAS}_{\notin \mathrm{UP}}$, while optimizing for CMAE_{45} was of secondary importance. That's why the regularizing value of α_1 was in addition divided by 10.

In addition, the provided evaluator that allows us to perform the optimization of $\tau_{\max}$ (the threshold of the AT algorithm), κ_{pll}, and κ_{nco} (from the PLL) based on the aforementioned metrics, provides additional metrics to quantify the accuracy for targeting the SW up-phase respect to the desired 45^{o}, such as the circular mean (CM) and circular standard deviation (CSD). We decided not to use those metrics for the optimizer as they are secondary to the main objective and it would introduce more complexity to a problem that is constrained by the costly nature of the algorithm to a few number of samples.

Chapter 3

Bayesian Optimization

3.1 Bayesian Optimization: Principles and Mathematical Formulation

Bayesian optimization (BO) is a powerful technique for optimizing black-box functions that are expensive to evaluate, lack an analytical expression, and may include noise. The algorithm builds a surrogate probabilistic model to approximate the unknown objective function $f : \mathbb{X} \to \mathbb{R}$ and uses it to decide where to evaluate next.

Let $f(x)$ be a function defined over a compact domain $\mathbb{X} \subset \mathbb{R}^d$, where $d \geq 1$, and suppose we wish to minimize f. The main idea in BO is to model f as a realization of a stochastic process, most commonly a Gaussian Process (GP):

$$f \sim \mathcal{GP}(\mu(x), k(x, x')) \tag{3.1}$$

Here, $\mu(x)$ is the mean function and $k(x, x')$ is the covariance kernel that defines the correlation between the points of the function. After observing a set of n evaluations $\mathcal{D}_n = \{(x_i, y_i)\}_{i=1}^n$ where $y_i = f(x_i) + \epsilon_i$ where $\epsilon_i \sim N(0, \sigma_\epsilon^2)$ models observational noise, the posterior distribution at a new point x is given by:

$$\begin{aligned} \mu_n(x) &= \mu(x) + \boldsymbol{k}^T (K + \sigma_\epsilon^2 I)^{-1} (\boldsymbol{y} - \boldsymbol{x}) \\ \sigma_n^2(x) &= k(x, x) - \boldsymbol{k}^T (K + \sigma_\epsilon^2 I)^{-1} \boldsymbol{k} \end{aligned} \tag{3.2}$$

where $\boldsymbol{k} = [k(x, x_1), \ldots, k(x, x_n)]^T$ and K is the $n \times n$ kernel matrix $K_{ij} = k(x_i, x_j)$.

The acquisition function $a(x)$ guides the search for the optimum by balancing exploration and exploitation. A common choice is the Expected Improvement (EI), which models the potential information gain of exploring a new sample (It takes into account both the benefits of exploration of a region with low samples and exploitation of a region where we found good samples):

$$EI(x) = (\rho - \mu_n(x))\Phi(z) + \sigma_n(x)\phi(z), \quad \text{where } z = \frac{\rho - \mu_n(x)}{\sigma_n(x)} \tag{3.3}$$

Here, $\rho = \min_i y_i$ is the best observed value so far, and ϕ, Φ are the PDF and CDF of the standard normal distribution, respectively.

The bayesian optimization algorithm proceeds iteratively as follows:

Algorithm 1 Bayesian Optimization

Initialize dataset $\mathcal{D}_0 = \{(x_i, y_i)\}_{i=1}^{n_0}$
for $t = n_0, \ldots, N$ **do**
    Fit GP surrogate to $\mathcal{D}_t$
    Select next point: $x_{t+1} = \arg\max_x a(x)$
    Evaluate $y_{t+1} = f(x_{t+1}) + \epsilon$
    Update dataset: $\mathcal{D}_{t+1} = \mathcal{D}_t \cup \{(x_{t+1}, y_{t+1})\}$
end for
Return $\arg\min_{x_i \in \mathcal{D}_N} y_i$

3.2 Strengths and Limitations of Bayesian Optimization

Bayesian Optimization is particularly effective in scenarios where function evaluations are expensive, such as when each evaluation involves running a costly physical simulation or training a complex machine learning model. It is also well-suited for problems where no gradient information is available, making it ideal for black-box optimization settings. Furthermore, Bayesian Optimization performs well when the objective function is noisy or non-convex, as the Gaussian Process surrogate can effectively model uncertainty and nonlinear behavior. Another favorable condition is when the search space is low-dimensional, typically with dimensionality $d < 20$, since the computational cost of Gaussian Processes scales poorly with dimensionality.

Despite its strengths, Bayesian Optimization has limitations. It may perform poorly when the function is high-dimensional, due to the fact that Gaussian Process inference scales cubically with the number of observations, making it computationally infeasible for large datasets. Additionally, if the function landscape is flat or highly discontinuous, the surrogate model may struggle to generalize, leading to inefficient search behavior. Challenges also arise when there are many local minima and low correlation across the input space, as the model can be easily misled. Finally, in cases where the evaluation budget is large, simpler and more scalable optimization methods such as random search or evolutionary strategies may yield better performance with less computational overhead.

3.3 Incorporating Contextual Information

Traditional Bayesian Optimization only considers the data collected during the current optimization process. However, in many real-world applications, prior knowledge or results from similar optimization tasks are available. These may be represented as contextual information.

Context $c \in \mathcal{C}$ refers to auxiliary information that characterizes the environment or problem instance. For example, in tuning a sleep optimization algorithm for various users, context may include age, gender, or prior EEG characteristics. Another example of contextual data being exploitable in other fields includes the use of different object characteristics in a robotic grasping use case or different dataset characteristics for AutoML, as shown in some multi-task papers[12].

Including contextual data allows the surrogate model to condition not only on input variables x but also on the context c. This leads to a model:

$$f(x, c) \sim \mathcal{GP}(\mu((x, c)), k((x, c), (x', c'))) \tag{3.4}$$

In such settings, the optimization problem becomes one of maximizing the objective function not only over x, but conditioned on the context. There are many ways of introducing context:

If we want to find the parameters that maximize the function over all the available contexts, we are looking for this set of parameters:

$$\boldsymbol{x}^* = \arg\max_x \mathbb{E}_C f(x, c) \tag{3.5}$$

If instead, like our case, we want to search the optimal parameters for a fixed context, which cannot be explored during optimization, the set of parameters we are looking for are:

$$\boldsymbol{x}^* = \arg\max_x f(x, c') \tag{3.6}$$

Where c' is a set of fixed parameters belonging to C, and can be better visualized in Figure 3.1, which is taken from Figure 2 of Kandasamy et al[13]. Another possibility is using Equation 3.6 with a multi-fidelity approach like the one presented in the paper Multi-task Bayesian optimization[12]. The multi-fidelity approach relies in being able to approximate the value of a point in the objective function with variable degrees of precision. Useful for sampling lower fidelity samples in exploration and higher in exploitation modes for real-time applications. In our case it is not directly translatable to using a fraction of the NREM windows to evaluate our hyper-parameters since these

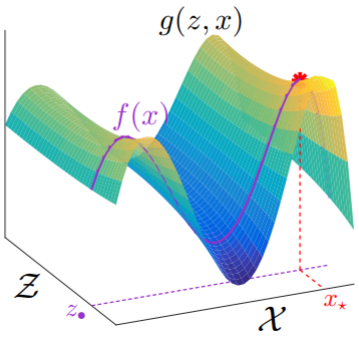


Figure 3.1: $g : Z \times X \to \mathbb{R}$ is a function defined on the product space of the fidelity space Z and domain X. The purple line is $f(x) = g(z_\bullet, x)$. We wish to find the maximizer $x_* \in \text{argmax}_{x \in X} f(x)$. The multi-fidelity framework is attractive when g is smooth across Z as illustrated in the figure.

windows are temporally correlated (they are taken in sequence) and we cannot choose to modify the precision of the samples during optimization like a pure multi-fidelity approach.

3.4 Motivation for Contextual Bayesian Optimization

Incorporating contextual information into the Bayesian Optimization process offers several advantages, particularly in scenarios involving related but distinct optimization tasks. One key benefit is the potential for improved convergence speed when prior optimization results from contextually similar scenarios are available. For instance, consider a setting where the goal is to optimize a function related to EEG data from human subjects. If one has previously optimized the objective for several children with a similar age range, and then we try to optimize for a 13-year-old participant, that previous information can be leveraged to accelerate the optimization, assuming age is a relevant factor influencing the objective function.

A second advantage lies in the opportunity to construct a global surrogate model, such as a Gaussian Process, trained on a wide variety of contexts. This global model can then be adapted or fine-tuned to individual users or tasks, enabling efficient transfer of knowledge while preserving task-specific accuracy. Such a hierarchical or multi-fidelity modeling approach allows the algorithm to generalize effectively across contexts without restarting the optimization process from scratch for each new task.

Moreover, contextual information can be used to externalize and abstract away

difficult to evaluate parameters from the optimization space. In practical applications, the objective function often serves as a surrogate for a more complex reality. Continuing with the EEG example, parameters such as a subject's age, gender, ethnicity, or lifestyle may significantly influence the function but are costly to obtain. Both in terms of time and financial resources. By treating such parameters as part of the context rather than as optimization variables, one can reduce the dimensionality and complexity of the problem, thereby accelerating the optimization process.

Lastly, the optimization landscape may involve heterogeneous parameter types, including continuous, discrete, and categorical variables. Standard Bayesian Optimization techniques often struggle to handle such mixed-type input spaces efficiently. By relegating some of these parameters to the contextual space, one can simplify the modeling requirements and avoid the limitations imposed by algorithms tailored to only one type of variable.

Overall, contextual Bayesian Optimization enables knowledge sharing across related tasks, enhances modeling flexibility, and reduces computational burden, making it particularly suitable for complex, high-dimensional, and resource-constrained optimization problems.

Chapter 4

BayesOpt: How it works and what it offers

4.1 General usage

BayesOpt is a library based on **C** and **C++** that implements **Bayesian optimization** efficiently to be used in problems such as *non-linear optimization*, *experimental design*, and *hyperparameter tuning*. To learn how to use the library, there is detailed *documentation*[1] that includes, among other things, a chapter on *installation*, *usage guide*, *demos*, *examples*, and even information on the *modules* and *classes* in the library.

The use of **BayesOpt** for optimizing a specific problem is done through the interfaces provided by the library. There are interfaces for **C**, **C++**, **Python**, **Matlab**, **Octave** and **Julia**, with **C++** having access to the largest number of features, and **Python** the least, as it is merely a wrapper of **C** and **C++** features using **Cython**. To discover the features offered by each of the interfaces, it is recommended to use as a starting point one of the demos available in the preferred language and consult the *documentation*.

Once the programming language is chosen, in general, three requirements must be defined: the optimization *function*, the optimization *parameters*, such as the number of iterations, random noise or kernel; and the *optimizer*, which can be *continuous*, *discrete*, or *categorical*, referring to the type of data in the problem to be solved. Usually, the parameters to modify are found in the *Basic parameter setup* section of the *documentation*, which also includes a list of all parameters and specific information on what they change.

[1] https://rmcantin.github.io/bayesopt/html/

4.2 Internal workings

There is a section in the *documentation* that shows the *class hierarchy*[2] of the library, where it can be seen that the number of classes and modules is quite extensive. To summarize, we can identify a series of relevant modules:

- **BayesOptBase** is the module that defines the basic functionality of *Bayesian optimization*, such as initializing the *Gaussian model*, executing one or all steps of the optimization, retrieveng the *optimum value*, saving or loading optimization data, or, for example, implementing the change of *contextual* data.

