Parameter Optimization using high-dimensional Bayesian Optimization

Bachelor Thesis

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

A. K. David Yenicelik

Advisor: J. Kirschner
M. Mutný

Department of Computer Science
Swiss Federal Institute of Technology in Zurich

This dissertation is submitted for the degree of
Bachelor of Science, Computer Science

August 2018

0.1 Project Proposal for Bachelor Thesis

0.1.1 Motivation

Tuning hyperparameters is considered a computationally intensive and tedious task, be it for neural networks, or complex physical instruments such as free electron lasers. Users for such applications could benefit from a 'one-click-search' feature, which would find optimal parameters in as few function evaluations as possible. This project aims to find such an algorithm which is both efficient and holds certain convergence guarantees. We focus our efforts on Bayesian Optimization (BO) and revise techniques for high-dimensional BO.

0.1.2 Background

In Bayesian Optimization, we want to use a Gaussian Process to find an optimal parameter setting $\mathbf{x}^*$ that maximizes a given utility function f. We assume the response surface to be Lipschitz-continuous.

Assume we have observations $\mathscr{Y} = \{y^{(1)}, \ldots, y^{(N)}\}$, each evaluated at a point $\mathscr{X} = \{\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(N)}\}$. The relationship between the observations y and individual parameter settings $\mathbf{x}$ is $y = f(\mathbf{x}) + \varepsilon$ where $\varepsilon \sim \mathscr{N}\left(0, \sigma_n^2\right)$. Any quantity to be predicted has a subscript-star (e.g. y_* is the function evaluation we want to predict).

In it's simplest form, a Gaussian Process is described by the following equation:

$$\begin{pmatrix} y \\ y_* \end{pmatrix} \sim N\left(\mu, \begin{pmatrix} K & K_*^T \\ K_* & K_{**} \end{pmatrix}\right), \tag{1}$$

Where μ is a mean function, $K = \text{kernel}(\mathbf{X}, \mathbf{X})$, $K_* = \text{kernel}(\mathbf{x}_*, \mathbf{X})$ and $K_{**} = \text{kernel}(\mathbf{x}_*, \mathbf{x}_*)$. We predict any new point y_*, (given all previously sampled points y) by estimating the probability $p(y_*|y) \sim N(K_* K^{-1} y, K_{**} - K_* K^{-1} K_*')$

This, in turn, can be used to build an acquisition function. This acquisition function describes where to best sample points next. Some popular acquisition functions include GP-UCB, Most probable improvement (MPI) and Expected Improvement (EI). The choice of the acquisition function has great influence on the performance of the optimization procedure.

We will talk about the problems and possible solutions for the task at hand in the next section.

0.1.3 Scope of the Project

Bayesian optimization suffers from the curse of dimensionality. The goal of this project is to arrive at a solution that resolves the curse of dimensionality for the specific task with regards to Bayesian optimization. This project includes, but is not limited to the following methods.

1. [27] Assume $f(x) \approx g(\mathbf{W}^T x)$ where $\mathbf{W} \in \mathbb{R}^{D \times d}$ and $D >> d$. We assume that $\mathbf{W}$ is orthogonal.

   This algorithm does not require gradient-information (thus, easier to implement, and robust to noise). The standard-deviation, kernel parameters and $\mathbf{W}$ can be found iteratively. First we fix $\mathbf{W}$, and optimize over the standard-deviation, kernel parameters. Then we fix the standard-deviation, kernel parameters. and optimize over $\mathbf{W}$. We repeat this procedure until the change of the log-likelihood between iterations is below some ε_l.

2. [25] Assume $f(x) = f^{(1)}(x^{(1)}) + f^{(2)}(x^{(2)}) + \ldots + f^{(M)}(x^{(M)})$ where $x^{(i)} \in \mathscr{X}^{(i)} \subseteq \mathscr{X}$, i.e. each function component $f^{(i)}$ takes some lower-dimensional subspace as the input. The lower-dimensional subspaces may overlap. The mean and covariance of $f(x)$ is then the sum of the individual component's means and covariances.

   An additive decomposition (as described above) can be represented by a dependency graph. The dependency graph is built by joining variables i and j with an edge whenever they appear together within some set $x(k)$.

   The goal is to maximize an acquisition function $\phi_t(x) = \sum_{i=1}^{M} \phi_t^{(i)}(x^{(i)})$. This maximization is achieved by maximizing the probability of Markov Random Fields within the graph. A junction tree is created from the graph, which is then used to find the global maximum of the acquisition function.

   The dependencies between the variable-subsets are represented through a graph, which can be learned through Gibbs sampling. This, in turn, is used to create a kernel for the GP.

3. [**?**] A function $f : \mathbf{R}^D \to \mathbf{R}$ is said to have effective dimensionality d_e (where $d_e < D$), if there exists a linear subspace $\mathscr{T}$ of dimension d_e such that for all $x_\top \in \mathscr{T} \subset \mathbf{R}^D$ and $x_\perp \in \mathscr{T}_\perp \subset \mathbf{R}^D$, we have $f(x) = f(x_\top + x_\perp) = f(x_\top)$. $\mathscr{T}^\perp$ is the orthogonal complement of $\mathscr{T}$.

   Assume $f : \mathbf{R}^D \to \mathbf{R}$ has effective dimensionality d_e. Given a random matrix $\mathbf{A} \in \mathbf{R}^{D \times d}$ (where $d \geq d_e$) with independent entries sampled fom $\mathscr{N}(0,1)$. For any $x \in \mathbf{R}^D$, there

exists a $y \in \mathbf{R}^d$ such that $f(x) = f(\mathbf{A}y)$. We now only need to optimize over all possible $y \in \mathbf{R}^d$, instead of all possible $x \in \mathbf{R}^D$.

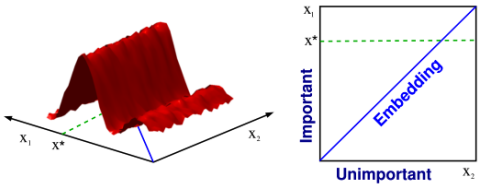


Fig. 1 This function in D=2 dimesions only has d=1 effective dimension. Hence, the 1-dimensional embedding includes the 2-dimensional function's optimizer. It is more efficient to search for the optimum along the 1-dimensional random embedding than in the original 2-dimensional space

If, for some reason, finding an active subspace or an effective lower dimension is not possible, we are open to adapt the procedure of optimization.

Abstract

In this thesis, I explore the possibilities of conducting bayesian optimization techniques in high dimensional domains. Although high dimensional domains can be defined to be between hundreds and thousands of dimensions, we will primarily focus on problem settings that occur between two and 20 dimensions. As such, we focus on solutions to practical problems, such as tuning the parameters for an electron accelerator, or for even simpler tasks that can be run and optimizated just in time with a common laptop at hand.

I follow a systematic methodology, where we identify problems currently occurring with Bayesian Optimization methods. For this, I take on the following steps:

I first provide a theoretic background to the mathematical foundations of Gaussian Processes at hand. I present the derivation of the surrogate functions as a foundation to better understand what parts of the process can be optimized. I then shortly discuss the different acquisition functions that are most often used in practice.

I continue by exploring different techniques that are currently considered as state-of-the-art. Most of these techniques concentrate on doing one of three things. The most common approach is to do a linear dimensionality reduction. The second most common approach is to identify variables that rely on each other, and create different Gaussian Processes for each group of variables that depend on each other.

I identify the shortcomings of the current methods and present quantitative ways on how we can measure improvements for the goal at hand. Together with my supervisors, we present a novel algorithm called "BORING" to do Bayesian Optimization at hand. This algorithm combines the advantages of linear projection methods, but also allows to take into consideration some small perturbations in the target function. Taking into account smaller perturbations is something that linear reduction models have not taken into account so far. BORING proves to have all the advantages that other state-of-the-art linear dimensionality reduction algorithms have. However, BORING improves this state by allowing small pertur-

bations to be taken into account when creating the surrogate function. This feature ultimately allows BORING to be more accurate when modelling functions that we want to optimize over.

Finally, we evaluate BORING and compare it to the competitive state-of-the-art in Bayesian Optimization. We do a quantitative analysis by applying the Gaussian Process algorithms on different, well defined functions and measuring the regret during optimization that it achieves. We do a qualitative analysis by visualizing the predicted surrogate function, and comparing this to the real function. In addition to that, we do a short experiment on whether the subspace identification method can be used to do feature selection, by choosing the projection in such a way, that the most important features will receive the highest matrix weights.

Our main contribution is BORING, an algorithm which is competitive with other state-of-the-art methods in Bayesian Optimization. The main features of BORING are 1.) the possibility to identify the subspace (unlike most other optimization algorithms), and 2.) to provide a much lower penalty to identify the subspace, as optimization is still the main goal.

Table of contents

List of figures

List of tables

Chapter 1

Background

1.1 Bayesian Optimization in high dimensions

Many technical problems can be boiled down to some flavour of black box optimization. Such problems include neural architecture search [15], hyper-parameter search for neural networks, parameter optimization for electron accelerators, or drone parameter optimization using safety constraints [2].

Bayesian optimization methods are a class of sequential black box optimization methods. A surrogate function surface is learned using a Gaussian prior, and a Gaussian likelihood function. Combining the prior and the likelihood results in the Gaussian posterior, which can then be used as a surface over which optimization can take place, with the help of a chosen acquisition function.

Bayesian optimization is a method that has increasingly gained attention in the last decades, as it requires relatively few points to find an appropriate response surface for the function over which we want to optimize over. It is a sequential model based optimization function, which means that we choose the best point x_i^* given all previous points $x_{i-1}^*, x_{i-2}^*, \ldots, x_0^*$. Given certain acquisition functions, it offers a good mixture between exploration and exploitation from an empirical standpoint [1].

However, as machine learning algorithms, and other problems become more complex, bayesian optimization needs to cope with the increasing number of dimensions that define the search space of possible configurations. Because BO methods lose effectiveness in higher dimensions due to the curse of dimensionality, this work explores Bayesian optimization methods that improve the optimization performance in higher dimensions. Finally, we

propose a novel method that improves on certain characteristics of the current state-of-the-art.

1.2 Gaussian Processes

Bayesian Optimization aims at using a Gaussian Process as an intermediate representation to find an optimal parameter setting $\mathbf{x}^*$ that maximizes a given utility function f. We assume the response surface to be Lipschitz-continuous.

Assume we have observations $\mathscr{Y} = \{y^{(1)}, \ldots, y^{(N)}\}$, each evaluated at a point $\mathscr{X} = \{\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(N)}\}$. The relationship between the observations y and individual parameter settings $\mathbf{x}$ is $y = f(\mathbf{x}) + \varepsilon$ where $\varepsilon \sim \mathscr{N}\left(0, \sigma_n^2\right)$. Any quantity to be predicted has a subscript-star (e.g. y_* is the function evaluation we want to predict).

In it's simplest form, a Gaussian Process is described by the following equation:

$$\begin{pmatrix} y \\ y_* \end{pmatrix} \sim N\left(\mu, \begin{pmatrix} K & K_*^T \\ K_* & K_{**} \end{pmatrix}\right), \tag{1.1}$$

Where μ is a mean function, $K = \text{kernel}(\mathbf{X}, \mathbf{X})$, $K_* = \text{kernel}(\mathbf{x}_*, \mathbf{X})$ and $K_{**} = \text{kernel}(\mathbf{x}_*, \mathbf{x}_*)$. Any new point y_* is predicted given all previously sampled points y by estimating the probability $p(y_*|y) \sim N(K_* K^{-1} y, K_{**} - K_* K^{-1} K_*')$

These results can then be used by an acquisition function, that described where to best sample the points next. Some popular acquisition functions include GP-UCB, Most probable improvement (MPI) and Expected Improvement (EI). The choice of the acquisition function has great influence on the performance of the optimization procedure.

In the following, I provide a short derivation of the core formulae used for Gaussian Processes.

