Improving Existing Bayesian Optimization Software by Incorporating Gradient Information

University of Colorado at Boulder

Computer Science Undergraduate Senior Thesis

Author: Alexey Yermakov

Affiliation: Computer Science, Applied Mathematics

Advisors:

Stephen Becker — Applied Mathematics

Eric Rozner — Computer Science

May 2, 2023

1. Acknowledgements.

I am extremely thankful to Dr. Stephen Becker. I had initially agreed to be a part of a project for my senior thesis before the summer of my senior year; however, as the summer rolled around I realized that the project I initially chose was not one which I thought would help push me in the direction of research I wanted to do in graduate school. So, I emailed Stephen out of the blue and to my surprise he agreed to be my advisor. He had no reason to commit to supporting me for an entire year, yet he gave me the perfect project to work on which I found both challenging and rewarding. Furthermore, Dr. Becker was generous enough to introduce me to his colleagues while he was on sabbatical, providing me networking and learning opportunities which I would not have imagined on my own. Stephen's guidance has been invaluable during the past year and I will forever be grateful for his willingness to support me in this capstone project.

I am also much obliged to Dr. Eric Rozner. When I first began looking for projects to work on, I went to him first since he is the Computer Science faculty to whom I look up to the most. He got me in touch with his graduate student Erika Hunhoff, which resulted in the initial project idea for my capstone. However, when I realized I wanted to switch research directions, Eric was more than happy to still be my CS advisor on paper so that I could pursue research I was passionate about. This allowed me to complete my CS capstone with Dr. Becker who was initially only Applied Mathematics faculty.

Moreover, I am thankful to Dr. Luigi Nardi and his research group. Dr. Nardi was more than happy to discuss this project with me and to extend his support in understanding the field as a whole. He introduced me to his graduate students and postdocs who work on the Hypermapper software package and even invited me to a few group meetings where they discussed Bayesian Optimization literature.

Furthermore, I am grateful to Dr. Joseph Salmon. Dr. Salmon also met with me about this project when it was an early idea. His expertise and guidance were critical in forming the initial ideas and motivations that resulted in this thesis.

2. Foreword. Machine learning is currently a hot field. It seems like every week there are new advancements in state-of-the-art models that each have broad societal implications. Having a background in Computer Science (CS) and Applied Mathematics (APPM), I am in a position where I can understand and potentially develop such models. A career in machine learning is very much an academic one, where there is a requirement to do cutting-edge research at the intersection of Computer Science and Applied Mathematics. Having done research in the Dispersive Hydrodynamics lab my Junior Year, I learned that I enjoy doing research. Doing the Computer Science Thesis Capstone is an excellent opportunity for me to pivot towards a career in machine learning research.

3. Motivation. Machine learning models are not all the same. In fact, the architecture of a model has a significant impact on the performance obtained. These differences can be small, such as adding an extra layer in a multi-layer perceptron (MLP), or large, such as the difference between a convolutional layer versus a vision transformer layer in image classification models. There are also other considerations, such as the type of loss function used during training, the optimizer used, or the learning rate of the chosen optimizer [15]. Furthermore, all of these options can be mixed and matched, resulting in a combinatorial explosion in the number of different configurations that can be used in specifying a machine learning model. Mathematically speaking, having one parameter which can be a real number on its own results in an uncountably infinite number of choices in determining a model. These choices are called hyperparameters. A parameter (without the 'hyper' prefix) is something that the model learns during training time. This typically is something like the values of a matrix, called a weight matrix. The architecture of a model determines that such a weight matrix exists, but the machine learning model itself learns what these values should be. We will be focusing on hyperparameters, which are not learned by a model, in this thesis.

As mentioned previously, there are a variety of different types of hyperparameters. There are discrete hyperparameters, like the number of different layers in a multi-layer perceptron, which can only be whole numbers. There are also continuous hyperparameters, such as the learning rate of a loss function, which lie on some interval in the real numbers. Types of functions can be hyperparameters too, such as the optimization algorithm and/or the activation function. Furthermore, there are constrained hyperparameters, which can be either discrete or continuous, that depend on other hyperparameters. An example of this would be limiting the total number of parameters in a model by requiring the product of the number of layers in a MLP model c_1 with the number of neurons at each layer c_2 to be less than some value α: $c_1 \cdot c_2 \leq \alpha$.

Traditionally, all of these considerations were handled by researchers. A machine learning model would be developed, it would be trained, and the final performance results would be obtained. The researcher would then have some intuition about what hyperparameters to tinker with to squeeze more performance out of their model. This method is called hyperparameter tuning. After several iterations some stopping criteria would be reached by the researcher, either being satisfied with their results, running out of time to modify the hyperparameters, or giving up entirely as a result of poor performance. Fortunately, there have been several developments in the field of Automated Machine Learning (AutoML) and human beings do not necessarily need to be involved in hyperparameter tuning anymore. One of these developments is the adoption of the technique called Bayesian Optimization [6].

We will consider a regularized Least-Squares method called Ridge Regression/Tikhonov regularization to explore a continuous hyperparameter. In our example, we will assume we have n unique vectors of dimension p. This will be our training input data X. We are also given some weight matrix $\mathbf{w}$, which in this case is just another vector of dimension p. Then, our training output data is generated by a matrix multiplication of X and $\mathbf{w}$ with the addition of Gaussian noise, we will call this $\mathbf{y}$: $\mathbf{y} = X\mathbf{w} + \mathbf{z}$, where $\mathbf{z} \sim \mathscr{N}(\mathbf{0}, I)$. The goal of our example is to try to learn what $\mathbf{w}$ is just from X and $\mathbf{y}$. This can be done by minimizing some loss function. In a general sense, our loss function is:

$$\mathscr{L}(\mathbf{w}) = \mathbb{E}_{\mathbf{x},y\sim\mathscr{D}}\ell(\mathbf{w}, (\mathbf{x}, y)) \tag{3.1}$$

where the total loss $\mathscr{L}(\mathbf{w})$ is the expected value of the individual loss ℓ of training data $\mathbf{x}$ and y, which are random variables from some unknown distribution $\mathscr{D}$, and some weight matrix $\mathbf{w}$. In the motivating example, our training data X and $\mathbf{y}$ are i.i.d. samples from the distributions $\mathbf{x}$ and y. Since we don't typically know the true distribution $\mathscr{D}$ we can approximate it with the empirical risk:

$$\hat{\mathscr{L}}(\mathbf{w}) = \frac{1}{n}\sum_{i=1}^{n}\ell(\mathbf{w}, (\mathbf{x}_i, y_i)) \tag{3.2}$$

Ridge Regression can be used in this scenario quite easily. The following formula for Ridge Regression minimizes the summation term in (3.2), effectively using Ridge Regression to minimize the loss function ℓ:

$$\arg\min_{\mathbf{w}} ||X\mathbf{w} - \mathbf{y}||_2^2 + \lambda||\mathbf{w}||_2^2, \quad \lambda \in (0, \infty) \tag{3.3}$$

Equation (3.3) admits a closed-form formula that is combined with (3.2) to give the empirical risk using Ridge Regression:

$$\hat{\mathscr{L}}(\hat{\mathbf{w}}) = \frac{||X\hat{\mathbf{w}} - \mathbf{y}||_2^2}{2n} \tag{3.4}$$

where

$$\hat{\mathbf{w}} = (X^T X + \lambda I)^{-1} X^T \mathbf{y}, \quad \lambda \in (0, \infty) \tag{3.5}$$

The value λ is a continuous hyperparameter that we can modify to influence the final weights $\hat{\mathbf{w}}$ from the

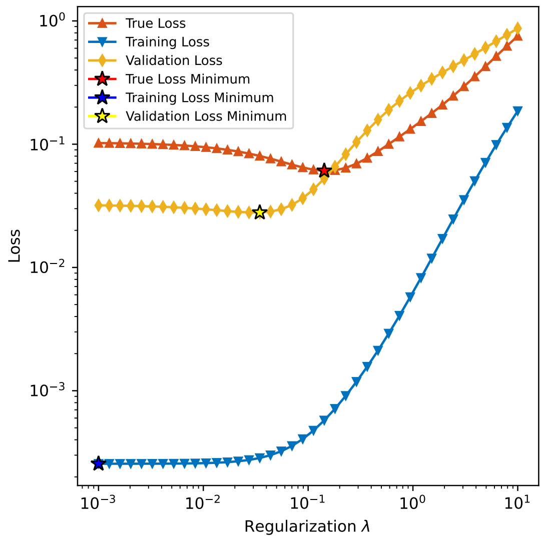


Fig. 3.1: The loss of the test problem plotted for the validation data (Validation loss), the training data (Training loss), and true data (True loss). The true loss is obtained by computing the loss from a large set of input/output samples (this set is much larger than the validation set, at about 200 times more samples). The validation loss is obtained by computing the loss from a smaller set of input/output examples (about 9 times less data than the training data).

training process! A small value for λ allows the value of $\hat{\mathbf{w}}$ from (3.5) to better fit the data. However, when the data are noisy larger values for λ influence the individual weights in $\hat{\mathbf{w}}$ to be smaller, which can be beneficial for finding a more generalizable solution.