- **ContinuousModel** and **DiscreteModel** derive from **BayesOptBase** to implement models that handle *continuous* or *discrete* data (which also includes *categorical* data), as the techniques are different.

- **NonParametricProcesses** is a module containing a set of nonparametric processes (Gaussian process, Student's t process, etc.) for surrogate modelling. This is what defines the behavior of critical functions such as evaluating a point in the model or fitting the surrogate model to an initial set of points.

- **BOptState** is responsible for defining the state in which the optimization is found and can load or save it to disk in case of interruption.

- **Criteria** is the module that defines the base functionality for other modules that derive from it to decide, using the information from all previous steps, where to search within the established bounds for the next sample of the *Gaussian model*. From this class, others are derived that implement different criteria such as **ExpectedImprovement** or **ThompsonSampling**.

- **Dataset** defines the functionality of the container for points in the *Gaussian model*. It is responsible for storing and handling *continuous* data, from which other classes access and modify the contained information.

- **Kernel** is the module that defines the behavior of how the different dimensions of the points are correlated. There are a set of common **Atomic Kernels** from which common kernels are derived, like the ARD, hamming or ISO kernels, which among others define the variations of the Matérn kernels (MaternARD1, 3, 5 and MaternISO1, 3, 5) of which the default MaternARD5 is used in our optimization.

[2] https://rmcantin.github.io/bayesopt/html/inherits.html

– **Parameters** allows modifying the behavior of the optimizer through the definition of various *hyperparameters*, including the number of optimization steps (both initialization and during optimization, and even taking multiple samples at each step), adding *random noise* to the signal, different *sampling* algorithms, how to handle situations such as encountering a *valley* during optimization, among others.

– **PosteriorModel** guides the *Bayesian optimization* using different *non-parametric processes* such as distributions over surrogate functions. In other words, it defines the strategy to follow in order to determine the most interesting point to explore in the objective function using the information obtained in previous iterations. It includes methods such as **EmpiricalBayes**, **MCMCModel**, or **PosteriorFixed**.

– **ProbabilityDistribution** defines the different probability distributions that can be used as a basis to shape a surrogate function with the goal of progressively resembling the objective function. The base distributions used are the *Gaussian distribution* or the *Student T distribution*.

4.2.1 Additional Features of BayesOpt

BayesOpt provides two main styles of interfacing with the optimizer: a **callback style**, where a user-defined function is passed as a pointer to the optimizer (available in **C**, **C++**, **Python**, **Matlab**, and **Octave**), and an **inheritance style**, especially in **C++** and **Python**, where users can extend built-in model classes and override methods such as `evaluateSample()` for greater customization and control. BayesOpt also offers flexible configuration by allowing users to dynamically select different kernel functions, mean functions, surrogate models, and acquisition criteria through helper methods such as `set_kernel()`, `set_mean()`, `set_surrogate()`, and `set_criteria()`, facilitating experimentation across diverse optimization strategies.

It is important to note that BayesOpt is fundamentally *memory-based*: it updates the posterior distribution over the objective function sequentially with each evaluation, applying Bayesian inference rigorously to incorporate all observed data. Lastly, the library is released under the **AGPL-3.0 license** and users are encouraged to cite the foundational publication by Martinez-Cantin (JMLR 2014) when employing BayesOpt in academic work[10].

Chapter 5

Extending BayesOpt for Context-Aware Optimization

This chapter outlines the engineering work conducted to enable context-aware optimization in the `BayesOpt` library, focusing on both the core C++ backend and the Python interface. The implementation facilitates use cases involving varying external conditions (contexts) and provides flexible, robust tooling for iterative and interactive optimization experiments.

5.1 Core C++ Modifications for Context Handling

The introduction of contextual variables required substantial changes across the core **C++** components of **BayesOpt**. The main objective was to allow the optimizer to operate in a reduced "optimizable" subspace while still being able to condition evaluations, dataset storage, acquisition computations, and plotting utilities on a fixed set of contextual values. This entailed adding new public methods to expose context management externally, extending internal data structures to track context state, and ensuring that the Gaussian process and acquisition routines handled mixed-dimensional points correctly.

Initially, the `setContext()` method was implemented inside the **BayesOptBase** class, but later moved into **BayesOptContinuous** after it became clear that only the continuous optimizer would support contextual augmentation of the input space. The discrete and categorical models whose internal point representations differ remain context-agnostic, and their inherited `setContext()` remains a no-op. The new `setContext()` call accepts a vector of context values and a corresponding mask indicating which input dimensions are contextual (mask value 1) and which are optimizable parameters (mask value –1, later changed to 0). This design provides full flexibility on how context is positioned inside the underlying input vector, instead of assuming contexts are appended at the end (initially it was intended for the hypothetical case of wanting to change the mask, but it later became clear that changing the mask mid optimization would break the process). Internally,

`setContext()` initializes the stored context vector, its mask, and the derived context length. It also triggers a full re-initialization of the Expected Improvement sampler and associated **NLOPT optimizer** so that optimizable variables are sampled only within their designated dimensions. Bounds for contextual dimensions are ignored entirely, and the lower/upper bound arrays are reset to align strictly with the reduced optimizable subspace.

To ensure consistency, several helper functions were added to **BayesOptBase** for expanding or reducing points depending on whether contextual information is required. These include `addContextToPoints()` and `removeContextFromPoints()` in both vector and matrix form. The matrix versions required additional support utilities for appending vectors to specific rows when building `zPoints` (points including context) or recovering `xPoints` (points consisting only of optimizable dimensions). A naming convention was introduced to avoid confusion between these two representations: variables beginning with x denote points or matrices containing only optimizable variables, while those beginning with z contain the fully expanded points passed into the Gaussian process, acquisition criteria, and evaluation routines.

Multiple parts of the core optimization loop required adjustment to use these transformations correctly. During *initializeOptimization()*, the initial generated samples are produced as `xNext` vectors, containing only optimizable dimensions. Before inserting them into the **Gaussian process** dataset, each sample is expanded into `zNext` by injecting the contextual values according to the mask. Likewise, in `stepOptimization()`, the query generated by the acquisition optimizer is originally an `xNext` vector, and before evaluation the context is appended to obtain the final `zNext` passed to `evaluateSample()`. The `nextPoint()` routine was corrected so that candidate points are generated exclusively over the optimizable dimensions. Previously the dimensionality incorrectly included context variables. To guarantee correctness, assertions were added that verify every generated sample has dimensionality equal to mDims minus the context length or the full mDims where applicable.

Special care had to be taken in routines involving coordinate remapping. Because context values lie outside the normalized [0,1] range used to generate candidate points, attempting to remap the full `zNext` vector would erroneously apply bounding transformations to contextual elements. The `remapPoint()` function is intended only for optimizable variables and is useful for plotting unnormalized values. Therefore, wherever remapping occurs, the point is first reduced to `xNext`, remapped, and then

re-expanded to `zNextWithRemap` for visualization. This modification corrected a noticeable inconsistency in one debugging test, where an optimizer targeting the range [–10,10] incorrectly returned a solution near 0.95 instead of near 9 due to context dimensions being included in the remapping calculation. These same corrections were applied in `getFinalResult()`, ensuring that the minimum point is properly decomposed, remapped, and reconstructed with its contextual values.

Additional fixes were required to ensure the evaluation pipeline also respected the split between optimizable and contextual dimensions. In `evaluateSampleInternal()`, earlier versions of the code attempted to remap the entire query vector, inadvertently transforming contextual values. The revised implementation removed the contextual dimensions before calling `remapPoint()`, then reattaches them afterward so that the evaluation sees the correct augmented input vector.

The combined effect of these modifications is a clear separation between the operational space of the optimizer and the full evaluation space of the Gaussian process. Optimizable variables are sampled, remapped, and bounded strictly within the domain defined by the problem, while context variables remain fixed and propagate through the system transparently. These changes required careful structural adjustments across multiple modules (mainly **BayesOptBase**, **BayesOptContinuou**), the plotting utilities, and the evaluation pipeline, but collectively enable **BayesOpt** to model contextual functions without altering the fundamental optimization algorithm. The resulting design remains fully non-context compatible while introducing a robust and flexible mechanism for context-conditioned Bayesian optimization.

5.2 BayesOpt execution flow

The first thing that happens when instantiating a **BayesOptBase** descendant, such as the **ContinuousModel** we use, the constructor begins by calling the **BayesOptBase** constructor with the provided dimension and parameters. Here the logging and verbosity settings are initialized, after which the seed initialization occurs as well as the initial sample count if applicable. The posterior model is created and the randomized sampler is passed onto the **SPSA** sampler. Here the **MCMC** values and flags also initialize, as well as the Unscented Transform object which is linked to the surrogate model.

Then the **ContinuousModel** constructor continues by setting up the

ExpectedImprovement callback for the optimizer, the **NLOPT** inner-optimizer with the full dimensionality (before context), and establishes a bounding box with normalized bounds. Now we have a fully-initialized empty optimizer with no data.

After this, `setContext()` would be called (if we want context) on the continuous model before `optimize()`, then the stored context vector and mask are saved and we need to shrink the internal optimizable dimensionality (by default we need to initialize the model with all dimensions before we can specify the context). The inner **NLOPT** optimizer and bounding box have to be reset and shrink their dimensions as well.

When calling `optimize()`, the optimize procedure begins by checking whether there is a saved state to support the **BayesOpt** functionality of loading and saving optimizations. If there is a load state and loading succeeds everything is resumed from the last point of the previous optimization. Otherwise, `initializeOptimization()` is called, which samples a defined number of random points (of optimizable variables dimension since the context values are fixed) according to the initial sampling method (LHS, Sobol, random) and fills the `xPoints` matrix. These `xPoints` need to be expanded into `zPoints` according to the mask and loads them to the surrogate process (Gaussian Process, for example). If the surrogate identifies any unevaluated samples then it performs `evaluateSampleInternal()` and adds them to the model. Once all initial samples are evaluated, the surrogate model fits itself, at which point it is ready for acquisition-based optimization.

For every `stepOptimization()` one new sample is produced and updates the surrogate. If there is a stagnation or epsilon-greedy exploration, a random jump occurs, in which case `xNext` is generated from uniformly sampling the optimizable subspace. If not, `xNext` is obtained from the acquisition search or MCMC-based method. Once `xNext` is sampled, it must be expanded to `zNext` for evaluation (the black-box expects full dimensionality). After evaluation, the `yNext` is added to the model, the surrogate updates and the best-point debug vector is updated. If hyper-parameter retraining is due (based on `n_iter_relearn`), the model performs a re-fit at the end of the step.

After all iterations of `stepOptimization()` are completed and `finalizeOptimization()` is called the method optionally performs a polishing stage (if `n_final_samples` > 0). Polishing means running a local inner optimization around the current best point, refining the minimum through multiple local

samples. The samples are updated and added to the surrogate. After polishing the hyper-parameters are updated and the surrogate refits a final time. Finally, `optimize()` retrieves the best result, which is in full dimensionality and with the optimizable dimensions remapped to the lower-upper bounds range instead of the normalized 0 1 that is used internally.