1.2.1 Derivation of the Gaussian Process Formula

The **prior** for the Gaussian Process is the following (assuming the commonly chosen 0-mean-prior).

$$u \sim GP(0, k(x, x')) \tag{1.2}$$

u is a random variable following a Gaussian Process distribution and its probability distribution is given by a normal distribution:

$$p(u) = N(0,K) \tag{1.3}$$

Additional data $X = \{(x_1,y_1),\ldots,(x_n,y_n)\}$ can be observed. The Gaussian Process incorporates an error term that takes into consideration possible noise in the measurement of the experiment. Thus, y has some noise term ε such that $y = u(x) + \varepsilon$. By the central limit theorem, commonly the assumption is taken that ε is normally distributed around 0 with σ_s standard deviation. Given the sampled datapoints, and the inherent noise that these datapoints have, the **likelihood** of the Gaussian Process can be represented as follows:

$$p(y|x,u) = N(u,\sigma_s^2 I) \tag{1.4}$$

Given the prior and likelihood of the Gaussian Process, the **posterior** of the Gaussian Process can be derived by simple application of Bayes rule.

$$p(u|x,y) = \frac{p(y|x,u)p(u)}{p(y|x)} \qquad = N(K(K+\sigma^2 I)^{-1}y, \sigma^2(K+\sigma^2 I)^{-1}K) \tag{1.5}$$

From the above posterior, we now want to predict for an arbitrary x_* the function value y_*. Predicting y_* for every possible x_* in the domain results in the surrogate response surface that the GP models.

We assume that the value y_* we want to predict is also distributed as a Gaussian probability distribution. Because the y_* that we want to predict relies on all the values collected in the past (which are again normally distributed), the probability distribution can be modelled as jointly Gaussian:

$$\begin{pmatrix} y \\ y_* \end{pmatrix} \sim N\left(\mu, \begin{pmatrix} K & K_*^T \\ K_* & K_{**} \end{pmatrix}\right), \tag{1.6}$$

To compute this equation, we use the results from Murphy's textbook [21] pages 110 to 111 to do inference in a joint Gaussian model.

1.3 Acquisition Functions

Given the above formula for the posterior mean μ and the poster variance σ^2, Bayesian Optimization makes use of an acquisition function. The following is a summary of the most popular acquisition functions in recent literature. A good summary is given by [31].

1.3.1 Upper Confident Bound (UCB)

[26] shows a proof, that for a certain tuning of the parameter β, the acquisition function has asymptotic regret bounds.

The upper confidence bound allows the user to control exploitation and exploration through a parameter $\beta > 0$, which can be chosen as specified in [26] to offer regret bounds. In addition to that, GP-UCB shows state-of-the-art empirical performance in numerous use-cases[6].

$$UCB(x) = \mu(x) + \sqrt{\beta}\sigma(x) \tag{1.7}$$

Here, the functions μ and σ are the predicted mean and variance of the Gaussian Process Posterior.

1.3.2 Probability of Improvement (PI)

The (maximum) probability of improvement [4] always selects the points where the mean plus uncertainty is above the maximum explored function threshold. The downside to this policy is that this leads to heavy exploitation. However, the intensity of exploitation can be controlled by a parameter $\xi > 0$.

$$PI(x) = P(f(x) \geq f(x^+) + \xi) \tag{1.8}$$

$$= \Phi(\frac{\mu(x) - f(x^+) - \xi}{\sigma(x)}) \tag{1.9}$$

1.3.3 Expected Improvement (EI)

As an improvement to the maximum probability of improvement, the expected improvement takes into consideration not only the probability that a point can improve the maximum found so far. But that the EI also takes into account the magnitude by which it can improve the maximum function value [4]. As in MPI, one can control the rate of exploitation by setting the parameter $\xi > 0$, which was introduced by [19].

$$EI(x) = \begin{cases} (\mu(x) - f(x^+) - \xi)\Phi(Z) + \sigma(x)\phi(Z) & \text{if } \sigma(x) > 0 \\ 0 & \text{if } \sigma(x) = 0 \end{cases} \tag{1.10}$$

$$\tag{1.11}$$

where

$$Z = \frac{\mu(x) - f(x^+) - \xi}{\sigma(x)} \tag{1.12}$$

and where ϕ denotes the PDF, and Φ denotes the CDF of the standard normal distribution respectively.

Given one of the above acquisition functions, one can then use an optimizer such as *LBFGS* or monte carlo sampling methods, to find an approximate global maximum of the respective function. The combination of Gaussian Processes and Acquisition function together result in a Bayesian Optimization algorithm, which has a prior assumption about the function to be learned, and uses data-samples to create a likelihood to further refine the posterior of the initial function assumption.

1.4 Resources

For the experiments and code basis, most of our functions rely on the Sheffield Machine Learning - GPy library [10]. In addition to that, I use the febo framework developed by Johannes Kirschner from the Learning and Adaptive Systems group at ETH Zurich.

Chapter 2

Related Work

This section covers current methods solving Bayesian Optimization in high dimensions that are considered state-of-the-art. This is not an exhaustive review. For each algorithm, I discuss the effectiveness with regards to the dataset or function that the respective paper does the evaluation on.

Unless specified differently, I use the following terminology during my discussion.

1. f is the real function to be optimized. Often I will use the term approximate, as many algorithms rely on a surrogate approximation of the function f such that the global optimum can be found.

2. Assuming the function f is of the form $f(x) = y$ with $x \in \mathbf{R}^D$, where D denotes the dimensionality of x, then we define the optimized value of f to be $x^* = \arg\max_x f(x)$.

3. g and any subscripted or superscripted derivative of g is a component that one uses to approximate f.

4. Anything that has a "hat" on (caret symbol on f is $\hat{f}$), refers to an empirical estimate. $\hat{f}$ would be an empirical estimate given datapoints D of f.

I will focus on three categories of Bayesian Optimization algorithms: Algorithms that make use of a projection matrix, algorithms that, algorithms that exploit additive substructures and "additional approaches" that are uncategorised.

2.1 Projection matrix based algorithms

In my work, I focus on algorithms that optimize the black box function f by using a lower-dimensional projection. I proceed with discussing some algorithms that I have implemented

for my experiments in the next subsection (all except the algorithm "Active learning of linear subspaces"). I will briefly describe additional interesting algorithms that raise interesting ideas for possible future work afterwards. For this family of algorithms, the approximation is a function of $f(x) \sim g(x;A)$, where the properties of A are algorithm-specific. The following descriptions aims at giving a brief overview at the methods at hand. I refer the curious reader to the individual paper for a more detailed and formal description of the respective topic.

2.1.1 Active learning of linear subspaces

Algorithm 1 Simultaneous active learning of functions and their linear embeddings (pseudocode) :: Active learning of linear subspace [9]

Require: d, D; kernel κ, mean function μ; prior $p(R)$
  $X \leftarrow \emptyset$
  $Y \leftarrow \emptyset$
  while budget not depleted **do**
    $q(R) \leftarrow \text{LAPLACEAPPROX}(p(R|X,Y,\kappa,\mu))$
    $q(f) \leftarrow APPROXMARGINAL(p(f|R),q(R))$
    $x_* \leftarrow OPTIMIZEUTILITY(q(f),q(R))$
    $y \leftarrow OBSERVE(f(x_*))$
    $X \leftarrow [X;x_*]$
    $Y \leftarrow [Y;y_*]$
  end while
  return $q(R), q(f)$

[9] The assumption of this algorithm is that f depends only on $x := uR^T$ with $R \in \mathbf{R}^{d \times D}$, $u \in \mathbf{R}^d$ and where $d << D$. The algorithm learns a projection matrix R and the surrogate function$g(u)$, with $f(x) \sim g(u)$.

The **Laplace approximation** for R is using the mode of the probability distribution $\log P(R|D)$ as a mean, and the covariance is taken as the inverse Hessian of the negative logarithm of the posterior evaluated at the mean of the distribution. Together, this describes the probability distribution $p(R|X,Y,\kappa,\mu)$, where μ is the mean function, and κ is the covariance function.

The **approximate marginal** subroutine is a novel method proposed in the paper that integrates over the different parameters in the paper. This marginal approximation does a local expansion of $q(x_*|\theta)$ to $p(x_*|D,\theta)$.

The **sequential optimization of utility** (choice of next best point) is done using Bayesian Active Learning by disagreement, where the utility function is the expected reduction in the mutual information, as opposed to uncertainty sampling reducing entropy, which is not well defined for all values.

The metrics used in this paper are negative log-likelihoods for the test points, and the mean symmetric kullback leiber divergence between approximate and true posteriors. The proposed method always outperforms the naive MAP method for the presented functions close to a factor of 2 for both loss functions. Tests are conducted on a real, and synthetic dataset with up to $D = 318$ and selecting $N = 100$ observations.

2.1.2 Random embeddings (REMBO)

REMBO is an algorithm that allows the optimizer to search in a smaller search space. The result of the optimized value x^* is then projected to the higher dimensional space (including projections), to retrieve the actual optimized argmax value of the function f. More specifically, REMBO proposes the following model for Bayesian Optimization:

[30] Let $x \in \mathbb{R}^D$ and $y \in \mathbb{R}^d$. Assume, that $f(x) = g(Ay)$. We can generate $A \in \mathbb{R}^{D \times d}$ by randomly generating this matrix. A must have the property that $A^T \times A = I$, where $I \in \mathbf{R}^{d \times d}$ denotes the identity matrix with d diagonal elements. The space over which the user searches, as such, is d-dimensional. This implies that REMBO is more efficient in sampling datapoints, and learning the structure of a neighborhood in the high-dimensional space (by using the neighborhood of the low-dimensional space).

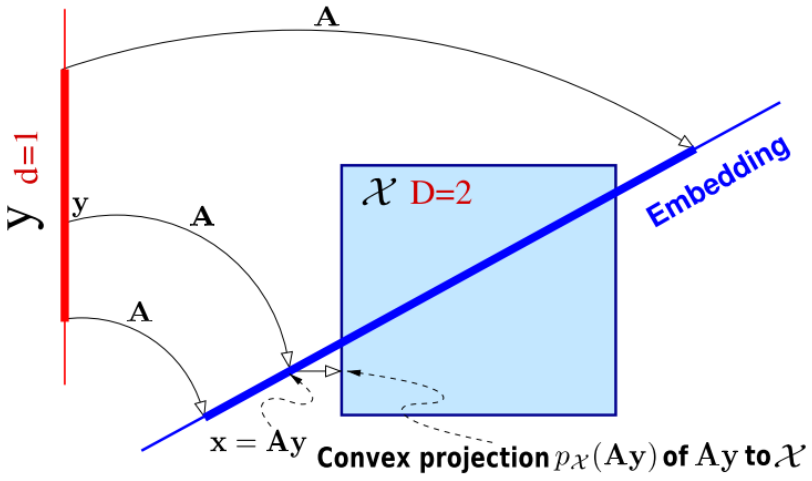


Fig. 2.1 Parabola Original

Fig. 2.2 Source [30]: Embedding from $d = 1$ into $D = 2$. The box illustrates the 2D constrained space $\mathbf{X}$, while the thicker red line illustrates the 1D constrained space $\mathbf{Y}$. Note that if $A \times y$ is outside of $\mathbf{X}$, it is projected onto $\mathbf{X}$ using a convex projection. The set $\mathbf{Y}$ must be chosen large enough so that the projection of its image, $A \times y$ with $y \in \mathbf{Y}$, onto the effective subspace (vertical axis in this diagram) covers the vertical side of the box.

The elegance of REMBO lies in the fact that the matrix A can be chosen as a random orthogonal matrix. The authors argue with an empirical proof, that if the optimization domain's parameters are well-chosen, that the chance of getting a bad projection matrix has a certain threshold.

I now proceed with a more formal treatment of the intuitive concept explained above:

[**?**] A function $f : \mathbf{R}^D \to \mathbf{R}$ is said to have effective dimensionality d_e (where $d_e < D$), if there exists a linear subspace $\mathscr{T}$ of dimension d_e such that for all $x_\top \in \mathscr{T} \subset \mathbf{R}^D$ and $x_\perp \in \mathscr{T}_\perp \subset \mathbf{R}^D$, we have $f(x) = f(x_\top + x_\perp) = f(x_\top)$. $\mathscr{T}^\perp$ is the orthogonal complement of $\mathscr{T}$.

Assume $f : \mathbf{R}^D \to \mathbf{R}$ has effective dimensionality d_e. Given a random matrix $\mathbf{A} \in \mathbf{R}^{D \times d}$ (where $d \geq d_e$) with independent entries sampled from $\mathscr{N}(0,1)$. For any $x \in \mathbf{R}^D$, there exists a $y \in \mathbf{R}^d$ such that $f(x) = f(\mathbf{A}y)$. The user now only need to optimize over all possible $y \in \mathbf{R}^d$, instead of all possible $x \in \mathbf{R}^D$.