To illustrate the effect of regularization, we assume the training data X is from a standard normal distribution and that $\mathbf{w}$ is generated from a standard uniform distribution. We will estimate the value for $\mathbf{w}$ from our training data X and $\mathbf{y}$ and calculate the loss from (3.4) for the training data (the training loss), for a set of data generated the exact same way as the training data but 200 times larger (to approximate the true loss), and for a set of data generated the same way as the training data but 9 times smaller (for the validation loss). Figure 3.1 demonstrates the effect different values of λ have on the true loss, the validation loss, and the training loss.

First, the training loss from Figure 3.1 decreases as λ decreases, since the regularization term from equation (3.3) adds to the final loss, which is plotted in blue. Secondly, the minimum of the true loss is for some $\lambda > 0$ since regularization helps generalize beyond just the training data. Typically, the true loss is not known and is estimated by validation datasets. What is interesting, however, is that the minimum of the validation loss is at a different location than the true loss. This sheds light on a situation which is typically encountered in machine learning, where the model is overfitting to a subset of the underlying distribution of data that could be encountered, the training and/or validation data, which might not be representative of the whole set of data that can be encountered. The goal, however, is to choose a set of hyperparameters that will result in the minimum of the true loss, not the training or validation loss. Drawing mini-batches of the validation data can help obtain a better approximation of the true loss minimum, but introduces more noise in the evaluations of the loss function. Further, the loss function $\hat{\mathscr{L}}(\hat{\mathbf{w}})$ is differentiable with respect to λ, which provides additional information about the behavior of the loss function than simple observations [2].

This example illustrates that there exist problems which have noisy function evaluations and provide gradient information. Utilizing mini-batches of validation data with gradient evaluations of the hyperparameter(s) being optimized to minimize the true loss is a special case in optimization literature. These problems have been understudied in the context of Bayesian Optimization, motivating the exploration of incorporating gradient information in Bayesian Optimization in this thesis.

4. Bayesian Optimization Theory.

4.1. Bayesian Optimization High-level.

Bayesian Optimization (BayesOpt) is a technique that can be used to optimize finite-dimensional black-box/latent functions. BayesOpt typically doesn't use gradient information, can be applied to convex and non-convex functions, and requires a bounded domain to optimize over (typically a hyper-rectangle).

One common application of Bayesian Optimization is to automate the task of hyperparameter tuning. This is done by taking existing evaluations and modeling the underlying (latent) function with a Gaussian Process (GP): a model describing the probability distribution of functions within which we believe the latent function exists. Figure 4.1 shows a Gaussian Process fitted to noisy observations of the function $f(x) = \sin(x)$ on the interval $[0, 2\pi]$.

Then, once the Gaussian Process is fitted to the data, an acquisition function is used to determine the location of the next point to sample (the candidate point) with the goal of finding the global maxima [6]. Going back to the motivating example, a one-dimensional Gaussian Process would be fit to the observations (the mini-batch loss at certain λ's) to create a probability distribution of functions for the true loss. This would automate hyperparameter tuning by determining a new candidate point each iteration, hopefully converging toward a global *minima* for the loss function (BayesOpt literature is concerned with finding a maximum value, while machine learning is concerned with finding a minimum; we can switch back and forth by multiplying the function evaluations by -1).

4.2. Multivariate Normal Distributions.

To understand Gaussian Processes, we need to understand Multivariate Normal Distributions (MVNs). Recall that the probability distribution function (pdf) for a one-dimensional Gaussian $\mathscr{N}(\mu, \sigma)$ is:

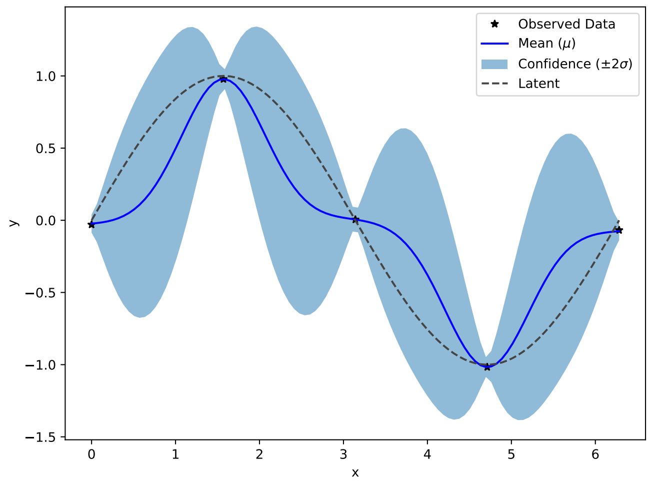


Fig. 4.1: A one-dimensional Gaussian Process (GP) fitted on noisy data. The original function and sampled data are shown, as well as the mean and two standard deviations from the mean of the GP. In this case, each x-value has a corresponding gaussian distribution across the y-values.

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{4.1}$$

where μ is the mean and σ is the standard deviation. This can be generalized to a d-dimensional Gaussian $\mathscr{N}(\boldsymbol{\mu}, \Sigma)$ [14]:

$$\boldsymbol{f}(\boldsymbol{x}) = \frac{1}{(2\pi)^{n/2}|\Sigma|^{1/2}} e^{-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu})^T\Sigma^{-1}(\boldsymbol{x}-\boldsymbol{\mu})} \tag{4.2}$$

where $\boldsymbol{\mu}$ is the mean vector and Σ is the *covariance matrix*. The covariance matrix describes the relationship between the different output variables. So, the standard deviation in the one-dimensional case describes the relationship x has on itself whereas in the d-dimensional case Σ describes the relationship each element in $\boldsymbol{x}$ has on each other element in $\boldsymbol{x}$, including itself. The d-dimensional case also extends the idea that $\sigma > 0$ by requiring the covariance matrix Σ to be symmetric ($\Sigma^T = \Sigma$) and positive semi-definite ($\boldsymbol{x}^T\Sigma\boldsymbol{x} \geq 0$ for all $\boldsymbol{x} \in \mathbb{R}^d$) [1]. If we want to generate samples from a MVN, the Cholesky Decomposition $\Sigma = LL^T$ is useful. If $\mathbf{x} \sim \mathscr{N}(0, \Sigma)$ then we can perform an affine transformation $\boldsymbol{\mu} + L\mathbf{x} \sim \mathscr{N}(\boldsymbol{\mu}, L\Sigma L^T)$, and sampling from the

[1] When Σ has 0 as an eigenvalue, Σ^{-1} doesn't exist and has to be treated differently. So for the rest of the thesis we consider only when Σ is positive definite.

original distribution $\mathbf{x}$ can result in sampling from any other MVN.

4.3. Gaussian Processes.

"A Gaussian Process *is a collection of random variables, any finite number of which have a joint Gaussian Distribution.*" [3] A Gaussian Process is completely described by a mean function $\mu(\mathbf{x})$ and a kernel $K(\mathbf{x}, \mathbf{x}')$ [3]. The kernel is used to generate a covariance matrix. Given some training data inputs $X = \{\mathbf{x}_1, ..., \mathbf{x}_n\}$ and outputs $Y = \{y_1, ..., y_n\}$ (where $y_i = g(\mathbf{x}_i)$, $g(\mathbf{x})$ is the latent function), to recreate Figure 4.1 we need to create a *prior* joint distribution by using the training data X and Y as well as the locations we want to plot the mean and confidence interval at $X_* = \{\mathbf{x}_1^*, ..., \mathbf{x}_m^*\}$. This is can be obtained by a MVN, where $\boldsymbol{f}$ and $\boldsymbol{f^*}$ are the random variables from the GP corresponding to the *range* at the training data X and testing data X_* respectively:

$$\begin{bmatrix} \boldsymbol{f} \\ \boldsymbol{f^*} \end{bmatrix} \sim \mathscr{N}\left(\begin{bmatrix} \boldsymbol{\mu}(X) \\ \boldsymbol{\mu}(X_*) \end{bmatrix}, \begin{bmatrix} K(X,X) & K(X,X_*) \\ K(X,X_*)^T & K(X_*,X_*) \end{bmatrix} \right) \tag{4.3}$$

Then, conditioning on the training data gives the *posterior* joint distribution:

$$\begin{aligned} (\boldsymbol{f^*}|X_*, X, \boldsymbol{f}) \sim & \mathscr{N}(\boldsymbol{\mu}(X_*) + K(X,X)^T K(X,X)^{-1}(\boldsymbol{f} - \boldsymbol{\mu}(X)), \\ & K(X_*,X_*) - K(X,X_*)^T K(X,X)^{-1} K(X,X_*)) \end{aligned} \tag{4.4}$$

where

$$K(X, X_*) = \begin{bmatrix} K(\mathbf{x}_1, \mathbf{x}_1^*) & K(\mathbf{x}_1, \mathbf{x}_2^*) & \dots & K(\mathbf{x}_1, \mathbf{x}_m^*) \\ K(\mathbf{x}_2, \mathbf{x}_1^*) & K(\mathbf{x}_2, \mathbf{x}_2^*) & \dots & K(\mathbf{x}_2, \mathbf{x}_m^*) \\ \vdots & \vdots & \ddots & \vdots \\ K(\mathbf{x}_n, \mathbf{x}_1^*) & K(\mathbf{x}_n, \mathbf{x}_2^*) & \dots & K(\mathbf{x}_n, \mathbf{x}_m^*) \end{bmatrix}$$

So, the posterior joint distribution is a MVN that provides the standard deviation and covariance matrix, describing the behavior of the test points X_*. The mean $\boldsymbol{\mu}$ of a GP is typically just the zero-vector. So, the last thing that we need to identify is what the kernel $K(\mathbf{x}, \mathbf{x}')$ is.