5.3 Building the Python Interface with Cython

After validating the C++ extensions, a new Python interface was developed using Cython to bridge between Python and C++. Cython enables seamless interaction between Python and C++ code by compiling Python-like syntax into C-extensions. It supports C++ data types, external function declarations, and performance-critical routines. An interface snippet shows how the contextual model is exposed to Python:

```
from libcpp.vector cimport vector
from libcpp.pair cimport pair

cdef extern from "bayesopt/bayesopt.hpp" namespace "bayesopt":
    cdef cppclass ExperimentalContModel:
        ExperimentalContModel(int, eval_func, void*, bopt_params) except +
        void initializeOptimization()
        void stepOptimization()
        void setContext(vector[double], vector[int])
        ...
```

Listing 5.1: Excerpt from the Cython interface exposing context functionality

The resulting Python-facing class, `InteractiveContModel`, provides full control over the optimization process. Users can initialize, step through iterations, set or change the context dynamically, and access or modify internal data such as past observations (`X`, `Y`) and the surrogate model.

Additionally, the interface was standardized by replacing `boost::ublas` vectors with `std::vector`, resulting in a cleaner and more maintainable Python interface.

5.4 Interactive and Automated Optimization Interfaces

Two main execution paradigms were introduced:

– **Batch Optimization Interface:** A high-level function allows users to perform a complete optimization run from start to finish, given the problem function and parameter configuration.

– **Interactive Optimization Interface:** A lower-level interface exposes the optimization process step-by-step, offering granular control over each iteration, context updates, and dataset management.

The interactive interface proved essential in experiments involving dynamic or segmented contexts. For example, users could load previous optimization states, append new data manually, and re-evaluate the model with different contexts.

5.5 Demonstrations and Developer Support

To facilitate adoption and future development, several demonstration scripts were created:

– `test_interactive_api.py`: Demonstrates the full lifecycle of a toy optimization problem using the interactive interface.

– `test_interactive_setgetapp.py`: Explores valid and invalid usage patterns when setting, retrieving, and appending values to the Gaussian process.

These examples serve both as tutorials for new users and as documentation for future contributors extending the context-aware capabilities of the library.

5.6 Integration into Real-World Pipelines

A motivating use case for this work was the optimization of EEG-based sleep intervention models. Early in the project, it became necessary to adapt existing code to a new data format introduced by upstream providers. This change enabled the processing of smaller segments of the night, aligning well with the context-aware optimization framework.

After completing the Python interface, it became clear that a modular, configuration-driven approach would be necessary to handle the many tests and variants required. Thus, a lightweight domain-specific language was implemented in Python to parse and execute optimization scripts described in text files. This system supports:

– Sequential processing of optimization commands

– Real-time progress logging

– Backup and resume functionality in case of interruptions (e.g., power outages)

This automation system not only accelerated experimentation but also enhanced robustness. Though developed for a specific use case, it offers general functionality that could be adapted into the core `BayesOpt` library for broader usage.

5.6.1 Automated Test Execution and Result Visualization

Given the large number of experiments and the time-consuming nature of Bayesian optimization runs, a comprehensive automation infrastructure was developed to streamline the testing and evaluation process. This system enables fully unattended execution of optimization experiments, minimizing manual intervention while providing robustness against unexpected interruptions, such as power loss or system crashes.

A set of modular shell scripts was created to launch different types of tests using configurable arguments. Each script supports parameters to select the subject, night, and the specific repetition index from which to resume execution, allowing experiments to be paused and resumed efficiently. The following scripts encapsulate the different testing protocols:

– `ARX_full.sh <AR_index: 1-6> <night_index: 0-2> <start_i: 1-5>`

– `ARX_incremental.sh <AR_index: 1-6> <start_i: 1-5>`

– `ARX_multinight.sh <AR_index: 1-6> <start_i: 1-5>`

– `ARX_multinight_short.sh <AR_index: 1-6> <start_i: 1-5>`

– `ARX_partial.sh <AR_index: 1-6> <length: [025, 05, 075, 100]> <start_i: 1-5>`

This setup allows sequential execution of multiple experiments while ensuring efficient use of computational resources. The check-pointing mechanism ensures that no completed work is lost and avoids redundant recomputation.

In addition to execution automation, several Python scripts were developed to support the analysis and visualization of results. These include:

– A converter to transform `.npz` save files into human-readable `.json` format, facilitating debugging and inspection.

– A tool to aggregate and export optimization results into Excel format for structured review and comparative analysis.

– `plot_optimization_mean.py`, which produces the evolution of the best observed score across steps with 95% confidence intervals for each subject.

– `plot_optimization_scores.py`, which generates detailed plots of all score components (e.g., true positives, false positives, detections) for in-depth evaluation.

Together, these tools form a robust framework that not only simplifies the large-scale execution of optimization experiments but also ensures reproducibility and clarity in the subsequent analysis. This infrastructure significantly accelerated experimentation and is envisioned as a reusable contribution to future developments within the BayesOpt ecosystem.

Chapter 6

Experiments: Capitalizing on the BayesOpt new context capabilities

6.1 EEG-Based Sleep Optimization

In this chapter, we present a practical use case for **Bayesian optimization** through the BayesOpt library using real-world EEG data collected from sleep clinic patients. The dataset comprises EEG recordings from six subjects, each of whom participated in sleep studies across three separate nights, with the exception of Subject 4, who was only recorded on two occasions. Furthermore, we averaged the results of our experiments among 5 randomized runs. Each session involved the use of an EEG-based auditory feedback device designed to improve sleep quality. This device dynamically responds to brainwave activity and has been associated with potential health benefits such as seizure mitigation, enhanced memory retention, and improved overall restfulness.

The device is configurable via several hyper-parameters that modulate its response to EEG signals. Among these using previous results, three have been identified as particularly influential: $\tau_{\max}$, κ_{pll}, and κ_{nco}, where τ_{max} corresponds to the maximum amplitude threshold and $(\kappa_{pll}, \kappa_{nco})$ are the *phase-locked loop* parameters following the slow wave detector from Ferster et al. [8]. The optimization bounds for each hyper-parameter are: $\tau_{max} = [-80, -20]$, $\kappa_{pll} = [0.1, 1.0]$, and $\kappa_{nco} = [0.8, 10]$.

To benchmark the performance of these hyper-parameters we have been provided with an EEG wave analyzer that returns multiple metrics: The circular mean absolute error with target angle 45^{o} (CMAE_{45}), calculated as Equation 2.1. The percentage of active stimulations (PAS) in the SW up-phase during NREM sleep ($\text{PAS}_{\in \text{UP}}$), calculated as Equation 2.2. The PAS in undesired SW phases ($\text{PAS}_{\notin \text{UP}}$), calculated as Equation 2.3. All these metrics are incorporated into the score function as a weighted mean as seen in Equation 2.5. Normally the weights would be calculated as a regularization factor based on the measured score with good default hyper-parameters. However, to be able to compare the results among experiments we have decided to

use fixed weights calculated as an average of the regularization factors among the first nights of all six subjects. As a result, the selected fixed weights are: $\alpha_1 = 0.502$, $\alpha_2 = 39.995$, $\alpha_3 = 186.681$.

The BayesOpt library offers different optimizers and can be configured in multiple ways. In our case, we use the continuous model with the expanded contextual capability, which optimizes a set of continuous variables while also being fed a set of contextual variables. The surrogate model uses a standard Gaussian process, where the hyper-parameters are $\mathbf{w}$ and σ_s^2. It also uses a Matérn kernel with $\nu = \frac{5}{2}$ using on Automatic Relevance Determination (ARD), which means that it uses independent parameters for every dimension of the problem. Therefore, it can be used to find the relevance of each feature in the input space. The posterior distribution of the model, which is necessary to compute the criterion function, cannot be computed in closed form if the kernel hyper-parameters are unknown. Thus, we need a find to approximate this posterior distribution conditional on the kernel hyper-parameters The algorithm we use is Markov Chain Montecarlo, which estimates the posterior distribution with a predefined number of random samples as opposed to computing a point estimate of the hyper-parameters based on some score function and using it as a "true" value . The exploration-exploitation trade-off is governed by the Expected Improvement (EI) acquisition function, which involves computing how much improvement we expect to achieve if we sample at a given point.

6.2 Taking a look at the hyper-parameter influence

Table 6.1 summarizes the results of baseline optimizations performed on the first night of sleep recordings for each of the six subjects. These optimizations were carried out without the use of contextual information from previous nights. The aim was to determine the best-performing values for $\tau_{\max}$, κ_{pll}, and κ_{nco}.

Each optimization includes analysis of the *importance* associated with each parameter (originally taken from the length-scale in log-space, then converted), which reflects the sensitivity of the objective function to variations in each dimension. A smaller length-scale value in normal space implies higher importance in the optimization landscape.

The metrics derived from the optimal parameter values include: total detections and stimulations performed by the EEG system, as well as the number of *true positives*

($\text{PAS}_{\in \text{UP}}$) and *false positives* ($\text{PAS}_{\notin \text{UP}}$). These values contribute to a composite score function optimized during training. The goal is to maximize detections, stimulations, and $\text{PAS}_{\in \text{UP}}$, while minimizing $\text{PAS}_{\notin \text{UP}}$.

Subject	**Optimized Params**			**Importance**			**Metrics**				
	$\tau_{\max}$	κ_{pll}	κ_{nco}	$\tau_{\max}$	κ_{pll}	κ_{nco}	CMAE_{45}	$\text{PAS}_{\in \text{UP}}$	$\text{PAS}_{\notin \text{UP}}$	*det*	*stim*
AR1	-65.000	0.775	3.100	75.3%	14.1%	10.6%	0.122	0.0154	0.0004	1117	352
AR2	-43.523	0.840	4.846	60.7%	30.8%	8.5%	0.134	0.0124	0.0015	1121	349
AR3	-65.000	0.775	3.100	70.4%	13.1%	16.5%	0.169	0.0252	0.0019	1741	683
AR4	-53.027	0.897	5.557	48.0%	22.3%	29.7%	0.138	0.0163	0.0013	985	395
AR5	-65.226	0.579	10.000	73.5%	25.7%	0.8%	0.089	0.0114	0.0008	1210	207
AR6	-70.179	0.509	3.637	66.5%	22.3%	11.3%	0.145	0.0286	0.0024	2423	636

Table 6.1: Optimization results for the first night of each subject without contextual information. Importance is inversely proportional to the length-scale, indicating which parameters most influenced the surrogate model. Metrics show the quality and effectiveness of the optimal parameter sets.

As seen in Table 6.1, a few notable trends emerge from this initial batch of optimizations. While some subjects such as AR3 and AR6 exhibit higher overall detection and stimulation counts, $\text{PAS}_{\in \text{UP}}$ remains relatively low across all subjects fluctuating between 1.1% and 2.8% although importantly far above $\text{PAS}_{\notin \text{UP}}$ which oscillates between 0.04% and 0.24%. This is expected, given the extremely limited sampling budget: only 32 evaluations are used to explore a potentially complex and high-dimensional function. With such sparse coverage of the search space, the surrogate model (a Gaussian Process in this case) may not have enough information to accurately approximate the underlying score landscape, especially near the true optimum. Looking at the influence of the parameters, $\tau_{\max}$ appears to exert the most influence on the outcome, followed by κ_{pll} and κ_{nco} though this trend varies slightly between subjects and will warrant further investigation once context is introduced in subsequent tests.

6.3 Optimizing the hyper-parameters. Night variation and cost

Each optimization process consists of an initialization phase followed by iterative exploration. During initialization, a predefined number of evaluations (*8 by default*) is performed to inform the surrogate model used by BayesOpt of the general layout of the objective function using a sampling function. We use Sobol, which divides the search space into an s-dimensional hypercube where the sequence of sampled points is mapped into the hypercube in a uniform manner. This guarantees the best possible coverage for any numbered of sampled points. Since the objective function is very

expensive to sample we have selected 8 steps as a compromise between cost and initial exploration of the 3D hyper-parameter space. The number of samples varies among sessions depending on the length of the recording, which directly affects the processing time, as can be seen in Figure 6.1. As shown, processing an entire night costs or average around 100 seconds per step. The fact that the objective function is so expensive to sample makes the Bayesian optimization algorithm a very good match for the problem, excelling in being able to converge with very few samples compared to the alternatives.