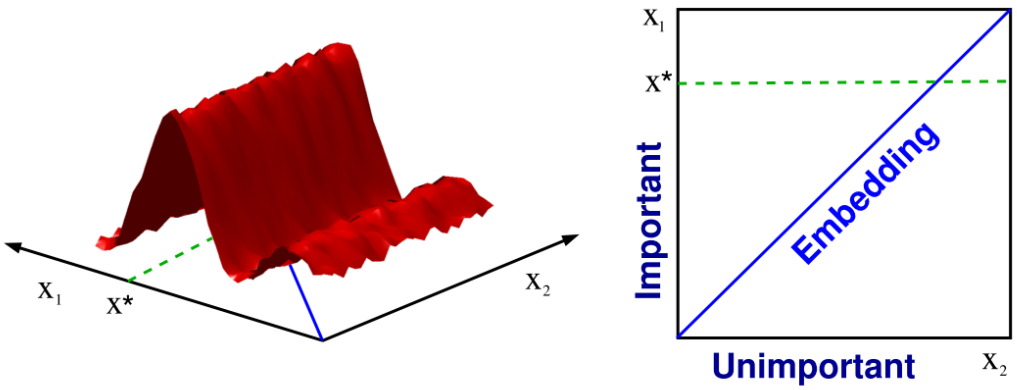


Fig. 2.3 Parabola Original

Fig. 2.4 Source [30]: This function in D=2 dimesions only has d=1 effective dimension. Hence, the 1-dimensional embedding includes the 2-dimensional function's optimized value x^*. It is more efficient to search for the optimum along the 1-dimensional random embedding than in the original 2-dimensional space.

REMBO has a fairly high probability of failing (more than 20% in the experiments conducted in the paper) by choosing an A that lies orthogonal the the directions of highest change in the function f. The authors propose interleaved runs, where for each k'th point selection, a differently sampled orthogonal random matrix is chosen. The probability of generating a bad embedding lower with higher k. However, each k has a set of observations X which is not shared across the different interleaved runs.
Extensions to REMBO include [3].

2.1.3 Applications to high-dimensional uncertainty propagation

I put most emphasis on this algorithm during my experiments. As such, I will be more detailed with the description of this algorithm.

[27] This algorithm assumes that $f(x) \sim g(\mathbf{W}^T y)$ where $\mathbf{W} \in \mathbb{R}^{D \times d}$ and $D >> d$. $\mathbf{W}$ again has the property that $\mathbf{W^T} \times \mathbf{W} = I$ where $I \in \mathbf{R}^{d \times d}$ denotes the identity matrix in d dimensions. This algorithm does not require gradient-information. This makes it easier to implement, and more robust to noise according to the authors of this paper.

I refer to the set of kernel parameters and the GP noise variance as GP-hyperparameters. I refer to the projection matrix $\mathbf{W}$ as the projection matrix.

The GP noise variance, kernel parameters and $\mathbf{W}$ can be found iteratively. The main idea of the algorithm is to first fix both the kernel parameters and the GP noise variance, and identify $\mathbf{W}$. Then, we fix $\mathbf{W}$ and train over the GP-hyperparameters. This procedure is repeated until the change of the log-likelihood between iterations is below some ε_l, or if the maximum number of steps of optimization is reached. We repeat this procedure many times (as dictated by the number of restarts). Empirially, the probability of finding a good projection increases with the number of restarts, and the maximum number of iterations the algorithm is allowed to take during the optimization of the joint parameter set $\mathbf{W}$ and the hyperparameters (as given in the next paragraphs).

I now proceed with a more formal description of the algorithm. The quantities of interest are

$$\mu_f = \int f(x)p(x)dx \tag{2.1}$$

$$\sigma_f^2 = \int (f(x) - \mu_f)^2 p(x)dx \tag{2.2}$$

$$f \sim p(f) = \int \delta(f - f(x))p(x)dx \tag{2.3}$$

The authors assume that w.l.o.g., the search space of the projection matrices can be restricted to matrices on the siefel manifold. The argument for this is that the projected subspace does not change, merely the representation of the subspace. Here, the family of orthogonal matrices of dimension $d \times D$ is denoted by $\mathbf{W} \in V_d(\mathbb{R}^D)$. This quantity is also known as the **Stiefel manifold** [23] [5] [17], where d is the found effective dimension of the function, and D is the real input dimension to the function.

The Matern32 Kernel

The optimization processes in the paper use the Matern-32 kernel function. This function has two input vectors a and b.

$$K(a,b,\theta) = s^2 \left(1 + \sqrt{3}\sum_{i=1}^{l} \frac{(a_i - b_i)^2}{l_i}\right) exp\left(-\sqrt{3}\sum_{i=1}^{l} \frac{(a_i - b_i)^2}{l_i}\right) \tag{2.4}$$

Because I want to avoid numerical testing and implementation, I use the derivative as provided in the GPy library. The $s, l_1, \ldots, l_l$ are hyper-parameters of the kernel, referred to as the kernel-variance, and the kernel-lengthscales. The concatenated vector of all these kernel hyperparameters is denoted by ϕ.

The only modification made to this kernel is the additional parameter W:

$$k_{AS} : \mathbb{R}^D \times \mathbb{R}^D \times V_d(\mathbb{R}^D) \times \phi -> \mathbb{R} \tag{2.5}$$

where the kernel has the form

$$k_{AS}(x, x'; W, \phi) = k_d(W^T x, W^T x'; \phi) \tag{2.6}$$

2.1.4 Overview of the algorithm

In the following, I will explain the individual steps of the algorithm. The shown two steps are repeated until the change of the log-likelihood reaches a certain threshold. This algorithm is repeated a number of restarts, as the initial sample of W plays a big role in finding a good converged embedding.

Step 1.: Determine the active projection matrix W

In this step, the algorithm optimizes $W \in V_d(\mathbb{R}^D)$ while keeping the kernel hyperparameters ϕ and the GP noise variance s_n fixed.

To simplify calculations later, we define the function, where all parameters but W are fixed as F. The other parameters are determined from previous runs, or are freshly sampled:

$$F(W) := \mathscr{L}(W, \phi, s_n; X, y) \tag{2.7}$$

$$= \log p(y|X, W, \phi, s_n) \tag{2.8}$$

$$= -\frac{1}{2}(y-m)^T (K + s_n^2 I_N)^{-1} (y-m) - \frac{1}{2}\log|K + s_n^2 I_N| - \frac{N}{2}\log 2\pi \tag{2.9}$$

$$\tag{2.10}$$

where $\phi, s_n; X, y$ are fixed and m is the prior mean function, which is 0 in our specific case.

To optimize over the loss function, the algorithm defines the derivative of F with regards to each individual element of the weights-matrix:

$$\nabla_{w_{i,j}} F(W) := \nabla_{w_{i,j}} \mathscr{L}(W, s_n; X, y) \tag{2.11}$$

$$= \frac{1}{2}\mathrm{tr}\left[\{(K+s_n^2 I_N)^{-1}(y-m)\left((K+s_n^2 I_N)^{-1}(y-m)\right)^T - (K+s_n^2 I_N)^{-1}\}\nabla_{w_{i,j}}(K+s_n^2 I_N)\right] \tag{2.12}$$

both these functions depend on the kernel K, and it's derivative $\nabla_{w_{i,j}} K$.

To optimize over F, a more sophisticated algorithm is used, that resembles iterative hill-climbing algorithms. First, the paper defines the function whose output is a matrix in the Stiefel manifold

$$\gamma(\tau; W) = (I_D - \frac{\tau}{2}A(W))^{-1}(I_D + \frac{\tau}{2}A(W))W \tag{2.13}$$

where W is a fix parameter, and τ is the variable which modifies the direction that we move on in the Stiefel manifold and with

$$A(W) = \nabla_W F(W)W - W(\nabla_W F(W))^T \tag{2.14}$$

One iteratively chooses fixed W, and does a grid-search over τ such that at each step, the log-likelihood $\mathscr{L}$ is increased.

Step 2.: Optimizing over GP noise variance and the kernel hyperparameters

We determine the hyperparameters by optimizing over the following loss function, where X are the input values, Y are the corresponding output samples. ϕ is the vector of the kernel hyperparameters and s_n, the GP noise variance.

One keeps the W fixed (either by taking W from the last iteration, or freshly sampling it), and then defines the loss function

$$L(\phi) = \mathscr{L}(W, \phi, s_n; X, y) \tag{2.15}$$

To optimize this loss function, a simple optimization algorithm such as $L-BFGS$ is used to individually maximize each element of the hyperparameter vector with regards to the log-likelihood. This is done for a maximum number of steps, or until the change of improvement becomes marginal.

Additional details

Because initialization is a major factor in this algorithm, these steps are iteratively applied for many hundred steps. There are also many tens or hundreds of restarts to ensure that the search on the Stiefel manifold results in a good local optimum, and does not get stuck on a flat region with no improvement. This algorithm is very sensitive on the initially sampled parameter W.

Identification of active subspace dimension

One cannot know the real active dimension of a problem that one does not know the solution to. As such, the proposed to apply the above algorithms iteratively by increasing the selected active dimension d. The moment where the relative change between the best found matrix between two iterations is below a relative threshold ε_s, the previous active dimension is chosen as the real active dimension. The algorithm identifies the loss for each possible active dimension. It then chooses a dimension, where the relative difference to the previous loss (of the one-lower dimension) is below a certain threshold.

2.2 Algorithms that exploit additive substructures

Functions with additive substructures can be decomposed into a summation over subfunctions, such that $f(x) \sim g_0(x) + g_1(x) + \dots g_2(x)$ where each g_i may operate only on a subset of dimensions of x.

2.2.1 Independent additive structures within the target function

[8] Assume that $f(x) = \sum_{i=1}^{|P|} f_i(x[P_i])$, i.e. f is fully additive, and can be represented as a sum of smaller-dimensional functions f_i, each of which accepts a subset of the input-variables. The kernel also results in an additive structure: $f(x) = \sum_{i=1}^{|P|} k_i(x[P_i], x[P_i])$. The posterior is calculated using the Metropolis Hastings algorithm. The two actions for the sampling algorithm are 'Merge two subsets', and 'Split one set into two subsets'. k models are sampled,

and we respectively approximate $p(f_*|D,x^*) = \frac{1}{k}\sum_{j=1}^{k} p(f(x^*|D,x,M_j))$, where M_j denotes the partition amongst all input-variables of the original function f.

2.3 Additional approaches

2.3.1 Elastic Gaussian Processes

[22] Use a process where the space is iteratively explored. The key insight here is that with low length-scales, the acquisition function is extremely flat, but with higher length-scales, the acquisition function starts to have significant gradients. The two key-steps is to 1.) additively increase the length-scale for the gaussian process if the length-scale is not maximal and if $||x_{init} - x^*|| = 0$. And 2.) exponentially decrease the length-scale for the gaussian process if the length-scale is below the optimum length-scale and if $||x_{init} - x^*|| = 0$.

2.3.2 High dimensional Gaussian bandits

[6] This model assumes that there exists a function $g : \mathbf{R}^k \implies [0,1]$ and a matrix $A \in \mathbf{R}^{d\times D}$ with orthogonal rows, such that $f(x) \sim g(Ax)$. Assume $g \in \mathscr{C}^2$.

The algorithm identifies A by the

Algorithm 2 The SI-BO algorithm [6]

Require: $m_X, m_\Phi, \lambda, \varepsilon, k$, oracle for the function f, kernel κ
  $C \leftarrow m_X$ samples uniformly from $\mathbb{S}^{d-1}$
  for $i \leftarrow 1$ to m_X **do**
    $\Phi_i \leftarrow m_\Phi$ samples uniformly from $\{-\frac{1}{\sqrt{m}}, \frac{1}{\sqrt{m}}\}^k$
  end for
  $y \leftarrow$ (compute using Equation 1 – INSERT Eq 1 here, or create a summary of all important equations here)
  select z_i according to a UCB acquisition function, evaluate f on it, and add it to the datasamples found so far

The SI-BO algorithm consists of a subspace identification step, and an optimization step using GP-UCB.

The **subspace identification** is treated as a low-rank matrix recovery problem as presentedin [13].

2.3.3 Bayesian Optimization using Dropout

[16] propose that the assumption of an active subspace is restrictive and often not fulfilled in real-world applications. They propose three algorithms, to iteratively optimize amongst certain dimensions that are not within the d 'most influential' dimensions: 1.) Dropout Random, which picks dimensions to be optimized at random, 2.) Dropout copy, which continuous optimizing the function values from the found local optimum configuration, and 3.) which does method 1. with probability p, and else method 2. The d 'most influential' dimensions are picked at random at each iteration.