There are two kernels typically used for GPs. The first is the Squared Exponential/ Radial Basis Function (RBF), which admits the following formula in one-dimension:

$$K(x,x') = e^{\left(-\frac{(x-x')^2}{2\ell^2}\right)}. \tag{4.5}$$

The other is the Matern 5/2 Kernel, which admits the following formula in one-dimension:

$$K_{5/2}(x,x') = \left(1 + \frac{\sqrt{5}(x-x')^2}{2\ell^2} + \frac{5(x-x')^4}{12\ell^4}\right) e^{\left(-\frac{\sqrt{5}(x-x')^2}{2\ell^2}\right)}. \tag{4.6}$$

The Squared Exponential is the most common while the Matern 5/2 Kernel is traditionally used in machine learning hyperparameter optimization due to being less smooth than the squared exponential [15]. Having a less smooth kernel allows for more complicated functions to be modeled by the GP, which is beneficial in machine learning problems where it can be beneficial to have less assumptions about the behavior of the latent function. The value ℓ represents the lengthscale of the kernel and is itself a hyperparameter, and is learnable with Maximum Likelihood Estimation (MLE) [7] [3].

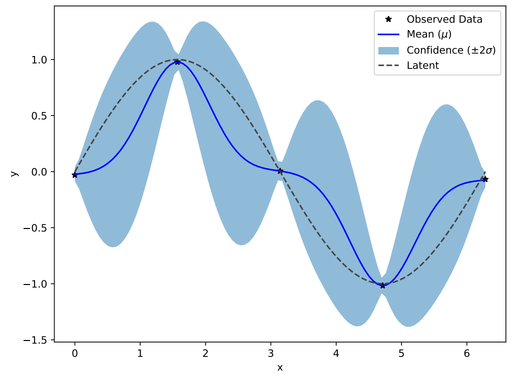
Accounting for Gaussian noise in the function evaluations: $\boldsymbol{y} = \boldsymbol{f}(X) + \boldsymbol{\epsilon}$ where $\boldsymbol{\epsilon} \sim \mathcal{N}(\boldsymbol{\mu}, \Sigma)$ is simple by adding to the diagonal of the first block of the covariance matrix:

$$\begin{bmatrix} \boldsymbol{y} \\ \boldsymbol{y^*} \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} \boldsymbol{\mu}(X) \\ \boldsymbol{\mu}(X_*) \end{bmatrix}, \begin{bmatrix} K(X,X) + \sigma^2 I & K(X,X_*) \\ K(X,X_*)^T & K(X_*,X_*) \end{bmatrix}\right) \tag{4.7}$$

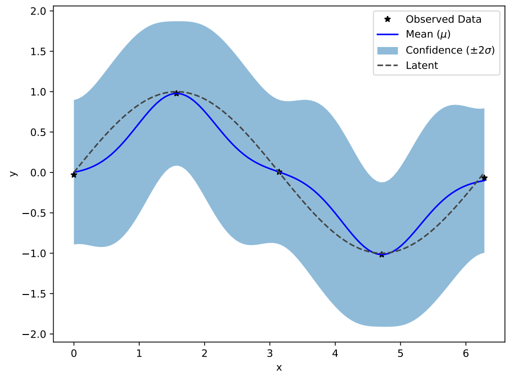
The posterior distribution is almost identical to equation (4.4) we saw previously:

$$\begin{aligned}(\boldsymbol{y^*}|X_*, X, \boldsymbol{y}) \sim & \mathcal{N}(\boldsymbol{\mu}(X_*) + K(X,X)^T(K(X,X)+\sigma^2 I)^{-1}(\boldsymbol{y} - \boldsymbol{\mu}(X)), \\ & K(X_*,X_*) - K(X,X_*)^T(K(X,X)+\sigma^2 I)^{-1}K(X,X_*))\end{aligned} \tag{4.8}$$

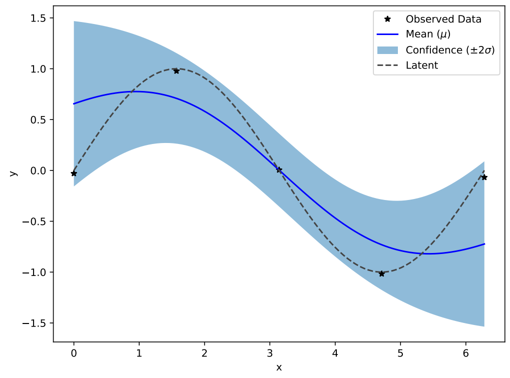
So what is σ? This is another hyperparameter! Choosing the hyperparameters of our GP, ℓ (the length-scale) and σ (signal noise), manually would defeat the purpose of using Bayesian Optimization for AutoML. Fortunately, there is a method called Maximum Likelihood Estimation which determines the best values for ℓ and σ automatically [7] [3]. The likelihood is defined as $p(\mathbf{y}|\mathbf{x}, l, \sigma)$. The idea is that the best values for ℓ and σ maximize the likelihood, hence the name Maximum Likelihood Estimation. Treating the likelihood as a loss function, it can be optimized with respect to ℓ and σ using machine learning techniques like gradient descent. Figure 4.2 shows the effect that varying the signal noise and lengthscale parameters has on the



(a) Gaussian Process with small signal noise and small lengthscale.



(b) Gaussian Process with large signal noise and small lengthscale.



(c) Gaussian Process with a moderate signal noise and large lengthscale.

Fig. 4.2: Multiple Gaussian Processes with varying signal noise and lengthscales obtained from the same training data. The training data is noisy function evaluations of $\sin(x)$ on the domain $[0, 2\pi]$.

resulting GP.

From Figure 4.2 we can easily tell that the signal noise increases the confidence bound at each point in the Gaussian Process. The lengthscale is a bit harder to understand, but essentially what it is doing is modifying the effect the training data has on their surrounding intervals. A larger lengthscale means the training data more strongly influences the surrounding values. If the lengthscale is large enough, then the posterior mean becomes more or less the mean of the training data. If the lengthscale is small enough, the mean of the posterior is simply the value of the training data at the training points and is zero elsewhere (or whatever else the prior mean is). Thus, we can observe that there are choices for σ and ℓ that better describe the training data than others.

The next bit of theory that needs to be developed is what needs to be changed in a Gaussian Process so that it incorporates derivative information. As mentioned earlier, incorporating gradient information allows for Bayesian Optimization to use more data than in the derivative-less case. We call the former FOBO for First Order Bayesian Optimization and the latter ZOBO for Zeroth Order Bayesian Optimization. One approach to FOBO is the following model, where the $\mu(\mathbf{x})$ function and kernel function $K(\mathbf{x}, \mathbf{x}')$ are adjusted [20]:

$$\begin{aligned} \tilde{\boldsymbol{\mu}}(\mathbf{x}) &= (\mu(\mathbf{x}), \nabla\mu(\mathbf{x}))^T \\ \tilde{K}(\mathbf{x}, \mathbf{x}') &= \left(\begin{bmatrix} K(\mathbf{x}, \mathbf{x}') & J(\mathbf{x}, \mathbf{x}') \\ J(\mathbf{x}', \mathbf{x})^T & H(\mathbf{x}, \mathbf{x}') \end{bmatrix} \right) \end{aligned} \tag{4.9}$$

where $J(\mathbf{x}, \mathbf{x}')$ is the gradient of $K(\mathbf{x}, \mathbf{x}')$ with respect to $\mathbf{x}'$ and $H(\mathbf{x}, \mathbf{x}')$ is the Hessian of $K(\mathbf{x}, \mathbf{x}')$. Then, conditioning on the prior distribution gives us a posterior similar to equation (4.8), assuming we are dealing with noisy observations:

$$\begin{aligned} (\mathbf{y}^*|X_*, X, \mathbf{y}) \sim & \mathcal{N}(\tilde{\boldsymbol{\mu}}(\mathbf{x}) + \tilde{K}(\mathbf{x}, X)(\tilde{K}(X, X) + I_n \otimes diag(\sigma_1^2, \ldots, \sigma_{d+1}^2))^{-1}((y, \nabla y)^{1:n} - \tilde{\boldsymbol{\mu}}(\mathbf{x})), \\ & \tilde{K}(\mathbf{x}, \mathbf{x}') - \tilde{K}(\mathbf{x}, X)(\tilde{K}(X, X) + I_n \otimes diag(\sigma_1^2, \ldots, \sigma_{d+1}^2))^{-1}\tilde{K}(X, \mathbf{x}')) \end{aligned} \tag{4.10}$$

where σ_i^2 $(1 \leq i \leq d+1)$ is the signal noise parameter for each of the function evaluation and gradients for each observation $\mathbf{y} = \boldsymbol{f}(X) + \boldsymbol{\sigma}$ for a d-dimensional problem. Once again, we can play around with the noise and lengthscale to obtain an intuition of what those parameters represent:

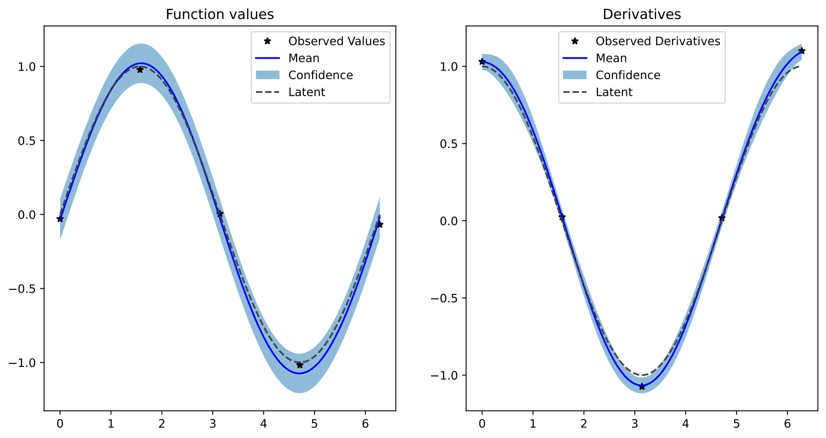


Fig. 4.3: A derivative-enabled Gaussian Process with small signal noise and a small lengthscale. The training data is noisy function evaluations of $\sin(x)$ on the domain $[0, 2\pi]$.

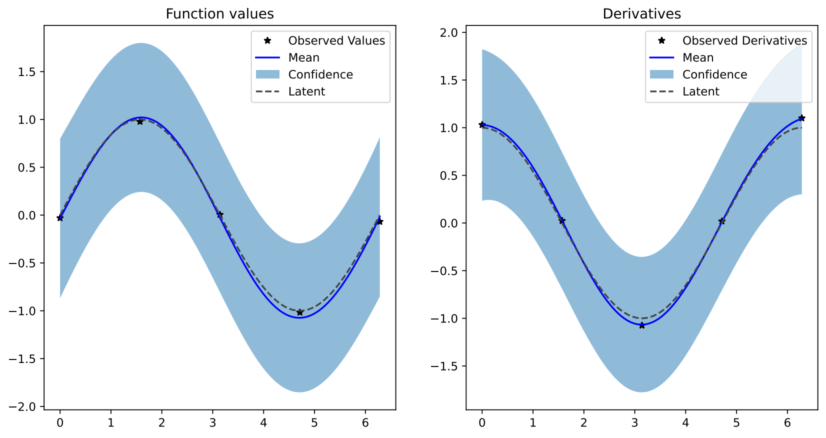


Fig. 4.4: A derivative-enabled Gaussian Process with large signal noise and small lengthscale. The training data is noisy function evaluations of $\sin(x)$ on the domain $[0, 2\pi]$.

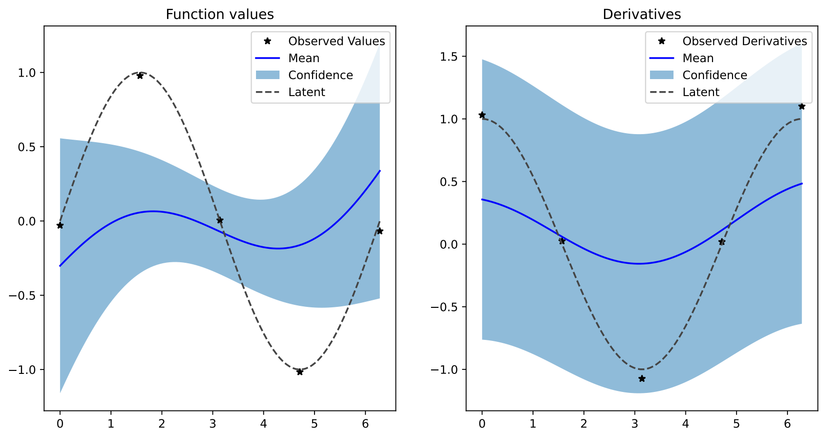


Fig. 4.5: A derivative-enabled Gaussian Process with a moderate signal noise and large lengthscale. The training data is noisy function evaluations of $\sin(x)$ on the domain $[0, 2\pi]$.

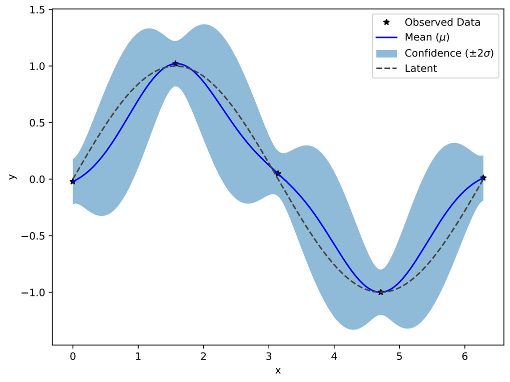
From Figure 4.3 we immediately see that incorporating gradient information allows the GP to have a better estimate of the latent function. This is observed by the GP having a smaller variance around the training data.

From Figure 4.3, Figure 4.4, and Figure 4.5 we see that the signal noise increases the confidence bound at each point in the Gaussian Process. The lengthscale also has a similar effect as seen for standard Gaussian Processes, where a larger value causes the training data to have a stronger effect on the mean of the GP around them.

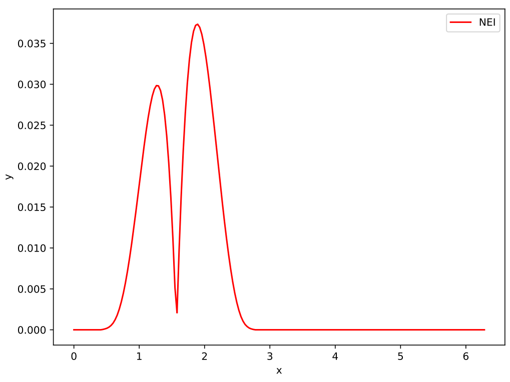
Recall that Bayesian Optimization has two steps that are in a loop: fitting the Gaussian Process (with MLE and updating the posterior distribution) and then obtaining a candidate point (which will be used to evaluate the latent function) through an acquisition function. An acquisition function takes a GP and a point in the domain as input and returns a value describing how useful evaluating that point would be. The maximum of the acquisition function is the candidate point.

4.4. Acquisition Functions.

Acquisition functions are easy to understand visually. Assume we have a Gaussian Process like the one shown in Figure 4.6 (a). The *Noisy Expected Improvement* (NEI) acquisition function will assign a usefulness value to each point in the domain, like in Figure 4.6 (b).



(a) A Gaussian Process fitted on noisy data.



(b) The values from Noisy Expected Improvement on a Gaussian Process fitted on noisy data.

Fig. 4.6: A Gaussian Process (left) fitted on noisy data from $sin(x)$ on the interval $[0, 2\pi]$ with the values of Noisy Expected Improvement over the domain (right).

Thus, Figure 4.6 would result in a candidate point around $x = 2$. Naturally, there are all sorts of acquisition functions that can be used. They vary from simple to complex and all try to balance the trade-off between exploring areas of high variance and exploiting areas of high mean. The two primary acquisition functions presented in this thesis are the Noisy Expected Improvement and Knowledge Gradient acquisition functions. Noisy Expected Improvement is a modification of the most common acquisition function Expected Improvement (EI) [6]:

$$\mathrm{EI}_n(\mathbf{x}') := \mathbb{E}_n\left(\max(f(\mathbf{x}') - f_n^*, 0) | X, Y\right) \tag{4.11}$$

where (X, Y) are the training data and f_n^* is the current best point (that is, the maximum value in Y). This formula states that the *usefulness* of a point is the probability that it results in an evaluation larger than the current best sampled point. Mathematically speaking, $\mathrm{EI}_n(\mathbf{x}')$ is the integral of the posterior at $\mathbf{x}'$ from f_n^* to ∞. Thus, both the confidence region and the mean can contribute to a higher usefulness score at a point $\mathbf{x}'$. However, this acquisition function does not really work with noisy function evaluations, since f_n^* is no longer expected to be a constant. This is where the modification to create Noisy Expected Improvement comes in [1]:

$$\mathrm{NEI}_n(\mathbf{x}') := \mathbb{E}_n\left(\max(y' - \max(Y_{\text{base}}), 0) | X, Y\right) \tag{4.12}$$

The expectation is over the variables y' and Y_{base}, where (X, Y) are the training data, and $y_{\text{base},i} \sim \mathcal{N}(\mu(\mathbf{x}_i|X,Y), \sigma^2(\mathbf{x}_i|X,Y))$ are the *random variables* at the points $\mathbf{x}_i$ from the fitted Gaussian Process. This is illustrated in Figure 4.6 (a) at the observed data points, which have a mean and non-zero confidence $\pm 2\sigma$. Similarly, $y' \sim \mathcal{N}(\mu(\mathbf{x}'|X,Y), \sigma^2(\mathbf{x}'|X,Y))$ is the random variable from some point $\mathbf{x}'$ in the domain. This modification takes into account the variability of the training data. So, NEI assigns a point $\mathbf{x}'$ its usefulness based on the probability that the resulting y-value from sampling at $\mathbf{x}'$ is, on average, higher than the average best y-value from the sampled points X.