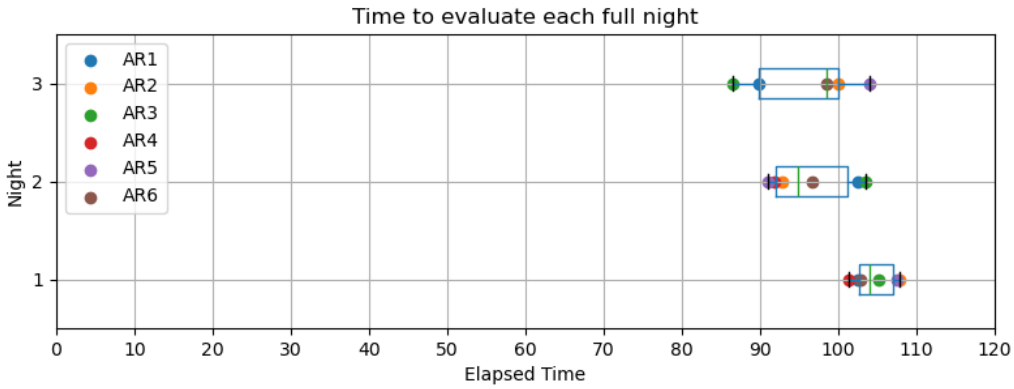


Figure 6.1: A visual representation of the time required to process each session (the session lengths are longer than the 100% of NREM windows from Figure 6.4). The vertical axis shows the different nights from each subject (in different colors). On the horizontal axis we see the time in seconds necessary to perform one evaluation. The box plot shows the variation in processing time for each night, which is directly correlated with the length of the recording. We can see that the subjects don't sleep the same amount of time each night but at the same time there isn't a huge variation, and seem to sleep more on the third night.

As a baseline we performed an optimization of the full nights using 8 initialization and 32 evaluation steps. In Figure 6.2 we can see the results of these optimizations for the first nights of the 6 subjects across 5 different runs. Following the results from Figure 6.1, this entire experiment takes 33 hours of compute to complete (100 seconds * 40 samples * 5 runs * 6 subjects), which emphasizes how effective Bayesian optimization is at converging even with very few samples. We can also see that across runs and subjects our confidence interval is very wide. This can be explained by the variance of the subjects alone which all experience different nights as seen in Figure 6.3, where the observed variance across runs is minimal.

One strategy to reduce the time needed to perform one evaluation is optimizing the hyper-parameters using only partial night data. For the first night of each subject we have been provided with information of the location of the NREM windows in ranges of samples. With this information we are able to figure out how the number of samples

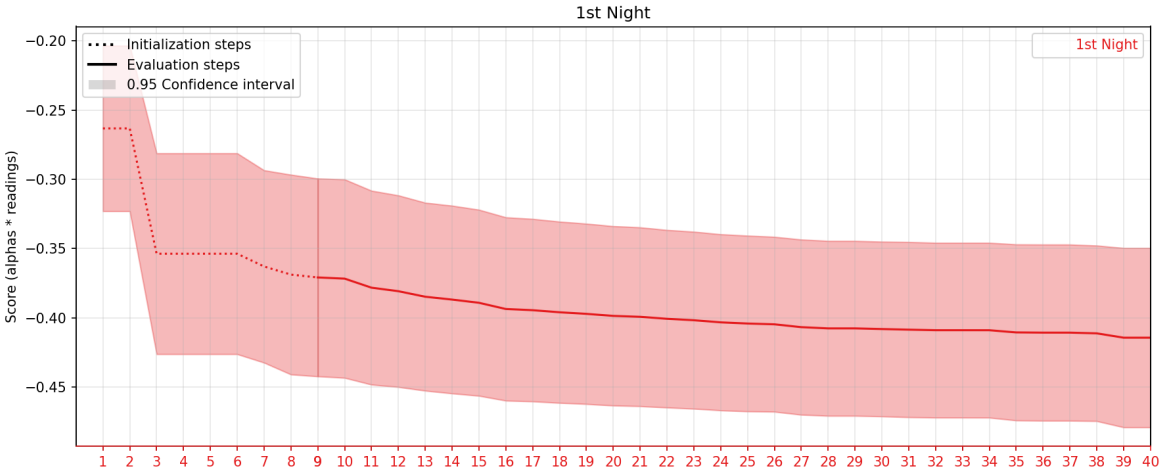


Figure 6.2: Score function across steps from all subjects for the 1st night. The line represents the running minimum of the score and the transparent area is the 95% confidence interval resulting from averaging the subjects and runs. We observe on initialization (dotted lines) how sampling randomly we arrive at -0.37 as our starting point, and through the optimization the algorithm keeps converging at a steady pace all the way to -0.42.

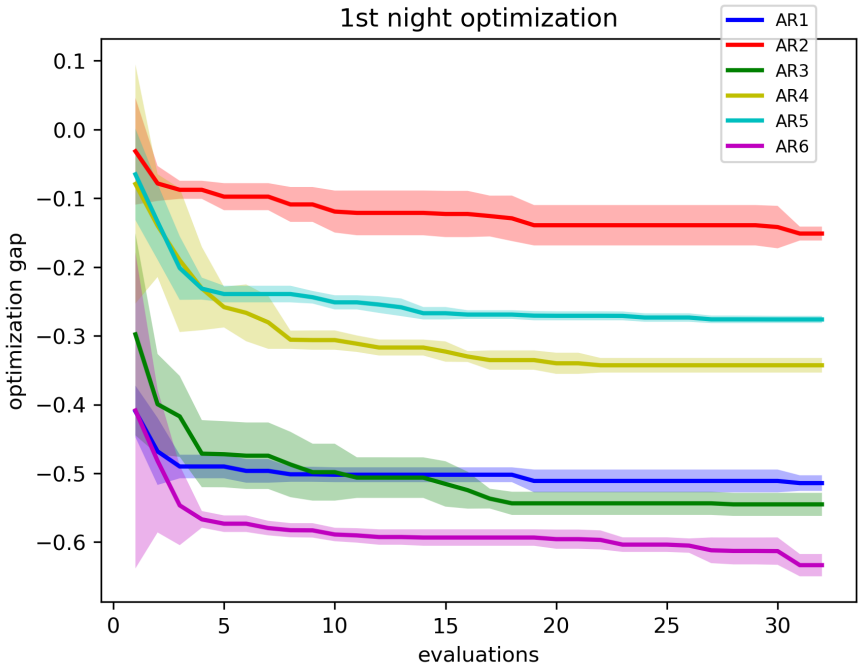


Figure 6.3: Score function across steps from all subjects for the 1st night. Each line represents the running minimum of the score for a different subject and the transparent area is the 95% confidence interval from averaging the runs. Each subject experiences a different night, which makes the optimizations converge very differently among subjects, but very similarly across runs.

directly affects the processing time per evaluation. We have decided to split the night in sizes of 25%, 50%, 75% and 100% of the NREM windows, so that we are sure not to split in the middle of a relevant night stage. In Figure 6.4 We can see that the processing time is linearly correlated with the number of samples. We can also see that the 25%, 50%, 75% and 100% splits are not going to take the same amount of time to process for each subject, but in the following experiments we are going to assume that when averaging the subjects, the 25% split has a quarter of the cost of the full night and so forth. This is to simplify the results and conclusions, which hold true regardless of the specific data variations.

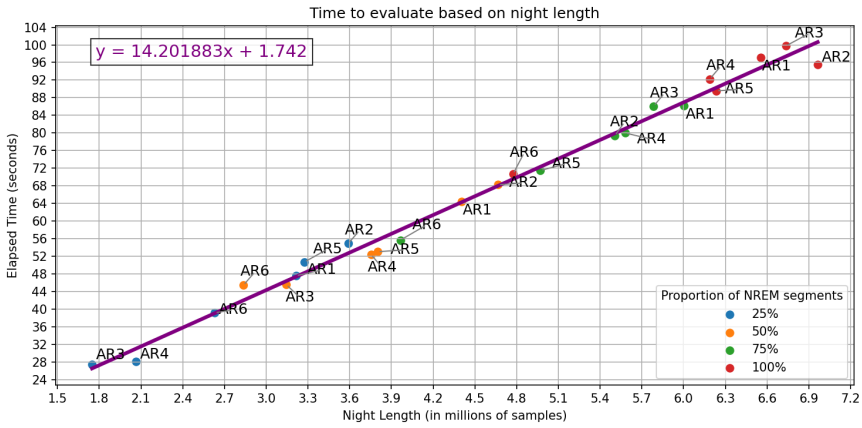


Figure 6.4: Plot showing the processing time of one evaluation (in seconds) based on the night length (in millions of samples). Each Proportion of NREM samples is shown in a different color. In purple we can see the linear regression line and formula, which indicates that the correlation is linear (with little noise that could be attributed to the core scheduler of the machine). We can also see that it takes on average 14.2 seconds to process a million samples and that 1.7 seconds is the cost of running BayesOpt.

6.4 Cost-saving strategies for optimization

As our next experiment, we decided to find out if we can get close to optimizing the objective function for each of the sessions while only utilizing a fraction of the night. We have plotted the achieved scores utilizing the first 25%, 50%, 75% and 100% of the NREM windows against the full first night. Since there is a difference in the number of samples we explore in 25% of the night vs the full night, we are going to give the optimization of the full night 32 evaluation steps to fully map the gaussian space, and we will only give the partial optimizations 8 evaluation steps. Contrary to our assumption, the results in Figure 6.5 indicate that the optimization of partial night data converges to lower minima than when using full night data.

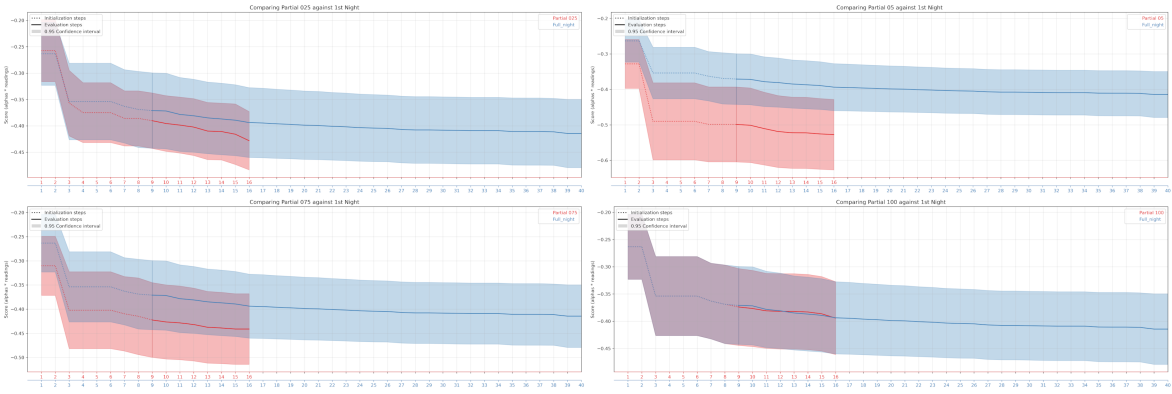


Figure 6.5: Plots of partial night optimizations with 8 evaluations (red) against the full night optimization with 32 evaluations (blue). Both have 8 initialization steps using sobol sampling. On the horizontal axis we have the step count while on the vertical axis we have the running minimum (lowest of the previous, lower is better). The different night partials are made of 25% (top left), 50% (top right), 75%, (bottom left) and 100% (bottom right) of the NREM samples.