Additional works I explored as background research or additional methods include [11], [20], [18], [29], [24].

Chapter 3

Fields of Improvement

3.1 Shortcomings of current methods

I will enumerate select models from the section "related work", and will shortly discuss what the shortcomings of these models are:

REMBO is a purely optimizational algorithm, which finds optimizations in lower dimensions

- **Suitable choice of the optimization domain:** REMBO is not purely robust, as there is a considerable chance that no suitable subspace will be found. Empirically, the choice of the optimization domain heavily affects the duration and effectiveness of the optimization. I have found that the proposed optimization domain $\left[-\sqrt{d}, \sqrt{d}\right]^d$ is not well chosen for smaller environments, such as the Camelback function embedded in 5 dimensions. In any case, this is a very sensitive hyperparameter
- **Identification of subspace:** In some settings, including optimization with safety constraints, knowing the subspace that the model projects to is advantageous. REMBO is an implicit optimizer, in that it does not find any subspace, but optimizes through a randomly sampled matrix.
- **Probability of failure:** REMBO has a relatively high probability of failure. The authors propose that restarting REMBO multiple times would allow for a good optimization domain to be found, which leads to interleaved runs.

Active subgradients can be a viable option if we have access to the gradients of the problem, from which we can learn the active subspace projection matrix in the manner by using that gradient matrices.

- **Access to gradients:** For optimization algorithms, the function we want to optimize over is usually a black-box function. Practically, many black-box functions don't offer access to gradient information. To approximate the gradients using sampled points, this would require a high number of datapoints per dimension. In addition to that, these points would have to be evenly distributed, such that the gradients can be effectively estimated for more than one region.
- **Robustness to noise:** According to [27], methods that approximate gradients and use this gradient information to create subspace projectsion are very sensitive to noise. Depending on the application, this can make the algorithm ineffective as it is not robust to small variations in the response surface.

Given the nature of real-world data, approximating the active subspace using the gradients of the data-samples is thus not a robust, and viable option.

Tripathy argues that its method is more robust to real-world noise. It also does not rely on gradient information of the response surface. Tripathy's method allows for a noise-robust way to identify the active subspace.

- **Duration of optimization:** In practice, Tripathy's method takes a long time, especially if the dimensions, or the number of data-points are high. This is due to the high number of matrix multiplications. Especially for easier problems, it is often desirable if the running time of optimizing for the next point is a few minutes or seconds, rather than hours.
- **Efficiency:** In practice, Tripathy's method relies on a high number of restarts. From our observations, the number of steps to optimize the orthogonal matrix becomes relevant as the number of dimensions grow. Given the nature of accepting any starting point, it does not allow for a very efficient way to search for the best possible projection matrix. A more efficient way to search all possible matrices - by incorporating heuristics for example - would be desirable.
- **Insensitive to small perturbations:** Although Tripathy's model finds an active subspace, it completely neglects other dimensions which could allow for small perturbations to allow for an additional increase the global optimum value. Although we

can control to what extent small perturbations should be part of the active subdimension, one usually wants to choose a significant cutoff dimension, but still incorporate additional small perturbations without sacrificing the effectiveness of the projection.

3.2 Method of measuring improvements

In the following sections, we will discuss and show how we can improve on the shortcomings of the above methods. Because practicality is important in our method, we will use synthetic functions to measure the efficiency of our method.

Some terms that allow us to measure the performance of a Bayesian optimization algorithm or a GP surrogate function include:

- Test if the expectation
$$E[f(Ax) - \hat{f}(\hat{A}x)]$$
decreases / approaches zero (for methods that identify a projection matrix). Often, the root mean square error is a good empirical approximate of this quantity:
$$RMSE = \sqrt{\frac{1}{T}(\sum_{t=1}^{T} f(Ax_t) - \hat{f}(\hat{A}x_t))^2} \tag{3.1}$$
The log likelihood estimate is also an estimate which tests this value for the training data.

- For optimization problems, one is often interested in the quantity of cumulative regret. Regret is defined as the difference between the best found function value so far, minus the function value chosen at this timestep t [**?**].
$$R_T = \frac{1}{T}\sum_{t=1}^{T} \max_x f(x) - f(x_t) \tag{3.2}$$
The cumulative regret sums all the entire episode of the run. This is a measure of how fast an optimizer can learn the optima of a function.

- Check if the test log-likelihood decreases for functions that are provided by a finite number of data-points.

- Check if the angle between the real projection matrix, and the found projection matrix decreases, as given in [12].

$$dist(A,B) = \left\| AA^T - BB^T \right\|_2 \tag{3.3}$$
$$= sin(\phi) \tag{3.4}$$

where $A, B \in \mathbf{R}^{D \times d}$

3.2.1 Synthetic Datasets

5 dimensional function with 2 dimensional linear embedding One can evaluate synthetic functions at any point, and immediately get a regret value. The following synthetic functions cover different use cases.

2D to 1D : A simple Parabola which is embedded in a 2D space. This function is supposed to check that the complexity of the model does not hinder discovering simpler structures - in other words, the model complexity should still allow for finding simpler embeddings and functions.

3D to 2D : The Camelback function which is embedded in a 3D space. This checks how tight the model can actually approximate the 2D subspace, or if a 3D UCB performs better than our models.

5D to 2D : The Camelback function which is embedded in a 5D space. This checks if more complicated models can be found within higher dimensional spaces.

Chapter 4

A New Model

Given the fields of improvements in the above section, we now propose an algorithm which addresses the majority of the issues. I will first present that algorithm, and then point out, as to why each individual concern is addressed.

4.1 The BORING Algorithm

We propose the following algorithm, called BORING. **BORING** stands for **B**ayesian **O**ptimization using **R**andom and **I**de**N**tifyable subspace **G**eneration.

The general idea of boring can be captured in one formula, where f stands for the real function that one wishes to approximate, and any subsequent function annotated by g refers to a component of the right hand side.

$$f(x) \approx g_0(Ax) + \sum_{i \in \mathbb{Z}^+}^{q} g_i(A^{\perp}x)_i) \tag{4.1}$$

Where the following variables have the following meaning

- A is the active subspace projection (an element of the stiefel manifold) learned through our algorithm, using Algorithm 1
- $A^{\perp}$ is an matrix whose subspace is orthonormal to the projection of A. We randomly generate $A^{\perp}$ using Algorithm 2.
- The subscript i in the right additive term denotes that we view each output dimension of $dot(A^{\perp}, X)$ as independent to the other output dimensions.

I will now proceed with a more detailed description.

4.1.1 Algorithm Description

Overview

We explore a novel method which is based on additive GPs and an active subspace projection matrix. We use different kinds of kernels. We want to calculate g_i and A as defined in 4.1, such that the log-likelihood of the data we have accumulated so far is maximized.

The following few steps are applied after a "burn-in" period, in which we use random sampling to acquire new points. We sample the datapoints using UCB, as empirically, this is similar to a random sampling of the datapoints (with the subspace identification in mind).

In simple terms, the algorithm proceeds as follows:

1. Pick the first n samples using UCB sampling. During this period, we use UCB with a naive kernel, which has as many dimensions as the domains dimensions. From the collected points, approximate the active projection matrix A using algorithm 1 from [27].

2. Generate a basis that is orthonormal to every element in A. Concatenating these basis vectors $v_1, \ldots, v_{n-q}$ amongst the column-dimension gives us the passive projection matrix $A^{\perp}$.

3. Maximize the GP for each individual expression of the space within A, and parallel to that also orthogonal to A (as given by $A^{\perp}x$) individually.

This addresses the curse of dimensionality, as we can freely choose $q \geq d_e$ to set the complexity of the second term while the first term still allows for creating proximity amongst different vectors by projecting the vectors onto a smaller subspace. The active subspace captures the direction of strongest direction, whereas the passive subspace projection captures an additional GP that adds to the robustness of the algorithm, should the subspace identification fail. The additive terms that get projected onto the passive subspace also allows to incorporate smaller perturbations in the space orthogonal to A to occur.

Algorithm 3 BORING Alg. 1 - Bayesian Optimization using BORING

$X \leftarrow \emptyset$
$Y \leftarrow \emptyset$
{Burn in rate - don't look for a subspace for the first 50 samples}
$i \leftarrow 0$
while i < 50 **do**
  $i++$
  $x_* \leftarrow \text{argmax}_x$ acquisitionFunction$(dot(Q^{\perp}, x))$ using standard UCB over the domain of X.
  Add x_* to X and $f(x_*)$ to Y.
end while
$A, d, \phi \leftarrow$ Calculate active subspace projection using Algorithm 2 from the paper by Tripathy.
$A^{\perp} \leftarrow$ Generate passive subspace projection using Algorithm 3.
$Q \leftarrow$ colwiseConcat($[A, A^{\perp}]$)
$gp \leftarrow GP(\text{dot}(Q^T, X), Y)$
kernel $\leftarrow$ activeKernel + $\sum_i^q$ passiveKernel$_i$
while we can choose a next point **do**
  $x_* \leftarrow \text{argmax}_x$ UCB$(dot(Q^{\perp}, x))$
  Add x_* to X and $f(x_*)$ to Y.
end while
return $A, A^{\perp}$

Where ϕ are the optimized kernel parameters for the activeKernel. The active projection matrix using the following algorithm, which is identical to the procedure described in [27]. The generation of the matrix $A^{\perp}$ is described next.

Finding a basis for the passive subspace (a subspace orthogonal to the active subspace)

$$A = \begin{bmatrix} \vdots & \vdots & & \vdots \\ a_1 & a_2 & \dots & a_{d_e} \\ \vdots & \vdots & & \vdots \end{bmatrix} \tag{4.2}$$

Given that we choose a maximal lower dimensional embedding (maximising the log-likelihood of the embedding for the given points), some other axes may be disregarded. However, the axes that are disregarded may still carry information that can make search faster or more robust.

To enable a trade-off between time and searchspace, we propose the following mechanism.

Assume an embedding maximizing 4.2 is found. Then the active subspace is characterized by it's column vector $a_1, a_2, ..., a_{d_e}$. We refer to the space spanned by these vectors as the *active subspace*.

However, we also want to address the subspace which is not addressed by the maximal embedding, which we will refer to *passive subspace*. This passive subspace is characterized by a set of vectors, that are pairwise orthogonal to all other column vectors in A, i.e. the vector space orthogonal to the active subspace spanned by the column vectors of A.

As such, we define the span of the active and passive subspace is defined by the matrix:

$$Q = \begin{bmatrix} A & A^{\perp} \end{bmatrix} \tag{4.3}$$

where $A^{\perp}$ describes the matrix that is orthogonal to the columnspace of A. For this, $A^{\perp}$ consists of any set of vectors that are orthogonal to all other vectors in A.

The vectors forming $A^{\perp}$ is generated by taking a random vector, and applying Gram Schmidt. This procedure is repeated for as many orthogonal vectors as we want. The procedure is summarised in Algorithm 3:

Algorithm 4 BORING Alg. 3 - generate orthogonal matrix to A(A, n)

Require: A a matrix to which we want to create $A^{\perp}$ for; n, the number of vectors in $A^{\perp}$.

```
normedA ← normalize each column of A
Q ← emptyMatrix() { The final concatenated Q will be A^⊥. }
for i = 1,...,n do
    i ← 0
    while True do
        i++
        q_i ← random vector with norm 1
        newBasis = apply gram schmidt single vector( [A, Q], q_i )
        if dot(normedA^T, newBasis) ≈ 0 and | newBasis | > 1e − 6 then
            Q ← colwiseConcatenate( (Q, newBasis)
            break
        end if
    end while
end for
return Q
```

Additive UCB acquisition function

Because the function is decomposed into multiple additive components, the computation of the mean and variance needs to be adapted accordingly. Although I don't use this method in my experiments, it is still useful to mention one way to approximate these terms for higher dimensional input. Referring to [25], the following method is used.

$$\mu_{t-1}^{(j)} = k^j(x_*^{(j)}, X^j)\Delta^{-1}y \tag{4.4}$$

$$\left(\sigma_{t-1}^{(j)}\right)^2 = k^j(x_*^j, x_*^j) - k^j(x_*^j, X^{(j)})\Delta^{-1}k^j(X^{(j)}, x_*^j) \tag{4.5}$$

where $k(a,b)$ is the piecewise kernel operator for vectors or matrices a and b and $\Delta = k(X,X) + \eta I_n$. A single GP with multiple kernels (where each kernel handles a different dimension of $dot(Q^T, x)$) is used. There are $k^{j=1,\ldots,q+1}$ kernels (the $+1$ comes from the first kernel being the kernel for the active subspace).

