Improving Existing Bayesian Optimization Software by Incorporating Gradient Information

University of Colorado at Boulder

Computer Science Undergraduate Senior Thesis

Authors:

Alexey Yermakov

Stephen Becker

Eric Rozner

Affiliations:

Computer Science, Applied Mathematics

Applied Mathematics

Computer Science

May 2, 2023

1. 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.

2. 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. 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 choosing 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. I 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 hyper-parameters, 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: $c_1 * c_2 \leq \alpha$.

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 hyper-parameters 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 don't necessarily need to be involved in hyperparameter tuning anymore. One of these developments is the adoption of the technique called Bayesian Optimization.

As an example of a continuous hyperparameter, we'll consider the Least-Squares method. In our example, we'll assume we have n vectors of dimension p where each element is generated from a standard normal distribution. This will be our training input data $\mathbf{X}$. Then, we create some weight matrix $\mathbf{w}$, which in this case is just another vector of dimension p where each element is generated from a standard uniform distribution. Then, our training output data is generated by a matrix multiplication of $\mathbf{X}$ and $\mathbf{w}$, we'll call this $\mathbf{Y}$. The goal of our example is to try to learn what $\mathbf{w}$ is just from $\mathbf{X}$ and $\mathbf{Y}$.

Least-Squares can be used in this scenario quite easily. The Least-Squares formula for this problem is:

$$\arg\min_{\mathbf{w}} ||\mathbf{X}\mathbf{w} - \mathbf{Y}||_2^2 \tag{2.1}$$

Fortunately, equation (2.1) admits a closed-form formula as well:

$$\mathbf{w} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{Y} \tag{2.2}$$

Great! So this problem is easy to solve as-is, however, in the real world data observations are often *noisy*. Let's modify our original example to include noisy data by modifying $\mathbf{Y}$ to be $\mathbf{Y}_* = \mathbf{Y} + \sigma\mathbf{Z}$ where $\mathbf{Z} \sim \mathscr{N}(\mathbf{0}, \mathbf{I})$ and $\sigma \in (0, \infty)$. This adds standard gaussian noise to each component of the original training output $\mathbf{Y}$. Noise is often handled by regularization, a technique which improves the solutions found by fitting techniques like Least-Squares. Tikhonov regularization, also known as Ridge Regression, regularizes least-squares by penalizing the norm of the weights $\mathbf{w}$:

$$\arg\min_{\mathbf{w}} ||\mathbf{X}\mathbf{w} - \mathbf{Y}||_2^2 + \lambda||\mathbf{w}||_2^2, \quad \lambda \in (0, \infty) \tag{2.3}$$

Once again, equation (2.3) admits a closed-form formula as well:

$$\mathbf{w} = (\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I})^{-1}\mathbf{X}^T\mathbf{Y}, \quad \lambda \in (0, \infty) \tag{2.4}$$

In keeping with the machine learning spirit, the regularized Least-Squares equation from equation (2.3) can be seen as minimizing the loss function $\mathscr{L}(\mathbf{w}) = ||\mathbf{X}\mathbf{w} - \mathbf{Y}||_2^2 + \lambda||\mathbf{w}||_2^2$. The value λ is a continuous hyper-parameter that we can modify to influence the final weights $\mathbf{w}$ from the training process! Figure 2.1 below demonstrates the effect different values of λ have on the true loss, the validation loss, and the training loss.