The final acquisition function that will be presented is Knowledge Gradient (KG). The insight into this acquisition function is that NEI and EI assume that at the end of the BayesOpt procedure you will only return a point that has been sampled previously. In contrast, KG removes this assumption and assumes that the result of the BayesOpt procedure can be any point in the domain, whether or not it has been sampled. The formula for KG is [6]:

$$KG_n(\mathbf{x}') := \mathbb{E}_n\left(\mu^*_{n+1} - \mu^*_n | \mathbf{x}' = y', X, Y\right)$$

where (X, Y) are the training data, μ^*_n is the maximum mean of the posterior of the GP, and μ^*_{n+1} is the maximum mean of the posterior of the GP *if* $\mathbf{x}' = \mathbf{y}'$ *is sampled*. So, KG tries to find the point $\mathbf{x}'$ that will result in the largest increase of the posterior mean when sampled. Since sampling takes into account both the mean and variance at a point $\mathbf{x}'$, KG also balances the trade-off between exploration and exploitation.

All of the above acquisition functions work almost the same with gradient-based Gaussian Processes as they do with standard Gaussian Processes. The only difference is that the output of the acquisition functions are $d+1$-dimensional vectors instead of scalars, so the function value that needs to be maximized is the one corresponding to function evaluations, not the gradient, which is often the first element in the posterior mean of the GP.

5. Software Development.

5.1. Selecting Software.

So, with the background and theory established, the next step is to find a software package implementing Bayesian Optimization. Initially we used the `fmfn/BayesianOptimization` Python library [12]. Trying out the library on the earlier example from section 3 led to strange behavior as seen in Figure 5.1:

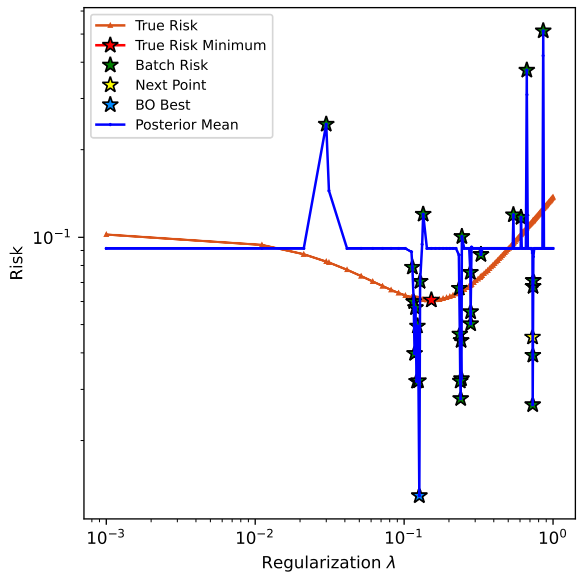


Fig. 5.1: The Python library `fmfn/BayesianOptimization` used to find the minimum of the true loss from the example scenario developed in section 3.

Trying out different things, like scaling the training data to be logarithmic (as seen in Figure 5.1) did not seem to result in consistent BayesOpt, since different seeds drastically changed the results. Furthermore, the package does not support derivative-based GPs (dGPs).

The next step was to figure out a software package that supports dGPs, is well-maintained, actively being developed, and is being used by the research community. There were many options to choose from:

- `luinardi/hypermapper` [11]
  - Well maintained by a small group
  - Does not have dGP support
  - We would be supported in its development by direct contacts
- `google/vizier` [16] [9]
  - Owned by Google
  - Primarily for industry-level usage
  - Does not have dGP support
- `wujian16/Cornell-MOE` [20] [19]
  - The Python library that is linked in the original dGP paper
  - No longer maintained
  - Does not work with current software packages, required versions unknown

    - Does have dGP support

- `facebook/Ax` [17]
    - Owned by Meta Open Source
    - Actively maintained
    - Primarily for industry-level usage
    - Uses BoTorch under the hood
- `pytorch/botorch` [1] [18]
    - Owned by Meta Open Source
    - Actively maintained
    - Has partial dGP support
    - Used heavily by the research community

So, the choice is to either make a no-longer maintained package work (`wujian16/Cornell-MOE`) that supports derivative-based GPs, or to choose an existing maintained and working package and to implement dGP support. We chose the latter option. Initially, BoTorch was chosen since it made more sense to contribute to the open source community and provide support to a widely used library. For a little while, we switched to Ax since the tutorials mention to use Ax instead, which is equivalently open-source and maintained. Ax was very easy to use, however we quickly realized that to implement dGP support it would be better to just use BoTorch for the reasons listed below.

BoTorch uses PyTorch, which is a machine learning library heavily used in academia, and GPyTorch, which is a library that implements GPs also based on PyTorch [8]. Fortunately, GPyTorch *does* support dGPs, and can be used within BoTorch [4]! The figures that have been included in this thesis were from BoTorch and the dGPs are from GPyTorch. Figure 5.2 shows the results of fitting a 2D dGP to noisy data.

EI worked immediately with dGPs. However, using NEI with dGPs resulted in the following error message:

```
torch._C._LinAlgError: linalg.cholesky: (Batch element 0): The factorization
could not be completed because the input is not positive-definite (the leading
minor of order 6 is not positive-definite).
```

This error is a result of the lengthscale of the dGP kernel sometimes being *very* small, resulting in the matrix having negative eigenvalues, meaning it is no longer positive definite:

```
Lengthscale: tensor([[2.6458e-129]], grad_fn=<SoftplusBackward0>)
Noise: 351585.73859480093
```

A work around to this is to add to the diagonal of the covariance matrix until it is positive definite. This is done by converting the covariance matrix from a PyTorch tensor to a tensor from the linear algebra

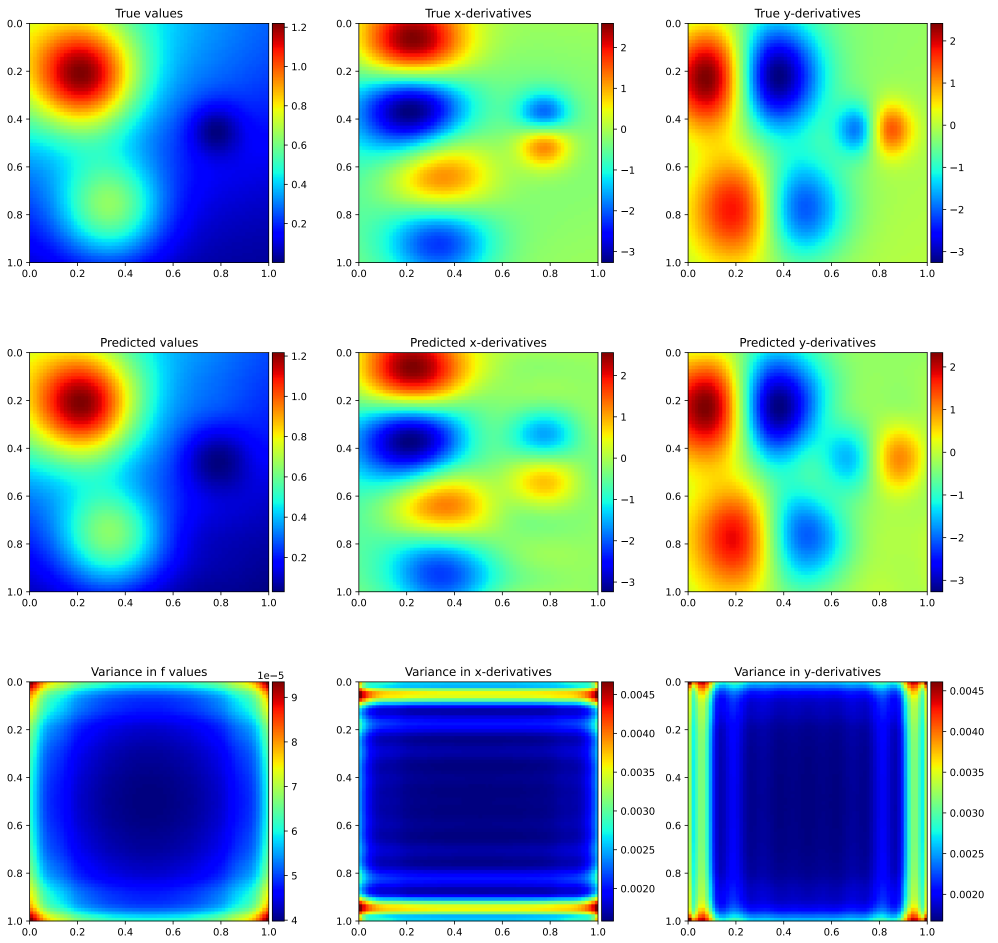


Fig. 5.2: A 2D dGP fitted to noisy data from Frankes function on the domain $[0, 1] \times [0, 1]$.