Using this information about each individual kernel component results in the simple additive mean and covariance functions, which can then be used for optimization by UCB:

$$\mu(x) = \sum_{i=1}^{M} \mu^{(i)}(x^{(i)}) \tag{4.6}$$

$$\kappa(x,x') = \sum_{i=1}^{M} \kappa^{(i)}(x^{(i)}, x'^{(i)}) \tag{4.7}$$

One should notice that this additive acquisition function is an approximation of the real acquisition function. For lower dimensions - such as $d < 3$ - it is not required to decompose the acquisition function into separate additive components.

How does our algorithm address the shortcomings from chapter 3?

1. Our algorithm intrinsically uses multiple restarts. As such, bad initial states and bad projection matrices are discarded as better ones are identified. This makes our algorithm more reliable than algorithms like naive REMBO (without interleavings).

2. Our algorithm allows to not only optimize on a given domain, but also identify the subspace on which the maximal embedding is allocated on. In addition to that, no gradient information is needed.

3. Our algorithms uses a "burn-in-rate" for the first few samples, which allows for efficient point search at the beginning, and later on switches to finding the actual, real subspace. This means that we need to compute the embedding only once, and can then apply optimization on that domain. Our algorithm allows for comfortable choice of how much computation should be put into identifying the subspace.

4. Our algorithm is more accurate and robust, as we don't assume that there is a singular maximal subspace. We also take into consideration that there might be perturbation on lower dimensions! In that sense, our algorithm mimicks the idea of projection pursuit [7], as it identifies multiple vectors to project to the lower subspace.

Chapter 5

Evaluation

5.1 Evaluation Settings

Appendix A presents a list of synthetic functions and real datasets that are used to evaluate the effectiveness of a Bayesian Optimization algorithm. I conduct experiments in the following settings as mentioned in chapter 3.2.1.

5.2 Quantitative evaluation

To recapitulate, I will use log-likelihood, angle-difference measure and cumulative regret to compare the performance of different algorithms. We present how the different algorithms perform on the regret measure using UCB as the acquisition function. It is important to point out that all experiments capped the matrix identification step to about 30 minutes. This is much less than in the original papers that we base the algorithm on. The reason for this is that we want to have an acceptable comparison for medium-sized experiments, where time and computational resources can be restrictive (like on a users laptop).

I want to have an indication of whether the contribution of the performance comes from our subspace identification, or from the algorithm. For this, I start the discussion of every function with a plot that shows how the algorithm performs when the real subspace matrix W_{true} is assumed to be found (tripathy's algorithm does not run, instead we return the W_{true} instead of an approximated $\hat{W}$.

To keep the measurements fair across algorithms, I fix the noise variance of the GP, and the kernel hyperparameters for each function.

I have multiple independent runs for each functions. However, as most of the runs show similar results, I display only one of them unless there is something interesting to see. The reader should notice that the individual runs do carry the same kernel parameters (unless an algorithm-specific function decides to change these). This means that the algorithms should theoretically have similar properties as UCB on the vanilla function, if the active subspace is identified or the dimensionality is effectively reduced.

In all of these graphs, we apply the subspace identification at the 100th timestep. This means that we use the first 100 sampled points from UCB to identify a subspace. Any other future projected point is projected onto this subspace, before one optimization is taken.

Due to numerical errors recognized at later stages, the maximum dimension that I test over is 5. When I set the real dimensionality of an environment (not the active dimensionality), the property $W^T \times W = I$ is violated. Apart from that, all algorithms were implemented from scratch. Any implemented gradients were unittested, and the respective analytical gradient was (successfully) compared with their numerical gradients. To test the functionality of the subspace identification algorithm, I also wrote tests to make sure that the log-likelihood increases, and that some other properties presented in the respective paper are satisfied.

5.2.1 Parabola

The function to be learned and optimized over is the following:

$$f(x) = \left(\begin{bmatrix} 0.500 \\ 0.192 \end{bmatrix}^T x \right)^2 \tag{5.1}$$

where we have $x \in \mathbf{R}^2$ and $W \in \mathbf{R}^{2\times 1}$.

Assume $\hat{W} = W_{\text{true}}$: I present how the respective algorithms perform if we assume that Tripathy's Stiefel Manifold optimization finds the perfect matrix. This measures how the algorithm performs when we assume perfect subspace identification.

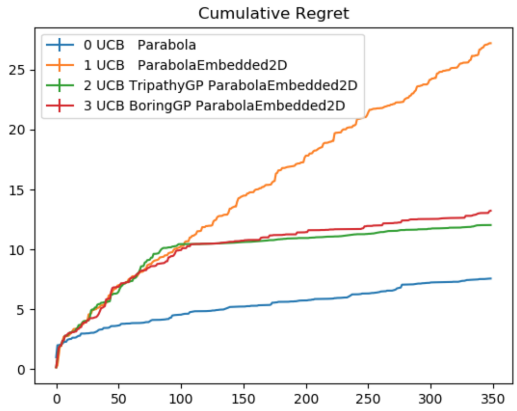


Fig. 5.1 UCB on a Parabola embedded in 2D space, when we assume that tripathy's method finds the real projection matrix.

Assume $\hat{W} \neq W_{\textbf{true}}$: I now proceed with how different algorithms perform on the function described above. This measures how the algorithm performs, when subspace identification is a part of the optimization process.

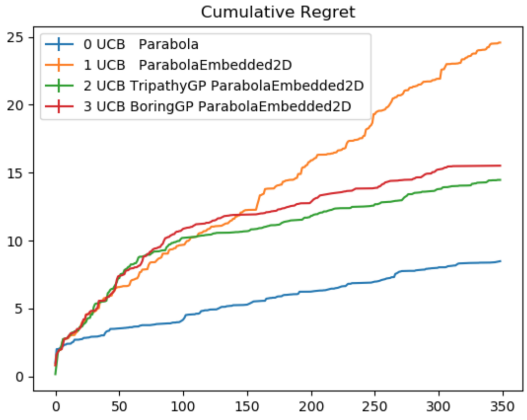


Fig. 5.2 UCB on a Parabola embedded in 2D space. Tripathy's algorithm is applied to find the a projection matrix $\hat{W}$.

One can easily see that the performance on UCB using tripathy's matrix identification algorithm is similar to the case, when we assume that tripathy executes perfectly.

The log likelihood of the GP w.r.t the collected datapoints of the tripathy GP with the real matrix is comparable to the log-likelihood of the GP of the tripathy model, where the active projection matrix is calculated using the algorithm (values of -1.37 and -1.38 or for a different run values of 195.32 and 210.25, where ranges are between -100 and 700). One should notice, however, that the angle between the found matrix and the real projection

matrix is almost always at 45 - a value that does not sound very intuitive, and for which the only reasonable explanation is that the optimization problem stays the same at this projection angle. The reader can view graphs in a subsequent subsection.

5.2.2 Camelback embedded in 3D

The function to be learned and optimized over is the following:

$$f(z_1, z_2) = \left(4 - 2.1 * z_1^2 + \frac{z_1^4}{3}\right) z_1^2 + z_1 * z_2 + \left(-4 + 4 * z_2^2\right) * z_2^2 \tag{5.2}$$

where we have $z = \left(\begin{bmatrix} -0.46554187 & -0.36224966 & 0.80749362 \\ 0.69737806 & -0.711918 & 0.08268378 \end{bmatrix}^T x \right)$, and $x \in \mathbf{R}^3$, $W \in \mathbf{R}^{2\times 3}$ and where z_1 denotes the first entry of the z vector, and z_2 denotes the second element of the z vector.

Assume $\hat{W} = W_{\text{true}}$: Again, I present how the respective algorithms perform if we assume that Tripathy's Stiefel Manifold optimization finds the perfect matrix.

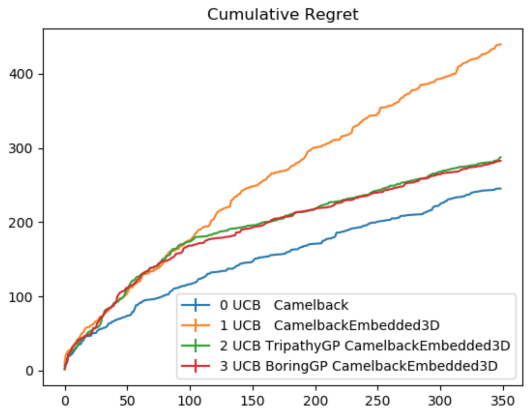


Fig. 5.3 UCB on a 2D Camelback function embedded in 3D space. This is when we assume that tripathy finds the real projection matrix W_{true}

Assume $\hat{W} \neq W_{\text{true}}$: Again, I now proceed with the performance, when the subspace identification is part of the optimization process.

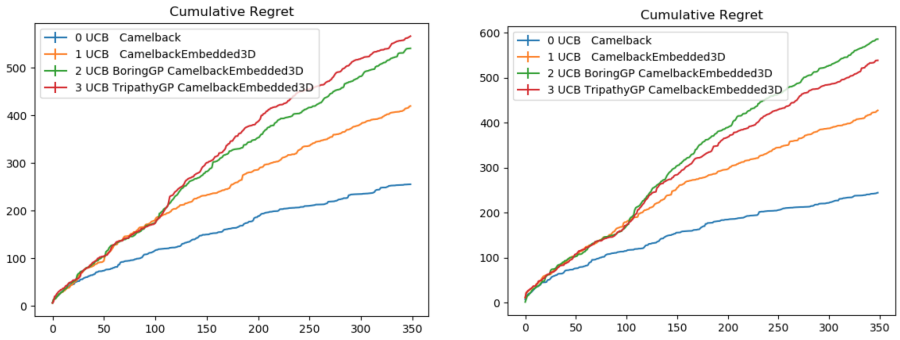


Fig. 5.4 UCB on a 2D Camelback function embedded in 3D space. We apply tripathy's algorithm to find a projection matrix W.

One can see that the subspace projection of tripathy's method from 3D to 2D is not efficient. The difference between BORING and Tripathy is marginal, as they both rely on a similar algorithm. This is an indication that the subspace projection is not close to the real subspace, but potentially just finds a subspace which is acceptable when higher dimensions are taken into consideration. To investigate this further, we increase the dimensionality of the domain in the next section.

5.2.3 Camelback embedded in 5D

$$f(z_1,z_2) = \left(4 - 2.1 * z_1^2 + \frac{z_1^4}{3}\right) z_1^2 + z_1 * z_2 + \left(-4 + 4 * z_2^2\right) * z_2^2 \tag{5.3}$$

where we have $z = \left(\begin{bmatrix} -0.31894555 & 0.78400512 & 0.38970008 & 0.06119476 & 0.35776912 \\ , -0.27150973 & 0.066002 & 0.42761931 & -0.32079484 & -0.79759551 \end{bmatrix}^T x \right)$, and $x \in \mathbf{R}^5$, $W \in \mathbf{R}^{2\times 5}$ and where z_1 denotes the first entry of the z vector, and z_2 denotes the second element of the z vector.

Assume $\hat{W} = W_{\text{true}}$: The following shows how tripathy performs when we assume perfect projection-matrix identification.

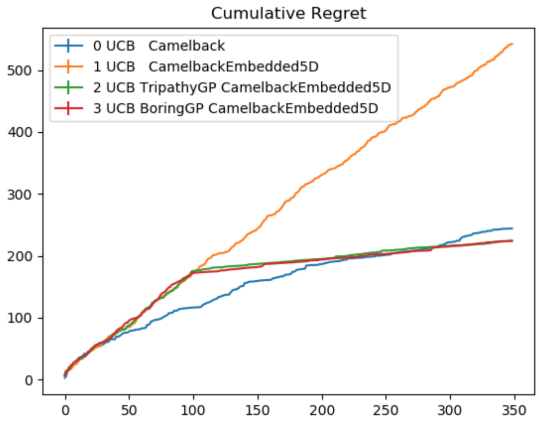


Fig. 5.5 UCB on a 2D Camelback function embedded in 5D space. This is when we assume that tripathy finds the real projection matrix W_{true}

The linear curve may be the result of kernel parameters that were not set very well, which may lead to the same point chosen repeatedly over and over again. However, because these kernel parameters are chosen by the algorithm, I do not modify these to get square-root behaved UCB curves.

Assume $\hat{W} \neq W_{\text{true}}$: The following curves present the performance of UCB when subspace identification is part of the optimization process.