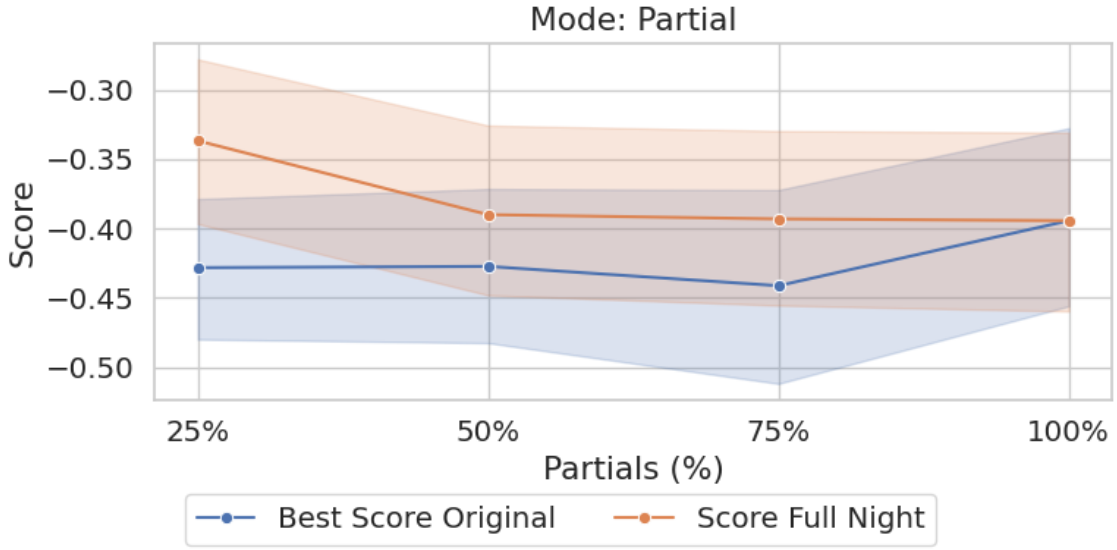


Figure 6.6: Score comparison for the optimized hyper-parameters for different night batches. In blue we have the optimum score obtained by analyzing 25, 50, 75 and 100% of the NREM samples and in orange we have the score obtained by analyzing the same hyper-parameters with all the samples. The blue line suggests that the more samples we add the worse minima we get, while the orange line indicates that the more samples we add the better the scores. The 100% NREM batch doesn't change since the number of samples is almost identical to the full night.

This can be explained because the partial optimizations overfit the hyper-parameters to minimize the scores of their specific batch of samples. When performing the evaluation of the best hyper-parameters for each partial experiment using all the night samples, we are able to reach the same conclusions, as seen in Figure 6.6. If we also take into account the time taken to evaluate each batch, it seems that the sweet spot between scores, generalization and processing time is the 50% batch.

We have observed in the previous experiment that we can take advantage of feeding more specific data to the model to save computing time at the detriment of some model accuracy and generalization, which by no means is a linear tradeoff. By choosing a sensible batch size we are able to minimize computing time for a small penalty. However, deciding the region with the least and best number of samples for jump-starting our optimization might not be apparent nor available at the time of processing. For example, in our case, it could be meaningful to adjust the hyper-parameters in real time, to directly tune the device to achieve better scores using only the data from the previous samples, instead of doing the processing after the fact.

6.5 Using context to guide the optimizations

6.5.1 Incremental segment size optimization

An option to make the partial batch size approach more generalizable is to build our optimization starting with a small batch size and continue increasing the batch in small increments, using contextual information as the way for BayesOpt to generalize across them, since the scores for the same hyper-parameters would differ. The next experiment is based on performing 8 optimization steps for the 25% sample batch, then continuing with 8 more for 50% of the night, then 8 for 75% and another 8 for the 100% of NREM windows. Each chunk of the experiment will have one dimension for the context, that should allow the Expected Improvement algorithm to use information from previous batches to make a more informed decision of where to sample the next point and reach lower minima. We will compare the score from this experiment with the previous experiment as a baseline which does not take advantage of the context and see the results.

In Figure 6.7 we can see that the first chunk of our contextual approach yields the same results as the baseline, since our experiments are seed normalized, and there is no previous context to use. In Figure 6.8 we see that it takes some steps for the model to infer the connection between the two sample sizes, and the final scores are very similar

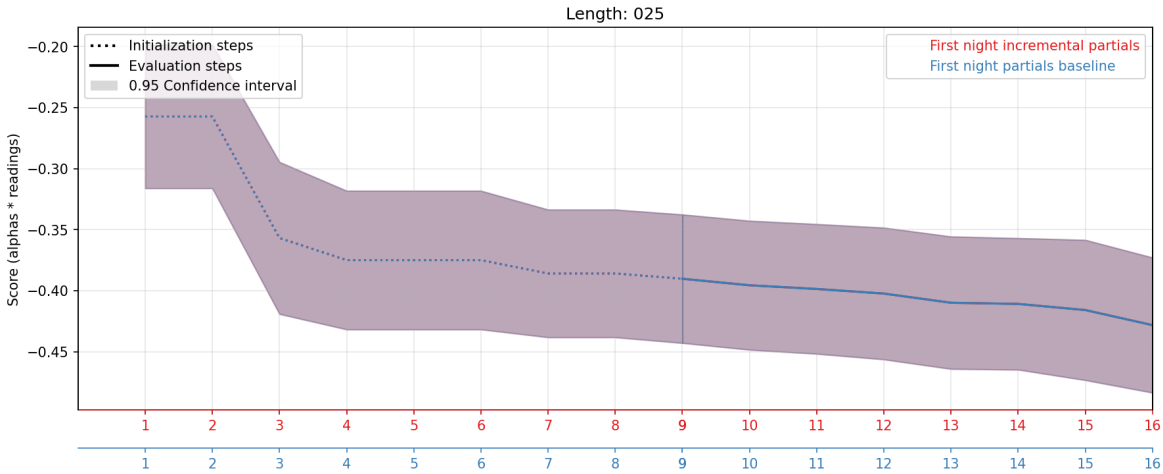


Figure 6.7: Running minimum scores across steps for the first night and the 25% batch. In red we use context to build knowledge as we increment the batch size. In blue we plot the baseline of optimizing with the 25% batch without contextual dimensions. The plots from red and blue are aligned since there is no previous contextual data to build on, and as a result it looks purple. The purple area is the 95% confidence interval result of averaging the subjects and the runs.

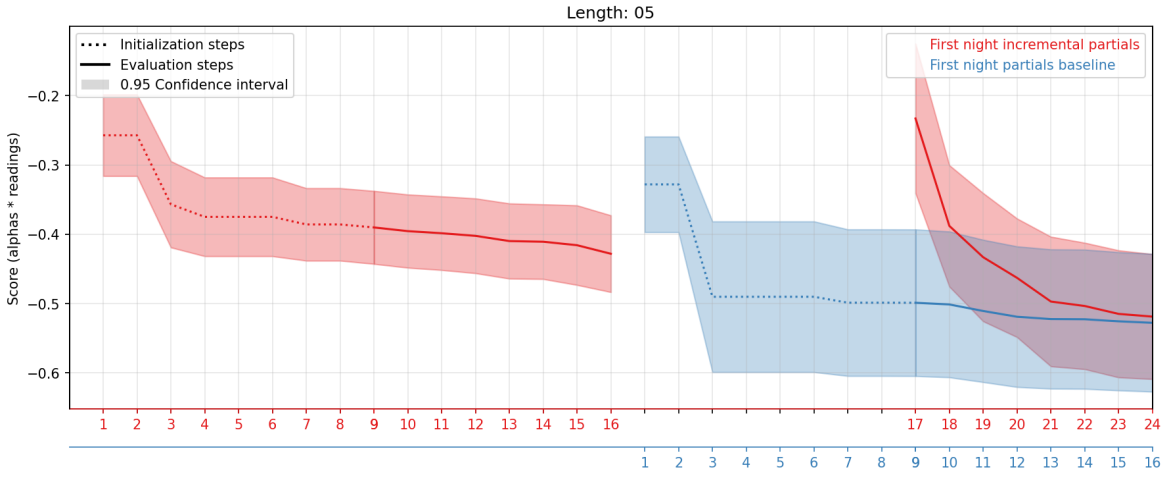


Figure 6.8: Scores across steps for the 50% batch. The lines are the running minimum (dotted for initialization and continuous for evaluation) and the areas are the 95% confidence interval. In red we see the incremental experiment, where the first 16 steps belong to the 25% batch and the last 8 to the 50% batch. In blue we have the optimization of the 50% batch without context. Using the 25% batch as context the 50% batch optimization is able to reach similar scores to the baseline without initialization.

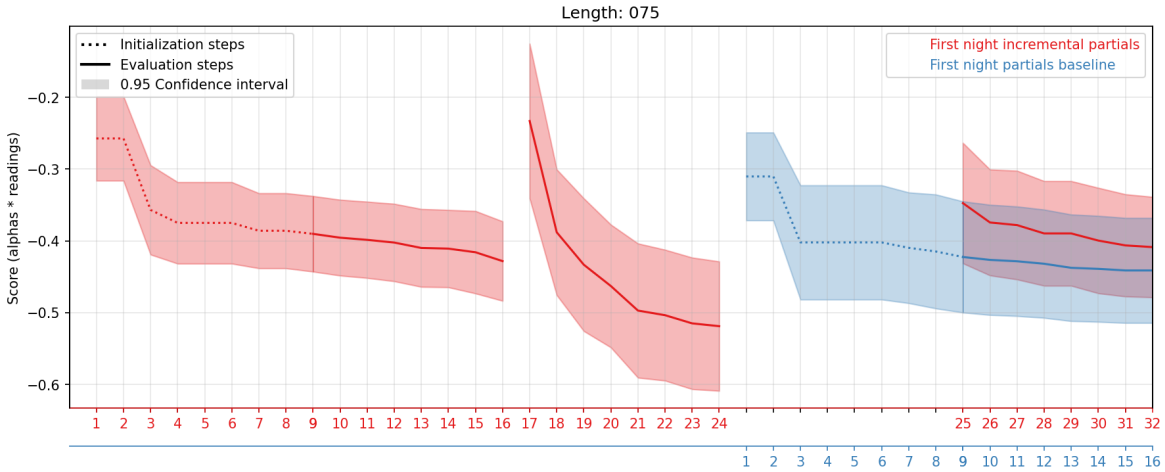


Figure 6.9: Scores across steps for the 75% batch. The lines are the running minimum (dotted for initialization and continuous for evaluation) and the areas are the 95% confidence interval. In red we see the incremental experiment, building the 75% batch optimization on top of the 50% and 25%. In blue we have the baseline 75% batch optimization without context. The last optimization is able to get better scores without initialization from the start, but doesn't achieve similar scores to the baseline.