package LinearOperator, which already implemented a function for adding to the diagonal of a matrix if it is not positive definite [5]. However, the lengthscale issue came back in another error:

```
RuntimeError:  3 elements of the 3 element gradient array ‘gradf‘ are NaN.
This often indicates numerical issues.
```

This time, the solution is provided by another contributor to BoTorch, James T. Wilson, who suggested standardizing the output training data that is used to fit the GP [21]. Standardization is typically only done for the range, but considering that we are dealing with gradients it can make sense to standardize the domain so that the gradients are reasonable values. The domain is standardized by first finding the mean $\boldsymbol{\mu} = (\mu_1, \mu_2, \ldots, \mu_d)$ and standard deviation $\boldsymbol{\sigma} = (\sigma_1, \sigma_2, \ldots, \sigma_d)$ for each dimension of the training data. Then, each point is standardized, for example the point $\mathbf{x}_i = (x_{(i,1)}, x_{(i,2)}, \ldots, x_{(i,d)})$ becomes $\hat{\mathbf{x}}_i = (\frac{x_{(i,1)}-\mu_1}{\sigma_1}, \frac{x_{(i,2)}-\mu_2}{\sigma_2}, \ldots, \frac{x_{(i,d)}-\mu_d}{\sigma_d})$. A similar standardization procedure can be done for the range.

Standardizing both the domain and range of the training data resulted in lengthscale values that were no longer incredibly small, resulting in NEI working for dGPs. We suspect the incredibly small lengthscale is a result of using the L-BFGS optimizer in the MLE method used to fit the dGPs in BoTorch. Using the Adam optimizer in MLE did not result in exaggerated lengthscales, however, it converged much slower. A work-around to the small-lengthscale problem was to put a constraint on the lengthscale, bounding it to be between two values.

The next step is to get KG to work with dGPs. Once again, using it with dGPs did not work right away. This time, we got the following error:

```
RuntimeError:  Cannot yet add fantasy observations to multitask GPs,
but this is coming soon!
```

Implementing KG to work with dGPs was simple due to an existing Pull Request from four years ago. KG now works with dGPS and a new pull request has been submitted to merge into the main BoTorch repository [22]. The PR is still open at the time of writing this thesis.

5.2. Evaluating FOBO. Now that we had a BayesOpt implementation that used dGPs with NEI, it was time to collect some results. In BayesOpt literature, there are a variety of different types of tests used to compare performance between optimization methods. we decided start with the Rosenbrock function since it allows for changing the dimensionality d of the problem, has a known global minimum of 0 at $(1, ..., 1)$, is positive elsewhere, is easy to implement, and is typically used in BayesOpt literature [20]:

$$f(\boldsymbol{x}) = \sum_{i=1}^{d-1} \left[100(x_{i+1} - x_i^2) + (1 - x_i)^2\right], \quad \boldsymbol{x} \in \mathbb{R}^d \tag{5.1}$$

We then created 12 different configurations of BO to see which method would perform the best. The criteria for best configuration was based on the lowest value of the Rosenbrock function after 50 iterations in the domain $[-2, 2]^3$. We modified whether or not the function evaluations and/or the gradient evaluations had gaussian noise. We also modified whether or not the domain and range were standardized. Lastly, we modified whether or not the lengthscale is bounded. The configurations are described in Table 5.1.

To generate the results, we ran 50 iterations of BayesOpt for 154 different pseudo-random number generator seeds. Each seed provided each configuration above with the same five initialization points which were obtained from randomly sampling the domain $[-2, 2]^3$. Note that this means we tested BayesOpt in three dimensions, which is equivalent to optimizing three hyperparameters at the same time. The evaluations of the Rosenbrock function serve as our “loss function”.

Name	$\boldsymbol{f}$ noise	$\boldsymbol{f'}$ noise	standardize x	standardize y	bound lengthscale
zobo1	✓	—	×	×	×
zobo2	×	—	×	×	×
zobo3	✓	—	✓	✓	×
zobo4	×	—	✓	✓	✓
zobo5	✓	—	×	×	✓
zobo6	×	—	×	×	✓
fobo1	✓	✓	×	×	×
fobo2	✓	×	×	×	×
fobo3	×	×	×	×	×
fobo4	✓	✓	✓	✓	×
fobo5	✓	×	✓	✓	×
fobo6	×	×	✓	✓	×

Table 5.1: ZOBO and FOBO configurations for trial 1.

The different methods can be compared visually through a *performance profile* [10]. A performance profile uses the following inequality as a convergence test, which is either satisfied or not:

$$f(x_0) - f(x) \geq (1-\tau)(f(x_0) - f_L) \tag{5.2}$$

where for a specific problem p, f_L is the best value obtained by all solvers s within a certain number of function evaluations. In my scenario, each seed is its own problem p, each configuration is its own solver s, and the number of function evaluations is 50. $\tau \in (0, \infty)$ is the tolerance, which is a parameter we choose when making the performance profiles. Larger values for τ allows for solvers s to be further from the value f_L and still pass the convergence test. $f(x_0)$ is the starting point for the problem (the initial minimum value obtained by each solver, which is the same).

Another value, $t_{p,s}$ can be modified depending on the context, but in my situation it is the number of Bayesian Optimization loops/ function evaluations until the convergence test is passed for a seed p and configuration s. This then leads to the performance ratio $r_{p,s}$, where S is the set of all solvers we are comparing:

$$r_{p,s} = \frac{t_{p,s}}{\min\{t_{p,s}, s \in S\}} \tag{5.3}$$

Then, the performance profile for a configuration s is defined as:

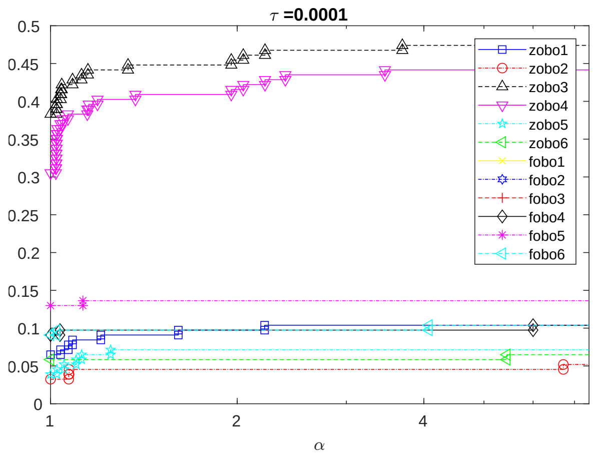
$$\rho_s(\alpha) = \frac{1}{|P|}\text{size}\{p \in P : r_{p,s} \leq \alpha\} \tag{5.4}$$

where $\alpha \in [1, \infty)$ is the domain of the plot for the performance profile. What $\rho_s(\alpha)$ represents is the proportion of problems solved (given some τ) by a specific configuration s within a certain number of Bayesian Optimization loops. The number of loops is specified by α. If $\alpha = 3$ then the number of iterations allowed for each configuration s is at most 3 times as many iterations as the smallest number of iterations used by any configuration to satisfy the convergence test. If $\alpha = 1$ then $\rho_s(\alpha)$ is simply the proportion of the time configuration s achieved the minimum value of all other configurations. The limit as $\alpha \to \infty$ means that each configuration can have as many iterations as needed to solve the problem, meaning $\rho_s(\alpha) \to 1$. Let us take a look at the performance profiles of our configurations for three values of τ in Figure 5.3.