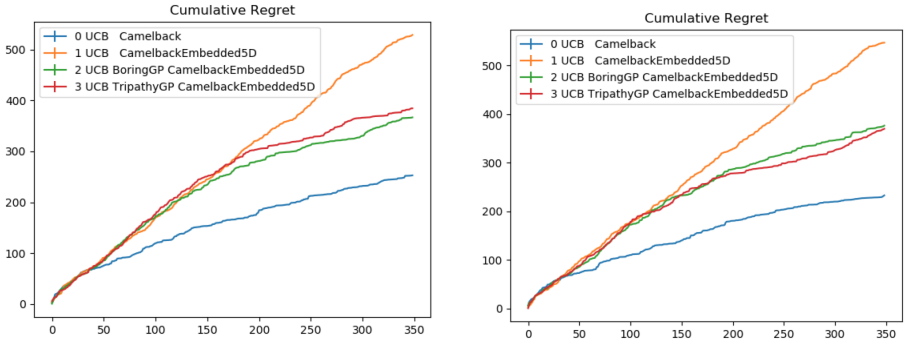


Fig. 5.6 UCB on a 2D Camelback function embedded in 5D space. This is when we apply tripathy's algorithm to find a projection matrix W, that is acceptable for optimization, but is not near close to the real projection matrix.

One can see for for higher dimensions, tripathy's method performs well, as it is able to reduce the dimensionality of the optimization problem. However, one can see that the regret achieved from the empirical projection-matrix identification (the real projection-matrix is not

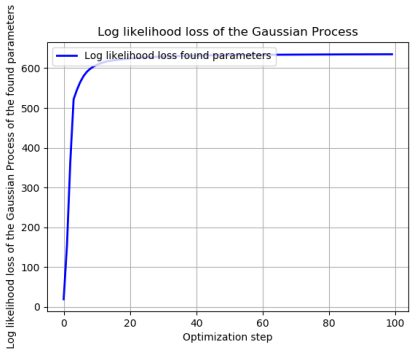
found) is much higher. This poses the question of how well the found projection matrix is compared to the real projection matrix. I analyse the log-likelihood of the GP to get a quick answer.

The log likelihood of the GP w.r.t the collected datapoints of the tripathy GP with the real matrix is not comparable to the log-likelihood of the GP of the tripathy model, where the active projection matrix is calculated using the algorithm. For different runs, the log-likelihood of the GP including the real matrix is at -2 and -389, whereas the log-likelihood of the GP with the estimated projection matrix is at -0.92 and -102. Although the values -2 and -0.92 are close to each other, the pair $(-389, -102)$ shows that the subspace identification task for this algorithm is much more unstable than that for the parabola. In the following subsection, I will investigate this issue further.

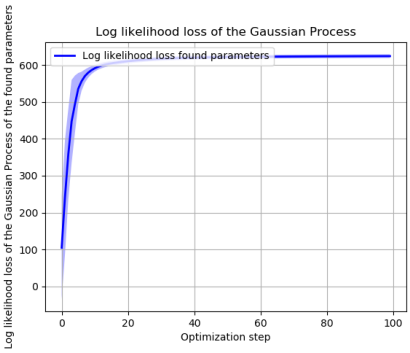
5.2.4 Log-Likelihood and Angle difference measures

An interesting quantity to take into consideration is the log-likelihood of the sampled data with respect to the GP, and the angle between the found projection matrix, and the real projection matrix. In the following, I describe how these quantities change over a function of time. More specifically, the time refers to the number of steps that I allow for tripathy's method to optimize over these parameters.

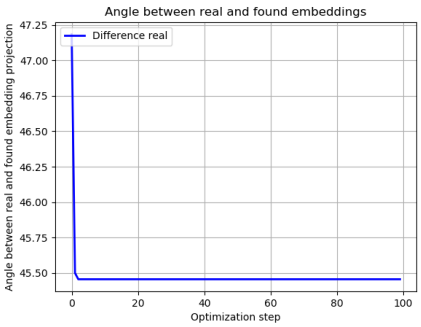
Parabola From the graphs, we can see that the parabola always seems to converge at a projection that is at a 45° to the real projection matrix. Intuitively, this seems odd. One should notice, that the maximum of the optimization problem is the same, when the space is rotated by 45°. Another explanation could be that 100 datapoints on a 2D space are numerous enough, such that any matrix (that does not directly map to the nullspace of the real projection matrix), is an acceptable matrix). Why the algorithm always converges at a matrix at 45° to the real projection matrix, would be unclear however in this case.



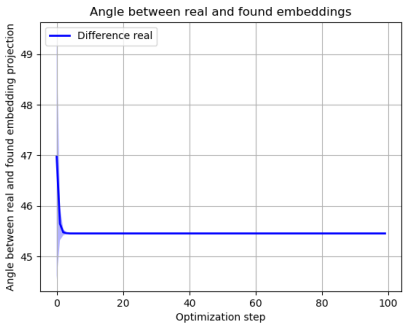
(a) The momentary W from tripathy's algorithm, and it's log-likelihood w.r.t. the sampled data



(b) The average of all momentary W from tripathy's algorithm, and it's log-likelihood to the sampled data



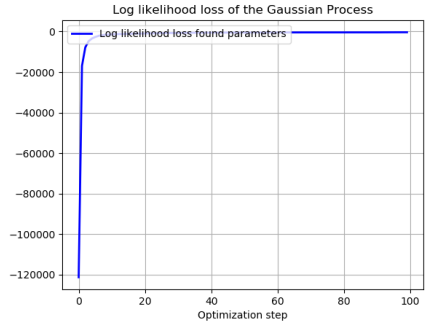
(c) The momentary W from tripathy's algorithm, and it's angle to the real projection matrix



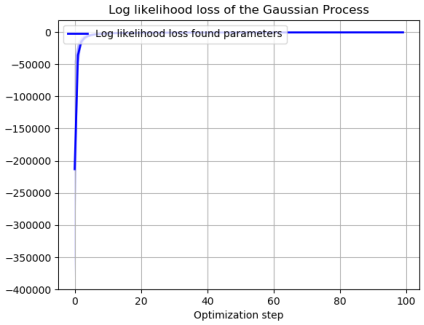
(d) The average of all momentary W from tripathy's algorithm, and it's angle to the real projection matrix

Fig. 5.7 Log-Likelihood (top) and Angle (bottom) performance measures for a 1D Parabola embedded in a 2D space. The left graphs show the values for the run that was chosen as the "found" projection matrix. The right graphs show the average values over all restarts of tripathy's method.

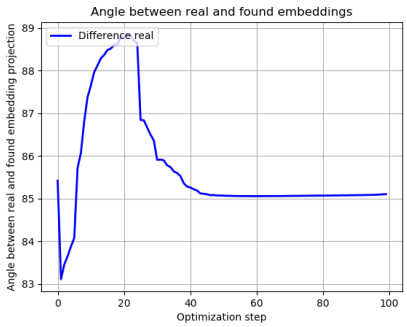
Camelback Because from the UCB experiments, we assume Camelback to be more instable, we show the results of two independent runs that exhibit different behavior. This is evidence, that tripathy's algorithm on Camelback does not run stably, and has high variance (i.e. is not as robust).



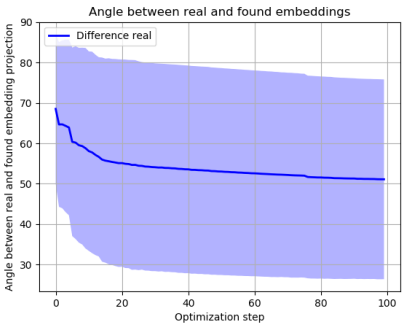
(a) The momentary W from tripathy's algorithm, and it's log-likelihood w.r.t the sampled data



(b) The average of all momentary W from tripathy's algorithm, and it's log-likelihood w.r.t the sampled data

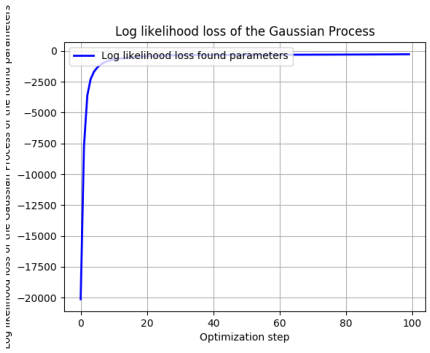


(c) The momentary W from tripathy's algorithm, and it's angle to the real projection matrix

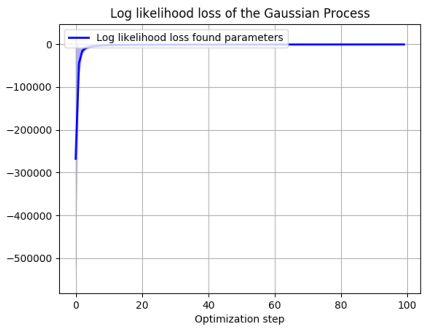


(d) The average of all momentary W from tripathy's algorithm, and it's angle to the real projection matrix

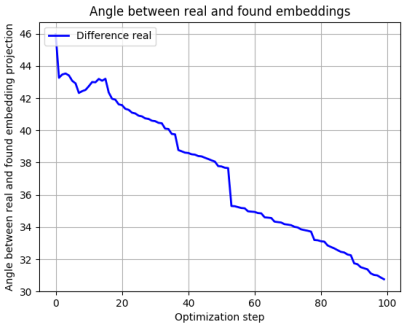
Fig. 5.8 Log-Likelihood (top) and Angle (bottom) performance measures for a 2D Camelback embedded in a 5D space. The left graphs show the values for the run that was chosen as the "found" projection matrix. The right graphs show the average values over all restarts of tripathy's method. These are the results for run 1.



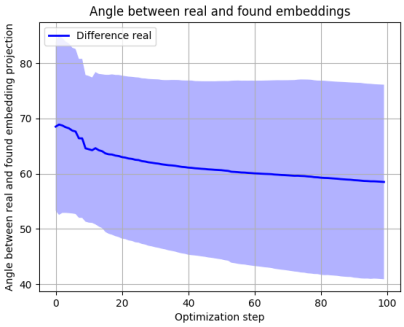
(a) The momentary W from tripathy's algorithm, and it's log-likelihood w.r.t the sampled data



(b) The momentary W from tripathy's algorithm, and it's angle to the real projection matrix



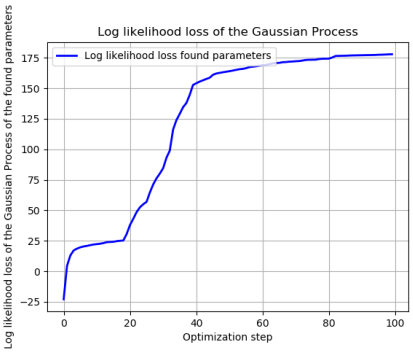
(c) The average of all momentary W from tripathy's algorithm, and it's log-likelihood w.r.t the sampled data



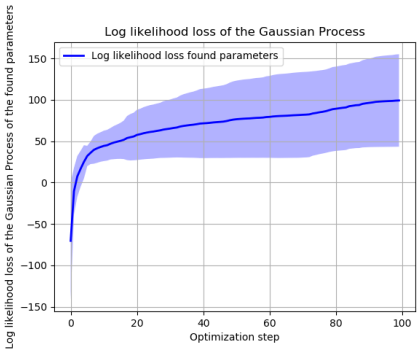
(d) The average of all momentary W from tripathy's algorithm, and it's angle to the real projection matrix

Fig. 5.9 Log-Likelihood (top) and Angle (bottom) performance measures for a 2D Camelback embedded in a 5D space. The left graphs show the values for the run that was chosen as the "found" projection matrix. The right graphs show the average values over all restarts of tripathy's method. These are the results for run 2.

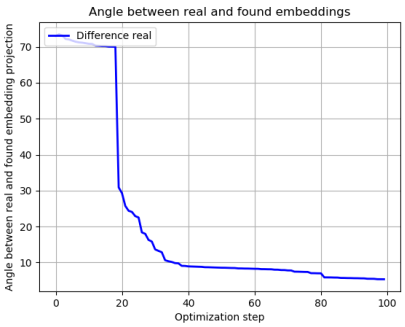
Sinusoidal : Although we don't show the UCB curves for the sinusoidal (which also prove to be successful, when this 2d function is hidden within a 5D space), good results can be obtained for subspace identification.