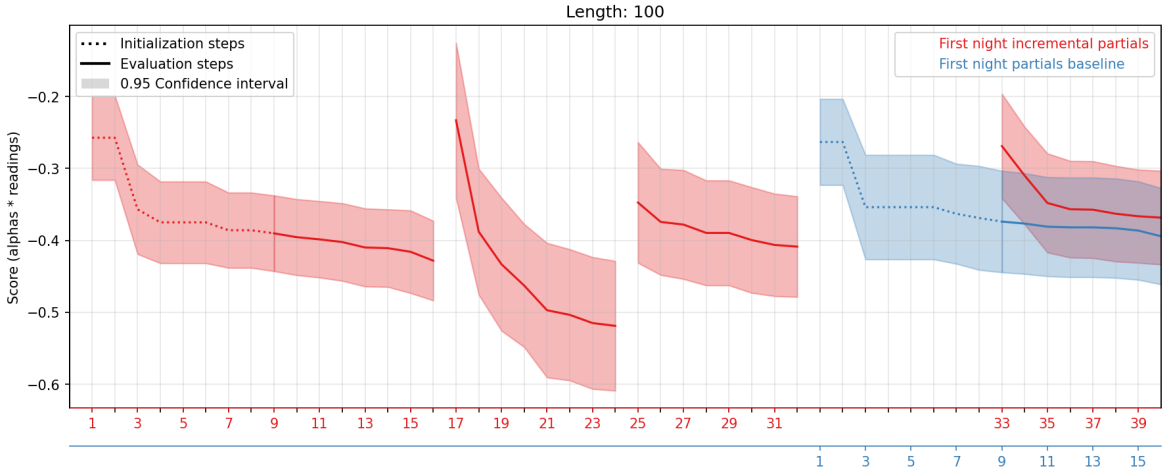


Figure 6.10: Scores across steps for the 100% batch. The lines are the running minimum (dotted for initialization and continuous for evaluation) and the areas are the 95% confidence interval. In red we see the incremental experiment, building the 100% batch optimization on top of the other ones. In blue we have the 100% batch optimization baseline. The last optimization with context doesn't achieve as good scores as the baseline, while not being far off.

when using 50% of the samples. In Figure 6.9 we see that once we have two chunks as previous data the number of steps to generalize from the previous information is lower, but we do not reach the same minima as when not having context and finally in Figure 6.10 it seems that we do not achieve the same minima either, but we get closer than before. If we look at comparing these results against the full night to correctly see the performance of the chosen hyper-parameters, in Figure 6.11 it seems like we get the opposite of achieving greater generalization.

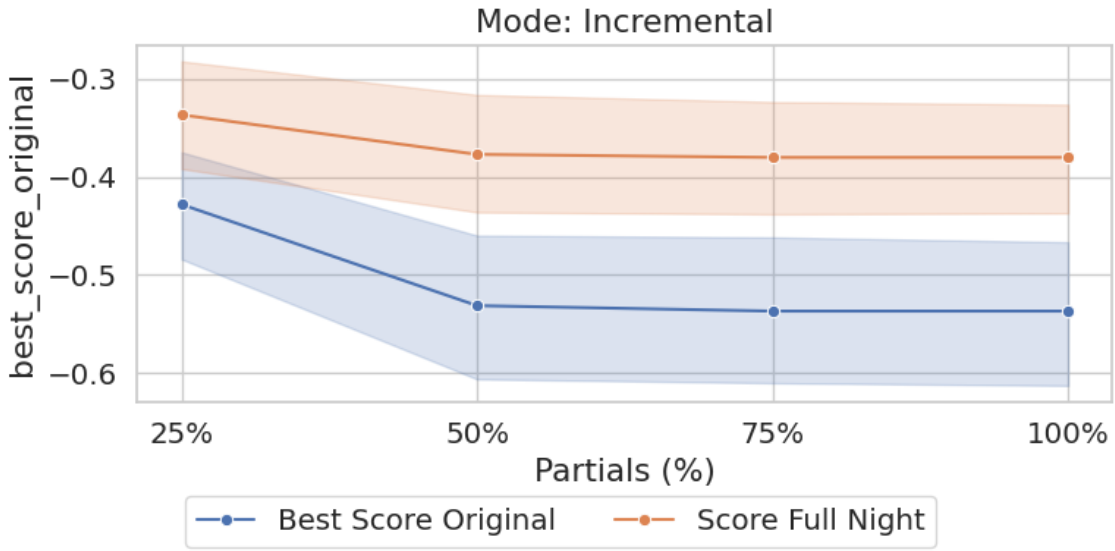


Figure 6.11: Score comparison for the optimized hyper-parameters for different night batches. In blue we have the optimum score obtained by analyzing the 25, 50, 75 and 100% batches from the incremental experiment and in orange we have the score obtained by analyzing the same hyper-parameters with all the samples. Here the blue and orange lines suggest that the more samples we add the worse minima we get. It should be observed that despite the minimum from the last batch optimizations in the plot, here we are analyzing the absolute minimum from all the optimization, which belongs to the 50% batch, which explains the flat blue line in this plot.

There is one benefit to using the incremental approach, which is that it's possible to load a previous optimization and resume it with other points while only costing the compute time of the new ones. This means that for the 100% batches, the incremental starting from scratch would cost $25*16+50*8+75*8+100*8 = 2200$ seconds while the baseline would cost $100*16 = 1600$ seconds; and if we load the previous results and optimize from there it costs $100*8 = 800$ seconds. Starting from zero the incremental approach is 37.5% more expensive, while loading the previous optimizations means a 50% decrease in cost per run.

6.5.2 Multi-night optimization

We now explore the impact of leveraging entire nights of prior optimization as contextual knowledge for subsequent nights. The idea is to use the results of the baseline optimization of the 1st night as context, then perform 32 optimization steps on the 2nd night, and lastly optimize the 3rd night using context from both previous nights. This transfer of contextual information allows us to analyze how knowledge accumulation influences optimization efficiency and convergence behavior. To account for how much the optimization step difference improves the results against the addition of contextual information, we are also going to perform an optimization with 8 steps per night instead of 32. This means that for the three nights the smaller step count multi-night optimization will perform 32 steps (8 initialization + 8 per night), the baseline will perform 40 (8 initialization + 32 for 1 night) and the 32 step multi-night will perform 120 (8 initialization per night + 32 evaluation per night). We are adding initialization to the beginning of all nights for the last one to serve as the best case scenario of the scores we can get and to more fairly compare against the baseline.

Looking at Figure 6.12 we observe how the first night results are exactly the same for all three experiments as there is no context to be used to sample new points. The results from this graph therefore also match the ones from Figure 6.2.

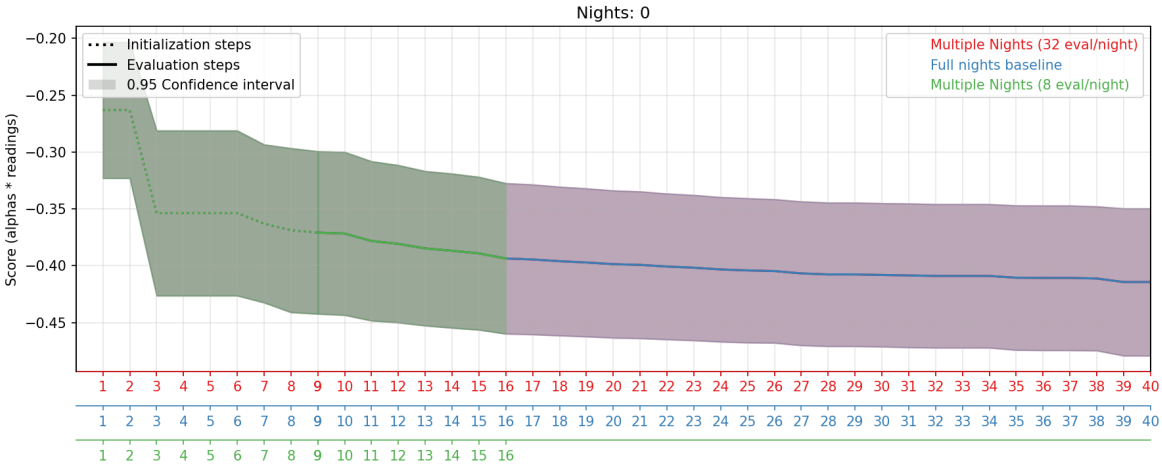


Figure 6.12: Scores across steps for the first night optimization. In red, the multi-night optimization, in blue the non-contextual baseline and in green the multi-night optimization with only 8 steps. All graphs align as there is no context to learn from.

Moving onto the second night optimization in Figure 6.13 we see how the 32 step multi-night leads to the best scores, closely followed by the baseline and trailed behind by the 8 step multi-night. This order also correlates with the number of steps needed

to achieve this scores as the best optimization employed 80 steps, the second used 40 and the 8-step optimization employed 24. If we take advantage of the fact that we can already have optimized for the first night in the multi-night experiments, then the best result takes 48 steps, the second 40 and the last would take only 8. We can see how adding context to the multi-night setups improved the starting point of the optimization, as there is some previous information to guide the sampling, and is further improved on the 32 step setup by using initialization, which is not explained by context alone.

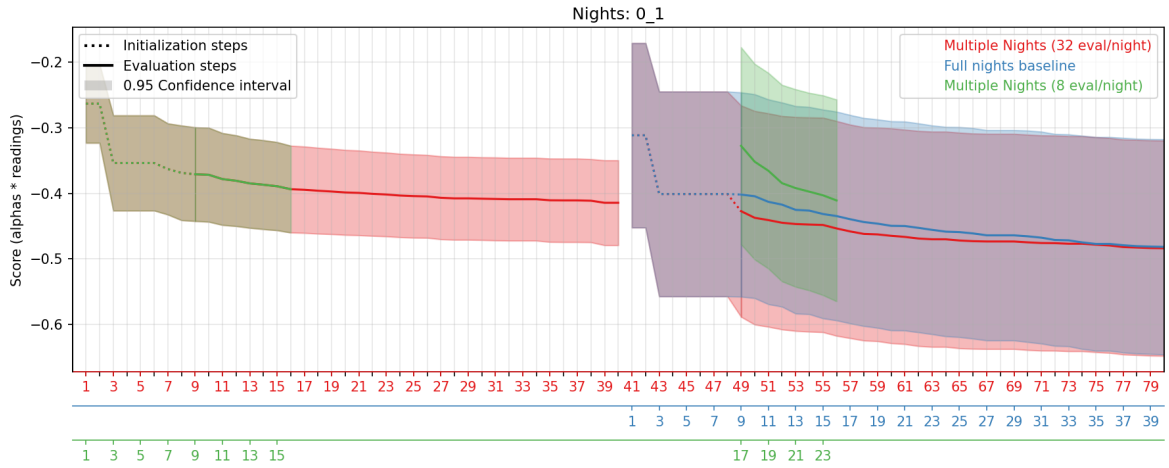


Figure 6.13: Scores across steps for the second night optimization for the baseline (blue), 32 step multi-night (red) and 8 step multi-night (green) optimizations. The first night results coincide with Figure 6.12, and we can see that the second night results have bigger variation across all experiments hinting that this is night-specific. The initialization doesn't use context, so it coincides across experiments. The 32 step multi-night optimization achieves the best scores, followed by the baseline, and then by the 8 step multi-night. This is also on trend with the number of steps.

Moving to the third night, in Figure 6.14 we see that the best performing setups are the 32 step multi-night, then the 32 step baseline and followed much closely this time by the 8 step multi-night setup. We can see that in this case the initialization on the third night by the 32 step multi-night setup makes its starting point be the same as the baseline, as opposed to the 8 step setup which has no initialization at the beginning of the night. However, this time the green line seems to descend much faster compared to the second night, implying that as the contextual information from the points in the model grows, the initialization at the beginning of the nights is less needed. Even if the green line doesn't get to the scores of the red and blue lines, it gets very close to it with a lot less samples. If we measure the number of steps it takes each experiment to achieve its best performance for the third night, the 32 step

multi-night setup takes 120 steps, followed by the 40 steps of the baseline and the 32 of the 8 step multi-night one. Assuming the previous night optimizations are available to resume, the best model takes 40 steps, the same as the baseline, and the 8 step multi-night only takes 8. This is a fifth of the cost for very similar performance.