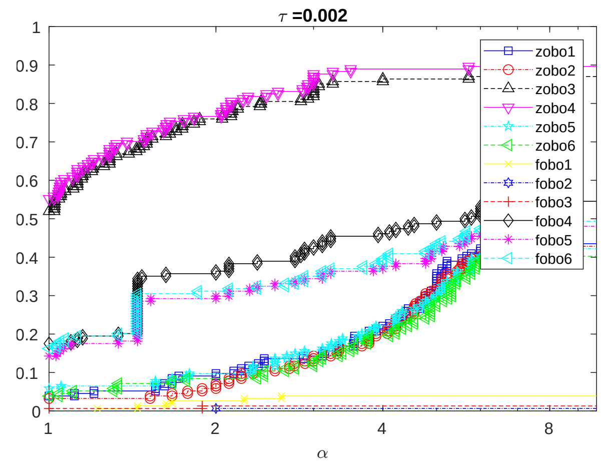
From Figure 5.3 (a) and Figure 5.3 (b) we see that the best BO configurations were zobo3 followed by zobo4 respectively for high-accuracy problems (problems where τ is relatively small). This means that across all seeds, zobo3 and zobo4 were performing better, on average, at finding the smallest value of the 3D Rosenbrock function when compared with the other configurations. From Figure 5.3 (c), where the tolerance was higher, we see fobo4, fobo5, and fobo6 perform the best for small α. Given the larger tolerance ($\tau = 0.009$), this means that those three fobo methods were approaching the minimum of the Rosenbrock function very quickly. However, we also see that if we allow the other methods to have many more function evaluations, zobo3 and zobo4 catch up with fobo4, fobo5, and fobo6. Given the high tolerance, however, we are not able to conclude that the fobo methods were obtaining smaller minima on the 3D Rosenbrock function than the zobo methods. In fact, from Figure 5.3 (a) (which has a small tolerance), we see that the zobo methods were finding better minima. So, we can conclude that the fobo methods were better at finding smaller values quickly up to a certain minimum and that zobo methods are better at finding even smaller values given enough iterations. This behavior can be seen in Figure 5.4:

These results come as a surprise since we would expect Bayesian Optimization with derivative information to perform better across the board. FOBO is more computationally expensive, yet uses more information than ZOBO methods.

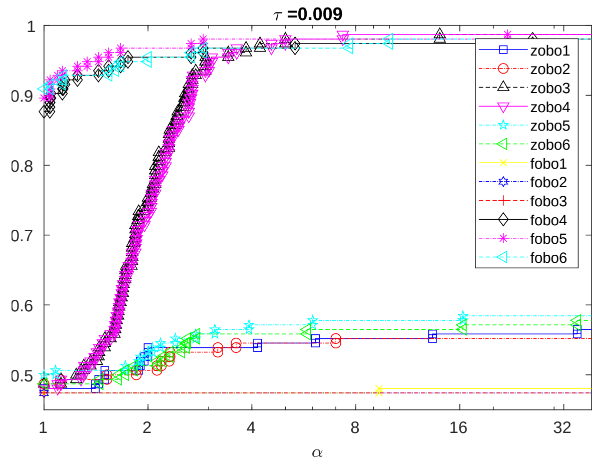
We also tried a naive method of adding first-order information to ZOBO to see if perhaps another FOBO method would perform better. This can be done by using derivative information to add “imaginary” sampled points by using linear approximations. For example, in the 1D case, sampling at a point x would result in the function evaluation $f(x)$ and its gradient $f'(x)$. Thus, the sampled point would be $(x, f(x))$ Two linear approximations could be added: $(x + h, f(x) + h \cdot f'(x))$ and $(x - h, f(x) - h \cdot f'(x))$. For the 3D case



(a)



(b)

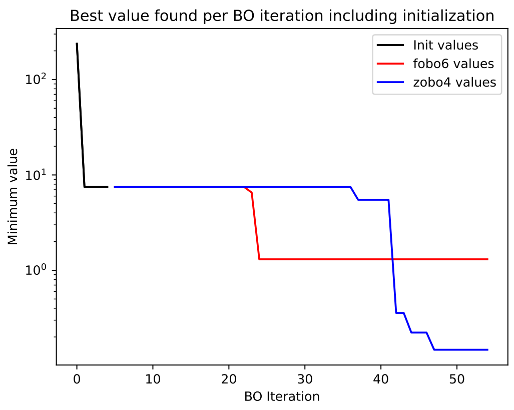


(c)

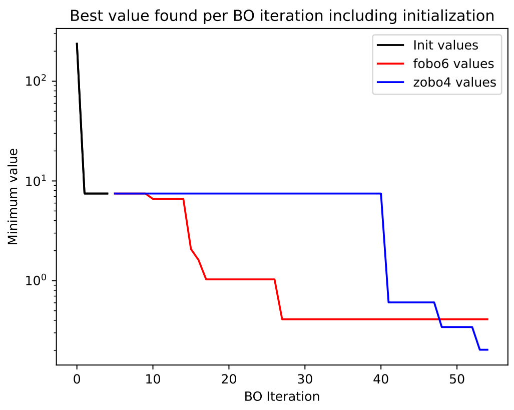
Fig. 5.3: Performance profiles for 154 seeds of BayesOpt on 12 different configurations from Table 5.1 with varying values of τ. The best possible configuration would achieve a performance profile of 1 for $\alpha = 0$.

this would result in 6 “imaginary” points (2 for each dimension). We used the value of $h = 0.025$. Thus, zobo7 and zobo8 from Table 5.2 were created using this idea. We then ran 98 different seeds on the 5 best performing configurations from the first trial, listed in Table 5.2, and created performance profiles for those runs, as seen in Figure 5.5.

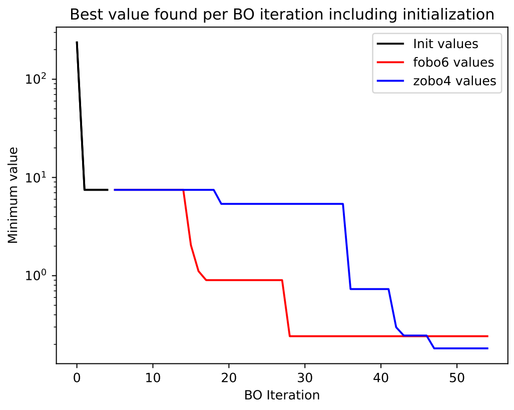
The performance profiles from Figure 5.5 show that for high-accuracy regimes, where τ is small, zobo7 and zobo8 are performing really well. Thus, incorporating derivative information is certainly useful for obtaining better results in optimizing a latent function than zeroth order methods. For low-accuracy regimes, where τ is large, fobo4, fobo5, and fobo6 are once again doing well for low α, showing that they are better at lowering their best function value early, but once again zobo methods catch up, with zobo7 and zobo8 surpassing all other methods relatively quickly. This shows that the implementation of FOBO from [20] may



(a) Seed 10



(b) Seed 14



(c) Seed 7

Fig. 5.4: Plot of the best-obtained-value for zobo4 and fobo6 per iteration of Bayesian Optimization.

Name	f noise	f' noise	standardize x	standardize y	bound lengthscale	"imaginary" points
zobo3	✓	—	✓	✓	×	×
zobo4	×	—	✓	✓	✓	×
zobo7	✓	—	✓	✓	×	✓
zobo8	×	—	✓	✓	✓	✓
fobo4	✓	✓	✓	✓	×	—
fobo5	✓	×	✓	✓	×	—
fobo6	×	×	✓	✓	×	—

Table 5.2: ZOBO and FOBO configurations for trial 2.

not be the best general-purpose method of implementing gradient information in Bayesian Optimization. However, the introduction of an additional hyperparameter h motivates the exploration of a better general-purpose FOBO method.