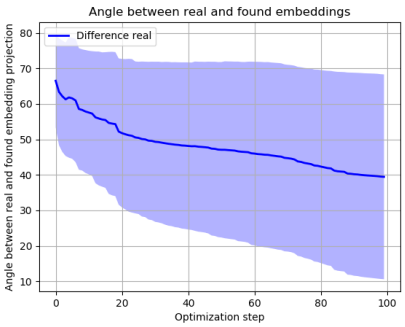
(a) The momentary W from tripathy's algorithm, and it's log-likelihood w.r.t the sampled data



(b) The average of all momentary W from tripathy's algorithm, and it's log-likelihood w.r.t the sampled data



(c) The momentary W from tripathy's algorithm, and it's angle to the real projection matrix



(d) The average of all momentary W from tripathy's algorithm, and it's angle to the real projection matrix

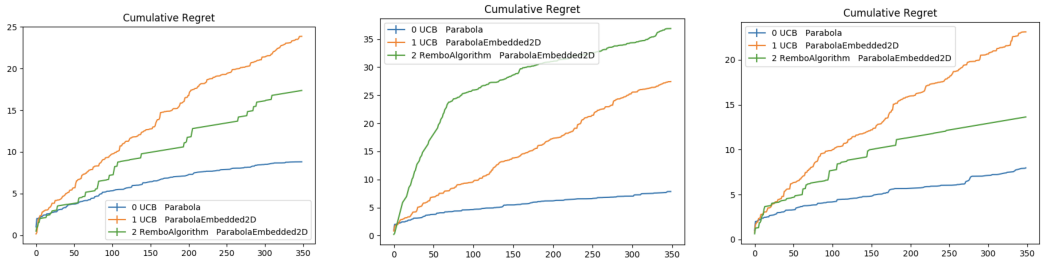
Fig. 5.10 Log-Likelihood (top) and Angle (bottom) performance measures for a 2D Sinusoidal function embedded in a 5D space. The left graphs show the values for the run that was chosen as the "found" projection matrix. The right graphs show the average values over all restarts of tripathy's method. These are the results for run 2.

5.3 REMBO

The final algorithm I am investingating is REMBO, as described in chapter 2. REMBO generally finds good embeddings under the assumption, that optimization domain is normalized, and scaled to $[-\sqrt{d}, \sqrt{d}]^d$ where d is at least the effective dimensionality of the optimization function at hand.

I shortly present results for UCB that use REMBO as their optimization algorithm. The high probability of failing implies a high variance amongst runs. As such, I perform 3 runs for each function addressed in the above section.

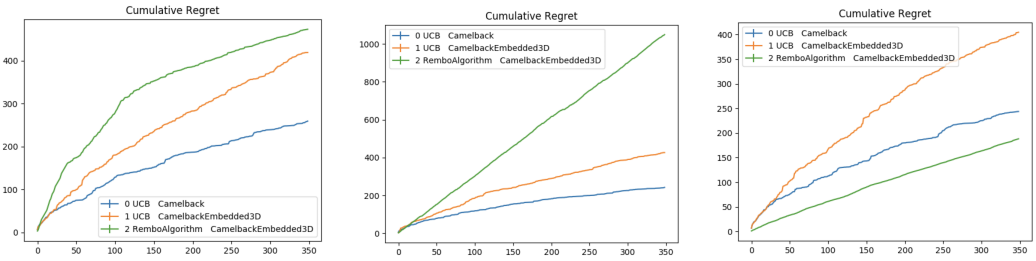
Parabola As one can see, REMBO usually finds an embedding that is acceptable and accelerates the optimization process. However, run 2 shows that it can also fail in the simplest case of the parabola. In this regards, tripathy's method is more robust, as it does use some sort of heuristic to check if the chosen matrix delivers good results.



(a) Run 1 (b) Run 2 (c) Run 3

Fig. 5.11 UCB using REMBO on a 1D Parabola embedded in a 2D space.

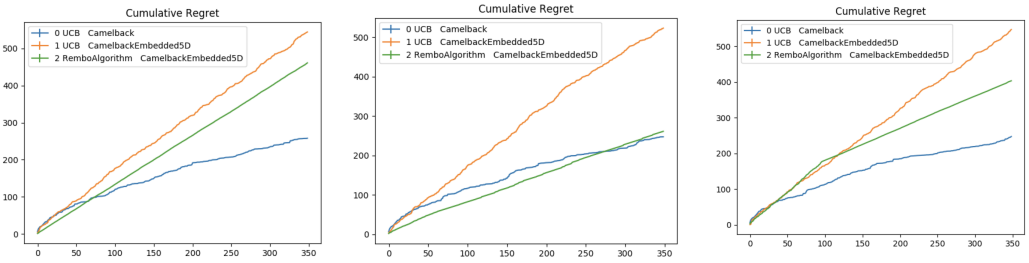
Camelback3D This example illustrates how REMBO can find an acceptable subspace. However, the linear lines are an indication for the hyperparameters to be no well chosen. I do not allow hyperparameter optimization, for the reason such that the results may be comparable to the other algorithm's results. As one can see, Camelback3D also provides a difficult for REMBO, although REMBO does find an acceptable subspace projection.



(a) Run 1 (b) Run 2 (c) Run 3

Fig. 5.12 UCB using REMBO on a 2D Camelback embedded in a 3D space.

Camelback5D The linearity of the UCB curves is still attained. Similar to the result with tripathy's method, REMBO is able to reduce the dimensionality of the subspace in almost any case. the effectiveness of this reduction can be small thought. The viewer can recognize well that there is high variance in the performance amongst runs. This emphasizes how important the interleaved runs are, even though the number of datasamples per projection are divided by the total number of projection (i.e. there is slower learning of the GP surface, however, dimensionality reduction is successful).



(a) Run 1 (b) Run 2 (c) Run 3

Fig. 5.13 UCB using REMBO on a 2D Camelback embedded in a 5D space.

5.4 Qualitative evaluation

It is interesting to see, that amongst the 1000 restarts that I generated, some of the resulting matrices are close to the real projection matrix up to an absolute value of 0.01. However, because the algorithm decides to choose the matrix with the highest likelihood, tripathy's algorithm in general does not select these matrices, but chooses a matrix that is not amongst the matrices that are very similar to the real matrices.

5.4.1 Feature selection

The goal of this task is to see, if the revised active subspace identification algorithms can effectively do feature selection. For this task, I set up a function f that looks as follows:

$$f\left(W\begin{bmatrix}(x-a_0)^2\\(y-a_1)^2\end{bmatrix}\right)\approx g(x) \tag{5.4}$$

where x_0, x_1 are constants.

For this specific experiment, the function f is chosen to be a one-dimensional parabola. As such, W is chosen as a matrix on the Stiefel manifold with dimensions $\mathbf{R}^{d\times D}$

Doing a feature extension over x and y, we can get the following feature representation:

$$\begin{bmatrix} x_0^2 \\ x_1^2 \\ x_0 \\ x_1 \\ 1 \end{bmatrix} \tag{5.5}$$

Fig. 5.14 Polynomial Kernel applied to vector $[x_0, x_1]$

$$\begin{bmatrix} w_0 \\ w_1 \\ -2w_0a_0 \\ -2w_1a_1 \\ w_0a_0^2 + w_1a_1^2 \end{bmatrix} \tag{5.6}$$

Fig. 5.15 Corresponding weight matrix equivalent to 5.4 when applied on a parabola

To run experiments, I instantiate the "real" matrix, which should be found by the algorithm with the values $w_0 = 0.589$, $w_1 = 0.808$ (randomly sampled as a matrix on the Stiefel manifold), $a_0 = -0.1$, $a_1 = 0.1$ (chosen by me as coefficients).

I apply the algorithm 1. from [27] to identify the active projection matrix. The optimization algorithm has 50 samples to discover the hidden matrix, which it seemingly does up do a certain degree of accuracy. Similar results are achieved for repeated tries. The following figure shows the real matrix, and the matrix the algorithm has found.

$$\begin{bmatrix} 0.589 \\ 0.808 \\ 0.118 \\ -0.162 \\ 0.823 \end{bmatrix} \tag{5.7}$$

Fig. 5.16 Real matrix

$$\begin{bmatrix} -0.355 \\ -0.533 \\ -0.908 \\ 0.099 \\ -0.756 \end{bmatrix} \tag{5.8}$$

Fig. 5.17 Matrix found by optimization algorithm

Although one can see that the element wise difference between the two matrices 5.7 and 5.8 are high (between 0.05 and 0.15), one can see that the matrix recovery is successful in finding an approximate structure that resembles the original structure of the features. One should observe that the found matrix is an approximate solution to the real matrix in the projection. I.e. the matrix found is close to the real matrix, but multiplied by -1.

Because in this case, I applied the feature selection algorithm on a vector-matrix (only one column), one can quantify the reconstruction of the real matrix through the found matrix by the normalized scalar product. This quantity is a metric between 0 and 1, where 0 means that both vectors are orthogonal, and 1 means that both vectors overlap.

$$\text{overlap}(u,v) = \frac{|\langle u,v\rangle|}{\langle u,u\rangle} \tag{5.9}$$

where u is the real vector, and v is the found vector.

Inserting the actual values into the field, we get 0.79, which is a good value for the feature vector found, and the trained number of datapoints which is 50.

This experiment shows that algorithm 1. from [27] successfully allows a viable option to other feature selection algorithms, by providing a measure, where the optimal linear projection is found. However, one must notice that other feature selection algorithms (such as SVM), are more efficient, and will provide better results with higher probability if applied on a similar kernel.

One major observation I made was the the increase in the log likelihood of the data w.r.t. the projection matrix did not correlate with the decrease in the angle between the real vs the found projection matrix. Also, most often, the angle was at around 40 degrees, which means that only slight improvements over a fully random embedding were made.

5.4.2 Subspace identification

One of the main reasons to use our method is because we allow for subspace identification. We have the following functions:

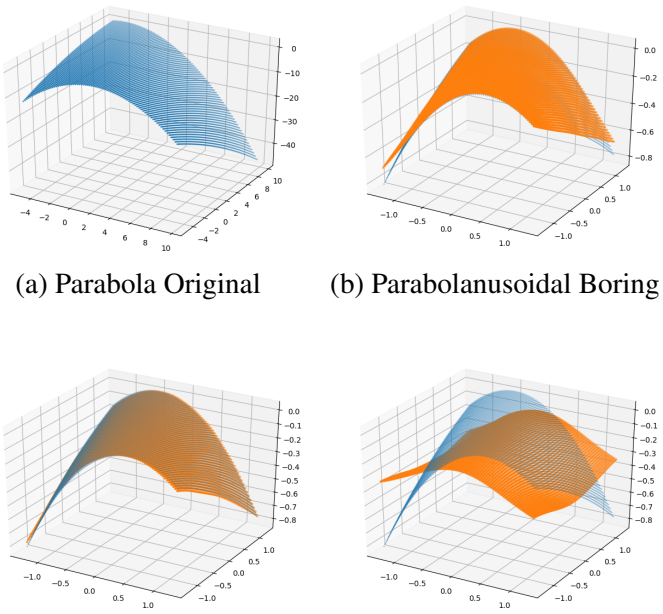
1. 1D Parabola embedded in a 2D space
2. 2D Camelback embedded in a 5D space
3. 2D Sinusoidal and Exponential function embedded in a 5D space

To be able to visualize the points, I proceed with the following procedure:

I generate testing points (points to be visualized) within the 2D-space in a uniform grid. I then project these testing points to the dimension of the original function ($2d$ for parabola, else $5d$). I then let each algorithm learn and predict the projection matrix and GP mean predictions. If because the transformation from $2D$ space, to $5D$ space and to GP mean prediction is each bijective, we can visualize the $2D$ points with the GP mean prediction

right away. As such, the dimension of the embedding learned does not have impact on the visualization!

In the following figures, blue point shows the sampled real function value. Orange points shows the sampled mean prediction of the trained GP. The GPs were each trained on 100 datapoints. The points shown below were not used for training at any point, as these are included in the test-set.

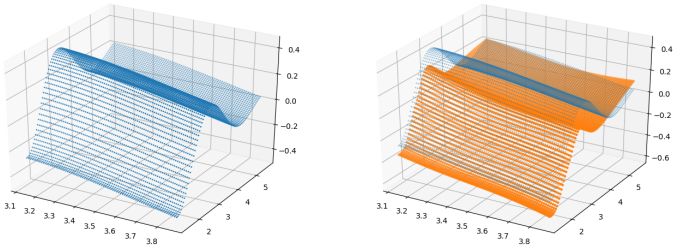


(a) Parabola Original (b) Parabolanusoidal Boring

(c) Parabola Tripathy (d) Parabola Rembo

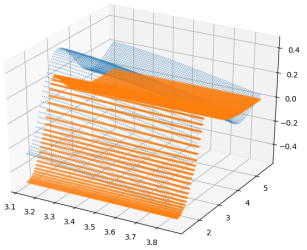
Fig. 5.18 Top-Left: The 1D Parabola which is embedded in a 2D space.