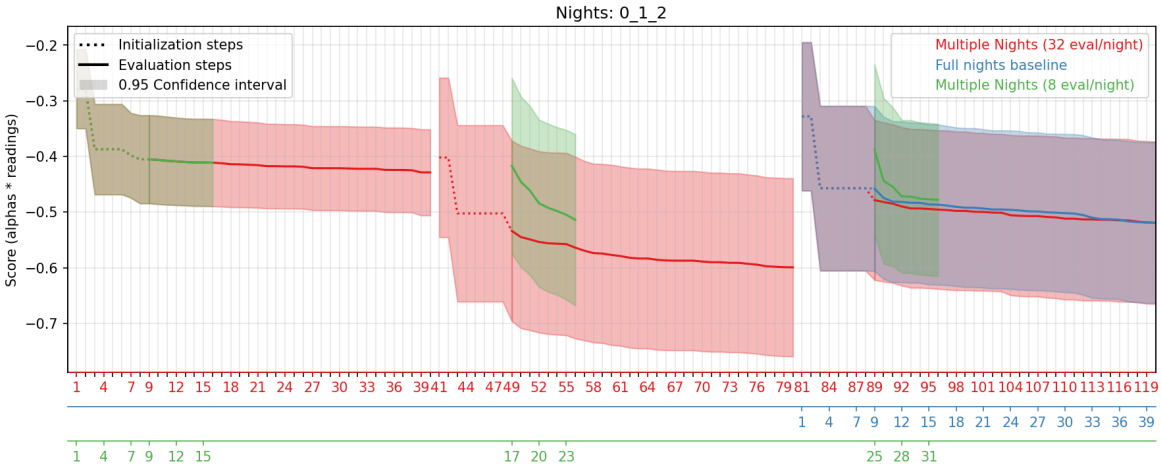


Figure 6.14: Scores across steps for the third night optimization for the baseline (blue), 32 step multi-night (red) and 8 step multi-night (green). The results from the first two nights don't exactly coincide as we are missing subject 4 since it doesn't have a third recording. For the first two night results look at Figure 6.13. The third night shows a similar trend to the second as to the fact that the best results originate from the 32 step multi-night, followed by the baseline, and more closely followed than before by the 8 step multi-night.

The results from the multi-night optimization experiments hint to the conclusion that increasing the amount of contextual data might be more cost effective to achieve a sort of generalizable best hyper-parameters than increasing the amount of steps per night. Sadly, we are limited in the amount of contextual information we can gather as we have only 3 nights per subject. In Figure 6.17, taken from the robotic grasping use case chapter later in the report, which has different object-grasping optimizations added in sequence, the same conclusion can be extracted. For each added object (or night) the context based optimization has a bigger margin over the non-contextual one which is not explained by the specific object being easier to infer using previous data (this is proven by inverting the order of the objects in the optimization).

6.6 External Adoption of Context-Based Optimization: Robotic Grasping Use Case

A compelling use case for the contextual extensions made to the **BayesOpt** framework can be found in the thesis by *Pérez González (2025)*[11], where the optimization of grasp strategies for robotic hands is addressed in a simulation environment using a humanoid robot platform. His work integrates visual features derived from a self-supervised transformer model (DINOv2) with grasp quality metrics into a Bayesian optimization pipeline enhanced with contextual learning capabilities.

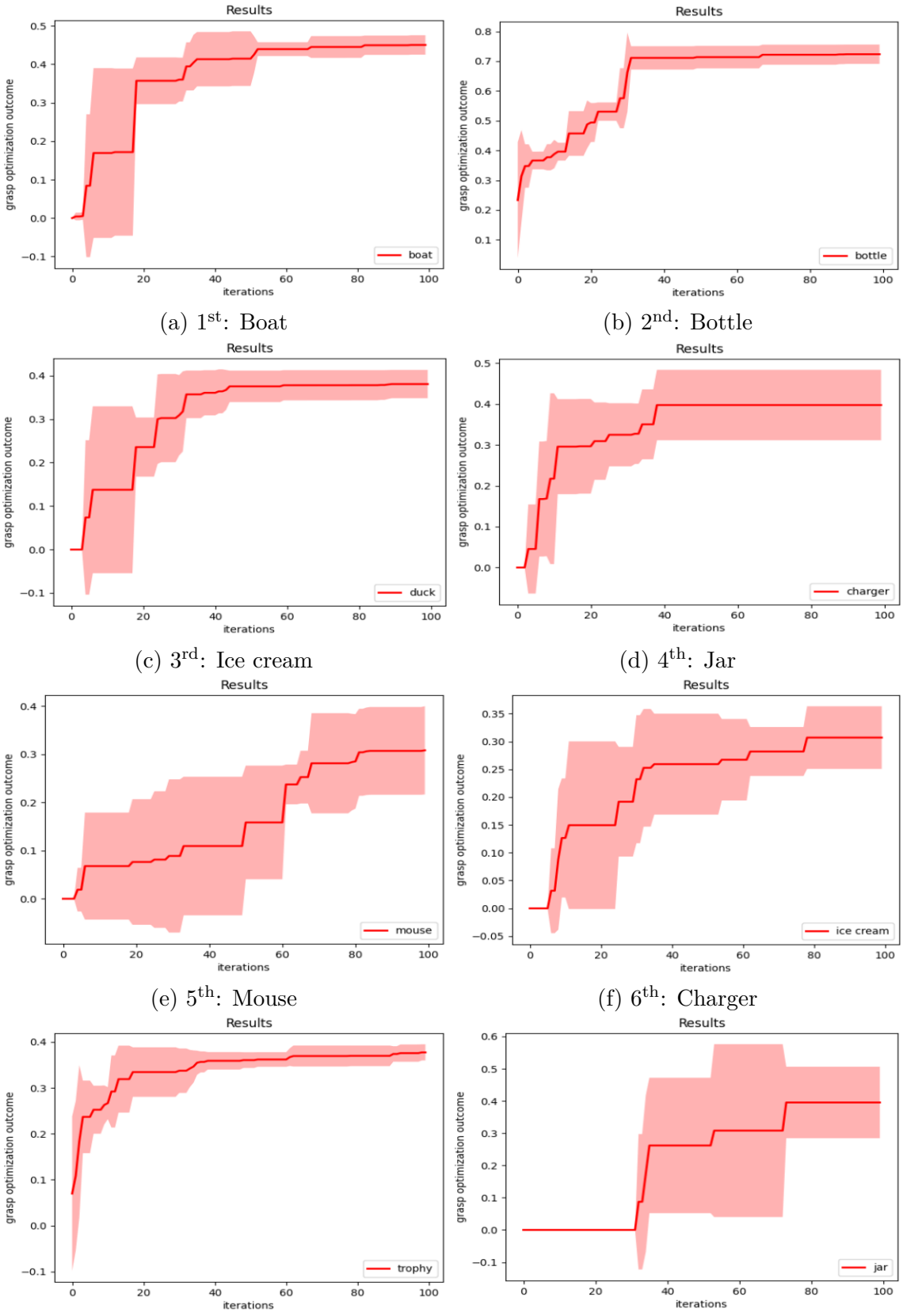
The specific implementation of the optional context enhancement part saw both of us collaborating to improve our respective projects. The findings in the development process of his framework gave some valuable feedback into developing a more user friendly interface for accessing the contextual features of **BayesOpt**, while on my end I helped him get up to speed quicker with using said interface and integrating it smoothly into his framework.

The pipeline is built around an iterative process of context-aware Bayesian optimization. For each object encountered, grasp metrics and visual embeddings are used to derive a **context vector** which combines the average grasp success and latent features extracted via deep kernel learning. These context vectors are then integrated into the Gaussian Process (GP) model to guide optimization in subsequent objects. This method enables the model to efficiently transfer knowledge across different grasping tasks and thereby accelerate convergence for new objects [11].

The implementation relied on using context vectors as additional input dimensions to the GP, as described in Section 2.6.3 of the thesis. At each iteration, the model uses these vectors to augment the acquisition function (Expected Improvement in this case), resulting in an adapted optimization trajectory that accounts for previously learned object characteristics.

A general view of all tested objects is presented in Figure 6.15, where the grasp quality improvements can be seen across different iterations. Notably, while simpler objects such as *bottle* and *jar* reach high grasp quality quickly, complex objects like *duck* and *mouse* benefit more significantly from context to achieve robust performance.

Figure 6.16 compares context-aware and non-context optimization strategies over



(a) 1st: Boat

(b) 2nd: Bottle

(c) 3rd: Ice cream

(d) 4th: Jar

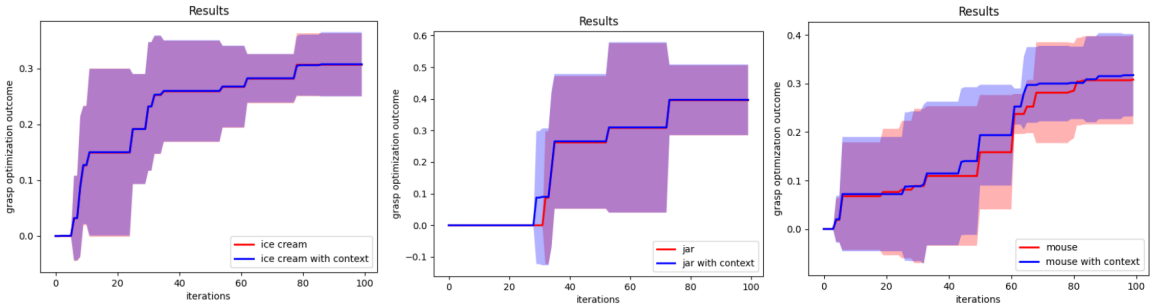
(e) 5th: Mouse

(f) 6th: Charger

(g) 7th: Duck

(h) 8th: Trophy

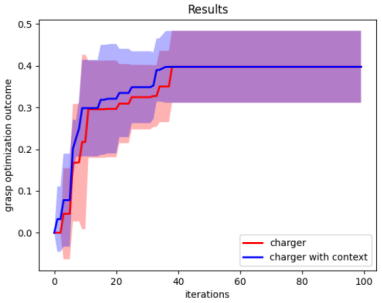
Figure 6.15: Grasp optimization results across objects using context-aware Bayesian optimization. Each subplot shows the quality score over iterations. Source: [11]