6. Conclusion.

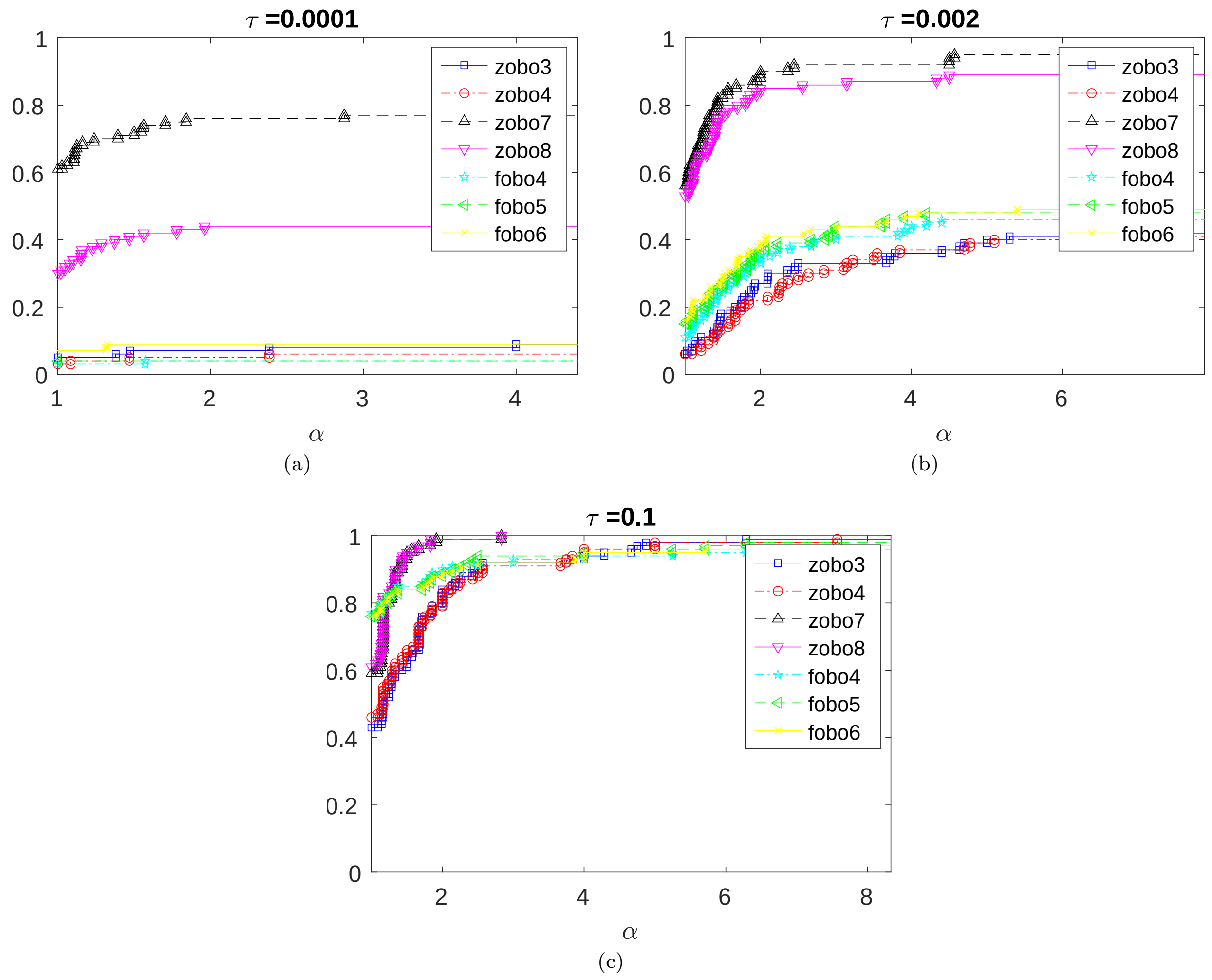


Fig. 5.5: Performance profiles for 98 seeds of BayesOpt on 7 different configurations from Table 5.2 with varying values of τ.

Bayesian Optimization is a method to do automatic hyperparameter tuning. The theory behind Bayesian Optimization is robust and well-studied. It has been used extensively in a variety of domains to solve a multitude of different problems. As a result, there are many software implementations of the theory each with their own benefits and problems. For this thesis, we covered the theory before attempting to tackle the motivating example. To our surprise, no maintained BayesOpt software was available that incorporated gradient information. This necessitated the contribution to the existing BayesOpt library BoTorch.

Unfortunately, the results we obtained were not supportive of the conclusion that the first-order Bayesian Optimization method described in [20] is better than zeroth-order Bayesian Optimization. There are many avenues for moving forward to try to improve FOBO such that it will perform better than ZOBO. The idea that FOBO can outperform ZOBO was shown by a naive linear approximation method.

The first issue that can be improved is that there is a single signal noise parameter for GPyTorch's implementation of dGPs [4]. In regular GPs, the signal noise parameter embeds information about how much noise there is in the function evaluations. Having a single signal noise parameter for dGPs assumes that the noise for function evaluations and gradient evaluations are the same. This assumption can lead to poor fitting when function evaluations are noisy but not the gradient observations and vice-versa.

Another possibility is that this method of incorporating gradient information to Gaussian Processes is flawed. From Wu et al's 2017 paper, "Bayesian Optimization with Gradients", using derivative information was not always beneficial [20]. Other methods for incorporating gradients into BayesOpt have been researched, and other unexplored implementations are bound to exist [13]. In this thesis we explored using linear approximations, but this method is not robust since another hyperparameter of h is introduced. Further, the use of linear approximations taints the training data by having a mix of true noisy function evaluations with the linear approximations.

Finally, exploring the effect of gradient information with the KG acquisition function and/or using different functions to evaluate performance may shed more light on what effect FOBO has when compared with ZOBO [20]. The experiment we performed was not comprehensive and there may be regimes where FOBO performs better than ZOBO. We did try to run an experiment with KG, but it took several hours to go through one seed and requires further investigation before the software is at a state where enough meaningful results can be generated.

After FOBO is developed to a point where it consistently performs better than ZOBO, it can be applied to the motivating example of this thesis and then to real machine learning problems where gradient information is available for hyperparameters [15].

Finally, Bayesian Optimization is just one option for automatic hyperparameter tuning. It requires hyperparameters itself, such as choosing an acquisition function, a kernel and mean for its Gaussian Process, and a domain over which to perform optimization. There are other methods that can be explored as well.

References.

[1] Maximilian Balandat et al. "BoTorch: A Framework for Efficient Monte-Carlo Bayesian Optimization". In: *Advances in Neural Information Processing Systems 33*. 2020. URL: http://arxiv.org/abs/1910.06403.

[2] Quentin Bertrand et al. "Implicit Differentiation for Fast Hyperparameter Selection in Non-Smooth Convex Learning". In: *J. Mach. Learn. Res.* 23.1 (Jan. 2022). ISSN: 1532-4435.

[3] Christopher K. I. Williams Carl Edward Rasmussen. *Gaussian Processes for Machine Learning*. Adaptive computation and machine learning. MIT Press, 2006.

[4] cornellius-gp. *GPyTorch*. 2017–. URL: https://github.com/cornellius-gp/gpytorch.

[5] cornellius-gp. *LinearOperator*. 2022–. URL: https://github.com/cornellius-gp/linear_operator.

[6] Peter I. Frazier. *A Tutorial on Bayesian Optimization*. 2018. arXiv: 1807.02811 `[stat.ML]`.

[7] Nando de Freitas. *Machine learning - Maximum likelihood and linear regression*. https://www.youtube.com/watch?v=voN8omBe2r4. Accessed April 20, 2023. 2013.

[8] Jacob Gardner et al. "Gpytorch: Blackbox matrix-matrix gaussian process inference with gpu acceleration". In: *Advances in neural information processing systems* 31 (2018).

[9] Daniel Golovin et al. "Google Vizier: A Service for Black-Box Optimization". In: *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Halifax, NS, Canada, August 13 - 17, 2017*. ACM, 2017, pp. 1487–1495. DOI: 10.1145/3097983.3098043. URL: https://doi.org/10.1145/3097983.3098043.

[10] Jorge J Moré and Stefan M Wild. "Benchmarking derivative-free optimization algorithms". In: *SIAM Journal on Optimization* 20.1 (2009), pp. 172–191.

[11] Luigi Nardi et al. "Hypermapper: a practical design space exploration framework". In: *2019 IEEE 27th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE. 2019, pp. 425–426.

[12] Fernando Nogueira. *Bayesian Optimization: Open source constrained global optimization tool for Python*. 2014–. URL: https://github.com/fmfn/BayesianOptimization.

[13] Santosh Penubothula, Chandramouli Kamanchi, and Shalabh Bhatnagar. "Novel first order bayesian optimization with an application to reinforcement learning". In: *Applied Intelligence* 51 (2021), 1565––1579.

[14] Alvin C. Rencher. *Methods of multivariate analysis*. 2nd ed. Wiley series in probability and mathematical statistics. J. Wiley, 2002.

[15] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. "Practical Bayesian Optimization of Machine Learning Algorithms". In: *Advances in Neural Information Processing Systems*. Ed. by F. Pereira et

al. Vol. 25. Curran Associates, Inc., 2012. URL: https://proceedings.neurips.cc/paper_files/paper/2012/file/05311655a15b75fab86956663e1819cd-Paper.pdf.

[16] Xingyou Song et al. “Open Source Vizier: Distributed Infrastructure and API for Reliable and Flexible Black-box Optimization”. In: *Automated Machine Learning Conference, Systems Track (AutoML-Conf Systems)*. 2022.

[17] Meta Open Source. *Ax*. 2019–. URL: https://github.com/facebook/Ax.

[18] Meta Open Source. *BoTorch*. 2018–. URL: https://github.com/pytorch/botorch.

[19] Jian Wu and Peter Frazier. “The parallel knowledge gradient method for batch bayesian optimization”. In: *Advances in Neural Information Processing Systems*. 2016, pp. 3126–3134.

[20] Jian Wu et al. “Bayesian Optimization with Gradients”. In: *Advances in Neural Information Processing Systems*. Ed. by I. Guyon et al. Vol. 30. Curran Associates, Inc., 2017. URL: https://proceedings.neurips.cc/paper_files/paper/2017/file/64a08e5f1e6c39faeb90108c430eb120-Paper.pdf.

[21] yyexela. *[Bug] Exaggerated Lengthscale*. https://github.com/pytorch/botorch/issues/1745. Accessed April 20, 2023. 2023.

[22] yyexela. *Enable fantasy models for multitask GPs Reborn*. https://github.com/cornellius-gp/gpytorch/pull/2317. Accessed April 20, 2023. 2023.