I set the number of restarts to 14 and number of randomly sampled datapoints to 100. Notice that the Tripathy approximation is slightly more accurate than the BORING approximation. This is because one of Tripathy's initial starting points were selected better, such that algorithm 3 ran many times before the relative loss terminated the algorithm. The active subspace projection matrix is of size $\mathbf{R}^{1\times 2}$

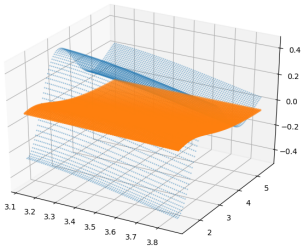


(a) Sinusoidal Original

(b) Sinusoidal Boring



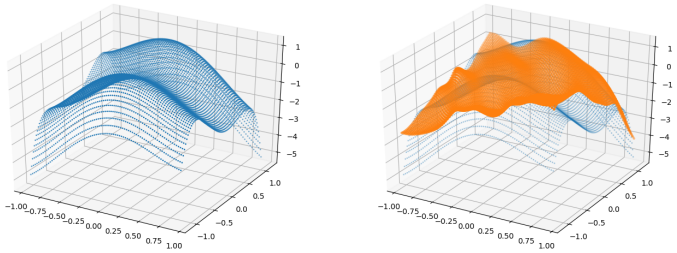
(c) Sinusoidal Tripathy



(d) Sinusoidal Rembo

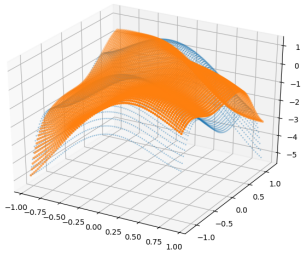
Fig. 5.19 Top-Left: The 2D Sinusoidal-Exponential Function which is embedded in a 5D space.

I set the number of restarts to 28 and number of randomly sampled datapoints to 100. The active subspace projection matrix is of size $\mathbf{R}^{1\times5}$, as this is a function that exhibits a strong principal component, but that still attains small perturbations among a different dimension. One can see very well here, that BORING is able to take into account the small perturbations, at a considerably lower cost than Tripathy would be able to.

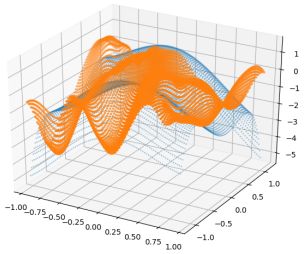


(a) Camelback Original

(b) Camelback Boring



(c) Camelback Tripathy



(d) Camelback Rembo

Fig. 5.20 Top-Left: The 2D Camelback Function which is embedded in a 5D space.

I set the number of restarts to 28 and number of randomly sampled datapoints to 100. The active subspace projection matrix is of size $\mathbf{R}^{2\times5}$, as this is a function that lives in a 2D space, and has two strong principal components. Notice that Tripathy and BORING use the exact same algorithm, as the visualization does not allow to add a third axis. In other words, BORING does not add any additional orthogonal vector to the model. As such, it does not add any additional kernels to the model aswell, and is equivalent to tripathy.

Chapter 6

Conclusion

6.1 Main Contributions

I present this thesis as a result of a systematic treatment of the topic "Bayesian Optimization in High Dimensions".

1. BORING allows the number of dimensions to be smaller than the real active subspace. Although in general, it does not perform better than other subspace identification algorithms (like tripathy's method), it does allow for better values when the subspace identification fails, as it incorporates an additional dimension which acts like a random embedding. As such, it allows for a good trade-off between subspace identification, and optimization using random projections.

2. BORING is more robust to smaller perturbations, as it does not rely on finding the exact active subspace. It allows to optimize over smaller perturbations as well, whereas current algorithms usually don't take these into consideration.

3. I analyse the relationship between improved log-likelihood and angle difference. I see that it is not surprising that the angle-difference between the real projection matrix, and the found projection matrix is not closely coupled to the log-likelihood. However, improving over the log-likelihood can reduce the angle-difference in many cases.

6.2 Future work

Future work could incorporate the synthesis of different methods, including additive GPs and Tripathy's method. Although tripathy's method is unbeaten in identifying the active subspace dimension, heuristics could be easily implemented to speed up the calculation time. One of

the most important aspects is hyperparameter tuning for the GP models itself. These can make or break the identification of the subspace. Choosing bad hyperparameters does not allow us to effectively compare different methods. In future, it would be beneficial to address this issue to a stronger extent. Recalculating the search space for each new point may be too time consuming, with which REMBO would still be considered the state of the art in terms of Bayesian Black Box Optimization.

References

[1] Abdelrahman, H., Berkenkamp, F., Poland, J., and Krause, A. (2016). Bayesian Optimization for Maximum Power Point Tracking in Photovoltaic Power Plants.

[2] Berkenkamp, F., Turchetta, M., Schoellig, A. P., and Krause, A. (2017). Safe model-based reinforcement learning with stability guarantees. In *Neural Information Processing Systems (NIPS)*.

[3] Binois, M., Ginsbourger, D., and Roustant, O. (2014). A warped kernel improving robustness in Bayesian optimization via random embeddings.

[4] Brochu, E., Cora, V. M., and De Freitas, N. (2010). A tutorial on Bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning.

[5] Camano-Garcia, G. (2006). Statistics on Stiefel manifolds.

[6] Djolonga, J., Krause, A., and Cevher, V. (2013). High-Dimensional Gaussian Process Bandits.

[7] Friedman, J. and Stuetzle, W. (1981). *Projection Pursuit Regression*. Journal of the American Statistical Association.

[8] Gardner, J. R., Guo, C., Weinberger, K. Q., Garnett, R., and Grosse, R. (2017). Discovering and Exploiting Additive Structure for Bayesian Optimization.

[9] Garnett, R., Osborne, M. A., and Hennig, P. (2013). Active Learning of Linear Embeddings for Gaussian Processes.

[10] GPy (since 2012). GPy: A gaussian process framework in python. http://github.com/SheffieldML/GPy.

[11] Graepel, T. and Herbrich, R. (2001). The Kernel Gibbs Sampler.

[12] GSeshadri, P., Yuchi, S., and Parks, G. T. (2018). Dimension Reduction via Gaussian Ridge Functions.

[13] Hemant, T. and Cevher, V. (2012). Active learning of multi-index function models. In Pereira, F., Burges, C. J. C., Bottou, L., and Weinberger, K. Q., editors, *Advances in Neural Information Processing Systems 25*, pages 1466–1474. Curran Associates, Inc.

[14] Jamil, M. and Yang, X.-S. (2018). A Literature Survey of Benchmark Functions For Global Optimization Problems Citation details: Momin Jamil and Xin-She Yang, A literature survey of benchmark functions for global optimization problems.

[15] Kandasamy, K., Neiswanger, W., Schneider, J., Poczos, B., and Xing, E. (2018). Neural Architecture Search with Bayesian Optimisation and Optimal Transport.

[16] Li, C., Gupta, S., Rana, S., Nguyen, V., Venkatesh, S., and Shilton, A. (2018). High Dimensional Bayesian Optimization Using Dropout.

[17] Lin, L., Rao, V., and Dunson, D. B. (2014). BAYESIAN NONPARAMETRIC INFERENCE ON THE STIEFEL MANIFOLD.

[18] Lin, X., Chowdhury, A., Wang, X., and Terejanu, G. (2017). Approximate Computational Approaches for Bayesian Sensor Placement in High Dimensions.

[19] Lizotte, D. (2008). Practical Bayesian Optimization.

[20] Marco, A., Berkenkamp, F., Hennig, P., Schoellig, A. P., Krause, A., Schaal, S., and Trimpe, S. (2017). Virtual vs. Real: Trading Off Simulations and Physical Experiments in Reinforcement Learning with Bayesian Optimization.

[21] Murphy, K. P. (2012). *Machine Learning, A Probabilistic Perspective*. MIT Press.

[22] Rana, S., Li, C., Gupta, S., Nguyen, V., and Venkatesh, S. (2017). High Dimensional Bayesian Optimization with Elastic Gaussian Process.

[23] Rao, V., Lin, L., Dunson, D. B., and Dunson, D. (2013). Bayesian inference on the Stiefel manifold.

[24] Rasmussen, C. E., Williams, C. K. I., Sutton, R. S., Barto, A. G., Spirtes, P., Glymour, C., Scheines, R., Schölkopf, B., and Smola, A. J. (2006). Gaussian Processes for Machine Learning.

[25] Rolland, P., Scarlett, J., Bogunovic, I., and Cevher, V. (2018). High-Dimensional Bayesian Optimization via Additive Models with Overlapping Groups.

[26] Srinivas, N., Krause, A., Kakade, S., and Seeger, M. (2009). Gaussian Process Optimization in the Bandit Setting: No Regret and Experimental Design.

[27] Tripathy, R., Bilionis, I., and Gonzalez, M. (2016). Gaussian processes with built-in dimensionality reduction: Applications in high-dimensional uncertainty propagation.

[28] University, S. F. (2017). Virtual library of simulation experiments: Test functions and datasets - optimization test problems.

[29] Wang, Z., Gehring, C., Kohli, P., and Jegelka, S. (2017). Batched Large-scale Bayesian Optimization in High-dimensional Spaces.

[30] Wang, Z., Hutter, F., Zoghi, M., Matheson, D., and De Freitas, N. (2013). Bayesian Optimization in High Dimensions via Random Embeddings.

[31] Wilson, J. T., Hutter, F., and Deisenroth, M. P. (2018). Maximizing acquisition functions for Bayesian optimization.

Appendix A

A.1 Benchmarking functions

The following is a list of synthetic functions and real datasets presented in previous works, where the goal is to evaluate bayesian optimization algorithms.

1. [9] Synthetic in-model data matching the proposed model, with $d = 2,3$, and $D = 10,20$.
2. [9] (Synthetic) Braning function, $d = 2$, hidden in a higher dimensional space $D = 10,20$.
3. [9] Temperature data $D = 106$ and $d = 2$.
4. [9] Communities and Crime dataset $d = 2$, and $D = 96$.
5. [9] Relative location of CT slices on axial axis with $d = 2$ and $D = 318$.
6. [6] (Synthetic) Random GP samples from 2-dimensional Matern-Kernel-output, embedded within 100 dimensions
7. [6] Gabor Filters: Determine visual stimuli that maximally excite some neurons which reacts to edges in the image. We have $f(x) = \exp(-(\theta^T x - 1)^2)$. θ is of size 17x17, and the set of admissible signals is d.
8. [30] (Synthethic) $d = 2$ and $D = 1 * 10^9$.
9. [30] $D = 47$ where each dimension is a parameter of a mixed integer linear programming solver.
10. [30] $D = 14$ with d for a random forest body part classifier.
11. [27] (Synthetic) Use $d = 1,10$ and $D = 10$.

12. [27] (Half-synthetic) Stochastic elliptic partial differential equation, where $D = 100$, and an assume value for d of 1 or 2.
13. [27] Granular crystals $X \in \mathbb{R}^{1000 \times 2n_p+1}$, and $y \in \mathbb{R}^{1000}$.
14. [8] (Synthetic) Styblinski–Tang function where D is freely choosable.
15. [8] (Synthetic) Michalewicz function where D is freely choosable.
16. [8] (Simulated) NASA cosmological constant data where $D = 9$.
17. [8] Simple matrix completion with $D = 3$.
18. [22] (Synthetic) Hertmann6d in [0, 1].
19. [22] (Synthetic) Unnormalized Gaussian PDF with a maximum of 1 in $[-1, 1]^d$ for $D = 20$ and $[-0.5, 0.5]^d$ for $D = 50$
20. [22] (Synthetic) Generalized Rosenbrock function $[-5, 10]^d$
21. [22] Training cascade classifiers, with $D = 10$ per group.
22. [22] Optimizing alloys $D = 13$.
23. [16] (Synthetic) Gaussian mixture function
24. [16] (Synthetic) Schwefel's 1.2 function.
25. [28] A list of optimiztation test functions can be found here.
26. [14] A more comprehensive list of general functions can be found here.