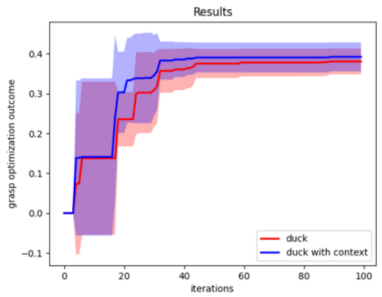
(a) 3rd: Ice cream

(b) 4th: Jar

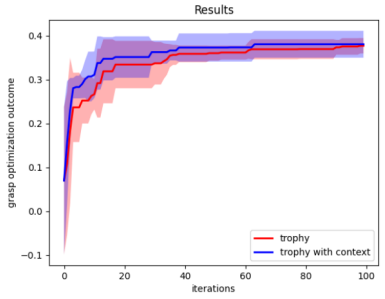
(c) 5th: Mouse



(d) 6th: Charger

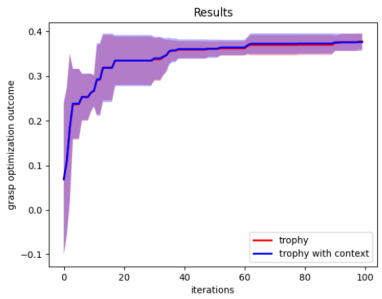


(e) 7th: Duck

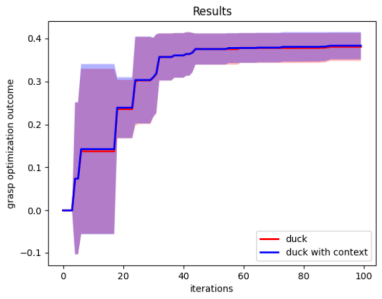


(f) 8th: Trophy

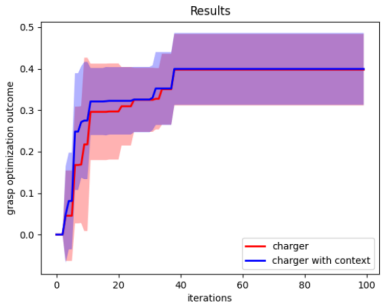
Figure 6.16: Optimization results for Iteration 1: Comparison between context-aware and non-context methods for six objects. Each plot highlights differences in stability, convergence speed, and final grasp quality. Source: [11]



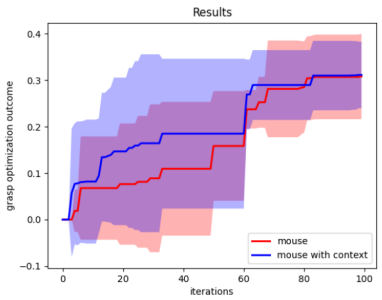
(a) 3rd: Trophy



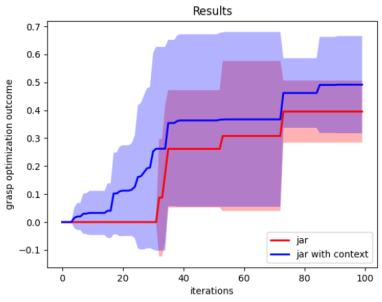
(b) 4th: Duck



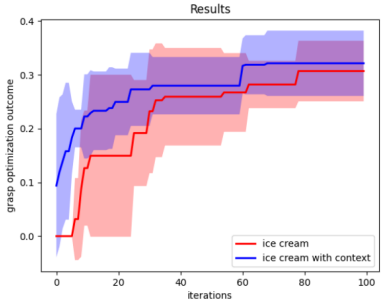
(c) 5th: Charger



(d) 6th: Mouse



(e) 7th: Jar



(f) 8th: Ice cream

Figure 6.17: Optimization results for Iteration 2 (reversed object order). Early-stage objects (trophy, duck) show minimal difference between the methods, while later objects (charger, mouse, jar, ice cream) exhibit clear improvements using contextual information. Source: [11]

six objects during Iteration 1. We observe that context provides benefits in several ways: for simple objects like the *ice cream* and *jar*, it improves stability by reducing the variance in the optimization trajectory. For more complex objects like the *duck* and *trophy*, context not only stabilizes the learning curve but also leads to improved final grasp quality.

A second iteration, shown in Figure 6.17, was conducted using the same set of objects but in reverse order. This sequence allowed the study of how accumulated contextual information impacts performance when complex objects are seen earlier or later in the pipeline. While early-stage objects (e.g., *trophy*, *duck*) showed little difference between the two methods due to lack of prior context, the benefits of context grew more apparent in later stages. For example, the *mouse*, *jar*, and *ice cream* objects in Iteration 2 show markedly better convergence and final performance when contextual information is used.

6.7 Comparing the results from the EEG and grasping use cases

When comparing these findings with the EEG optimization experiments presented earlier, several common insights arise. Both applications benefit from contextual initialization, leading to faster convergence and in many cases better minima. In Pérez González's robotic grasping work, context reduces optimization variance and improves efficiency, particularly in later tasks after sufficient context has been accumulated. This aligns with our observation that loading prior optimizations for previous nights led to flatter optimization curves with lower variance in early steps.

However, a key difference lies in the nature of the context. In the grasping framework, the context is derived from previous object-specific embeddings, which encapsulate learned visual and performance characteristics. In contrast, our application treats temporal partitions of a subject's EEG data as context, which may not always align structurally with the next optimization target. Consequently, we observed that when context is misaligned (e.g., optimizing with partial night samples and evaluating those optimal hyper-parameters against all the samples from the night), it may guide the optimization away from optimal regions, especially under constrained steps.

Furthermore, while robotic grasping showed incremental improvements as more context was accumulated, in our results the availability of new data with different

context was very limited, showing an insight that was not as explored in Eduardo's work, which is that if we increase the variability of the points for a specific context value, and we don't have a lot of them, if we then start exploring a new data segment too different from our available context, the initial iterations may be led away from the optimal compared to a context-less optimization.

Ultimately, both projects validate the power of contextual Bayesian optimization. Yet, they also underline its sensitivity to the quality, quantity, and alignment of the context being used. In other words. The contextual information available should be scaled accordingly to the problem size for it to provide significant improvements over a context-less optimization (a harder optimization problem needs a bigger and more varied contextual dataset before context-based optimization provides better results).

Chapter 7

Conclusions

In our work we have explored the inner workings of the Bayesian optimization algorithm. The fundamental principles of how it optimizes and why it is used in multiple fields like robotics, machine learning and in our case even EEG signal processing. Furthermore, we have studied the BayesOpt library as one of the possible implementations of Bayesian optimization to the point where we have extended its capabilities to be able to exploit contextual information for guiding the optimization. This designing and implementation work has highlighted the challenges and intricacies of managing a complex project with multitude of interconnected modules that need to work together in unison to provide a reliable and bug free platform for others to use. In addition, for ease of use we exposed these new contextual Bayesian optimization capabilities through a **Python** interface from the **C / C++** core of the library, which necessitated learning some basic **Cython** language.

Once we provided a reliable and easy to use contextual module, we explored the optimization of hyper-parameters for EEG signal processing as a use case in collaboration with BitBrain. They provided the Python code for evaluating the performance of a given set of values of hyper-parameters, while we exploited the characteristics of Bayesian optimization to perform optimization over that function treated as a black box. We showed how using Bayesian optimization for this use case was more effective at maximizing the intended behavior (providing correct stimulations during SW up-phases) than an expert making informed decisions of the hyper-parameters based on relevant papers from the field. When we added contextual data to the optimizations, we were able to ascertain the benefit of relating optimization information from different nights and subjects, which influence the objective function and are prohibitively expensive to be explored during optimization; to provide valuable information in subsequent optimizations. The experiments show that there are more and less effective ways to build information through the optimizations, and how it could be worthwhile to expand the value of the information context is able to provide through the use of specialized kernels or different acquisition functions like a multi-task approach. We were also able to show how the contextual expansion

was able to provide value in other fields than EEG signal processing, like the robot grasping use case in Eduardo's master's thesis[11], where subsequent optimizations were able to provide better grasping scores through the use of context. Comparing the findings from these two works, we are confident in saying that the behavior showed that the bigger the quantity and quality of contextual data we are able to provide, the bigger the improvement compared to not using it.

7.1 Future work

Through this work we have shown that Bayesian optimization has multiple fields where it provides meaningful value, like in our use case for improving the sleep of people of age or with sleep disorders. The exploration and documentation of BayesOpt should be used as a jump-start for getting familiarized with the use of this library and expand on several open fields outside the scope of this work. First of all, expanding on the size of the dataset and the number of contextual parameters and values should show a bigger unexploited margin of optimization gains based on the limited size of data we were able to extract conclusions from. The value of the contextual data could also be improved through the use of more specialized kernels that exploit this behavior of some variables that can be sampled and others that are fixed to some recording specifics. Another area of interest for the EEG signal processing use case is real-time processing of the previous night samples based on a big contextually trained model. This could open the window for the commercialization of this product based on a big general contextually trained model being fine-tuned for the specifics of the individual and night.

Chapter 8

Bibliography

[1] Joseph Roland D Espiritu. Aging-related sleep changes. *Clinics in geriatric medicine*, 24(1):1–14, 2008.

[2] Hans Brunner, Thomas C. Wetter, Birgit Hoegl, Alexander Yassouridis, Claudia Trenkwalder, and Elisabeth Friess. Microstructure of the non-rapid eye movement sleep electroencephalogram in patients with newly diagnosed Parkinson's disease: Effects of dopaminergic treatment. *Movement Disorders*, 17(5):928–33, 2002.

[3] Véronique Latreille, Julie Carrier, Marjolaine Lafortune, Ronald B. Postuma, Josie Anne Bertrand, Michel Panisset, Sylvain Chouinard, and Jean François Gagnon. Sleep spindles in Parkinson's disease may predict the development of dementia. *Neurobiology of Aging*, 36(2):1083–90, 2015.

[4] Hong-Viet V. Ngo, Thomas Martinetz, Jan Born, and Matthias Mölle. Auditory closed-loop stimulation of the sleep slow oscillation enhances memory. *Neuron*, 78(3):545–553, 2013.

[5] Nelly A. Papalambros, Giovanni Santostasi, Roneil G. Malkani, Rosemary Braun, Sandra Weintraub, Ken A. Paller, and Phyllis C. Zee. Acoustic enhancement of sleep slow oscillations and concomitant memory improvement in older adults. *Frontiers in Human Neuroscience*, 11:109, 2017.

[6] Ruddy K. et al. Fattinger S., de Beukelaar T. Deep sleep maintains learning efficiency of the human brain. *Nature Commun*, 8, 2017.

[7] Giovanni Santostasi, Roneil Malkani, Brady Riedner, Michele Bellesi, Giulio Tononi, Ken A. Paller, and Phyllis C. Zee. Phase-locked loop for precisely timed acoustic stimulation during sleep. *Journal of Neuroscience Methods*, 259:101–14, 2015.

[8] Melanie L. Ferster, Guidon Gaggioni, Giovanni Santostasi, Eus J. W. Van Someren, Pierre J. Arnal, Mirjam Münch, Derk-Jan Dijk, Reto Huber, Lisa Marshall, Hong-Viet V. Ngo, et al. Benchmarking real-time algorithms for in-phase

auditory stimulation of low amplitude slow waves with wearable eeg devices during sleep. *Frontiers in Neuroscience*, 16:955228, 2022.

[9] Dominique Petit, Jean François Gagnon, Maria Livia Fantini, Luigi Ferini-Strambi, and Jacques Montplaisir. Sleep and quantitative EEG in neurodegenerative disorders. *Journal of Psychosomatic Research*, 56(5):487–96, 2004.

[10] Ruben Martinez-Cantin. Bayesopt: A bayesian optimization library for nonlinear optimization, experimental design and bandits. *Journal of Machine Learning Research*, 15(115):3915–3919, 2014.

[11] Eduardo Pérez González. Robot grasping with bayesian optimization informed with foundational models. Master's thesis, Universidad de Zaragoza, 2025. `https://zaguan.unizar.es/record/152263/files/TAZ-TFM-2025-047.pdf`.

[12] Kevin Swersky, Jasper Snoek, and Ryan P Adams. Multi-task bayesian optimization. In C.J. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 26. Curran Associates, Inc., 2013.

[13] Kirthevasan Kandasamy, Gautam Dasarathy, Jeff Schneider, and Barnabás Póczos. Multi-fidelity Bayesian optimisation with continuous approximations. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 1799–1808. PMLR, 06–11 Aug 2017.