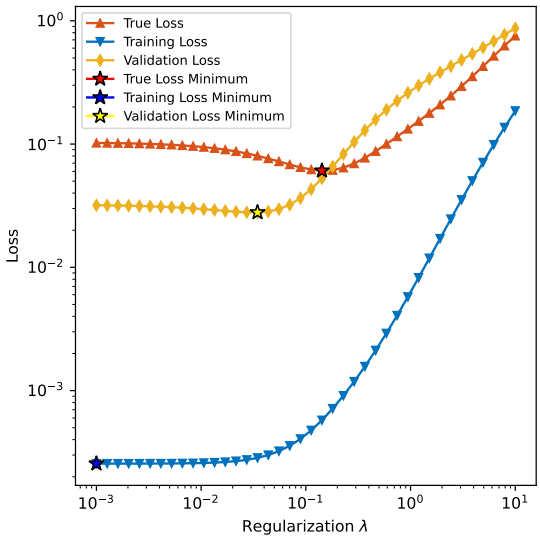


Fig. 2.1: The test problem's loss plotted for the validation data (Validation loss), the training data (Training loss), and true data (True loss). The true loss is obtained by computing the loss from a large set of input/output samples (this set is much larger than the validation set, at about 200 times more samples).

First, the training loss from Figure 2.1 decreases as λ decreases, since the regularization term from equation (2.3) adds to the final loss, which is plotted in blue. Secondly, the minimum of the true loss is for some $\lambda > 0$ since our training data is noisy and is a subset of the true loss. Typically, the true loss is not known and is estimated by testing datasets. What's 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 validation data, which might not be representative of the whole set of data that can be encountered. In imagining an image classification model, there is simply no way that a model can be trained on all possible images. Thus, additional regularization can be performed by drawing mini-batches from the validation data to try to better estimate the location of the true loss. Lastly, the loss

function $\mathscr{L}$ is differentiable with respect to λ, which provides additional information about the behavior of the loss function than simple observations. However, given that the loss curve of each mini-batch isn't necessarily the same, the gradient evaluations will also be noisy. Gradient information is useful because it provides additional information about the underlying function being optimized. Gradient information has been recognized to be very important in optimization techniques, including but not limited to gradient descent.

Bayesian Optimization can be used to find a λ near the minimum of the true loss using mini-batches of the validation data.

3. Bayesian Optimization Theory.

DEFINITION 3.1 (Bayesian Optimization). " Bayesian Optimization (BayesOpt) is an established technique for sequential optimization of costly-to-evaluate black-box functions. It can be applied to a wide variety of problems, including hyperparameter optimization for machine learning algorithms, A/B testing, as well as many scientific and engineering problems. "

At a high-level, one 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). Figure 3.1 below shows a Gaussian Process fitted to noisy observations of the function $f(x) = \sin(x)$ on the interval $[0, 2\pi]$.

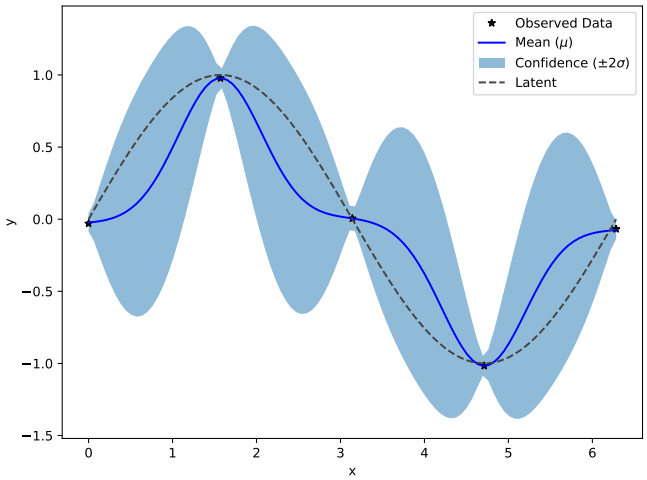


Fig. 3.1: A one-dimensional Gaussian Process (GP) fitted on noisy data. The original function and sampled data are shown, as well as the mean and two standard deviations from the mean of the GP. In this case, each x-value has a corresponding gaussian distribution across the y-values.

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. Going back to the motivating example, a 1-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 maxima.

3.1. Multivariate Normal Distributions.

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

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{3.1}$$

where μ is the mean and σ is the standard deviation. This can be generalized to an n-dimensional Gaussian $\mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$:

$$\boldsymbol{f}(\boldsymbol{x}) = \frac{1}{(2\pi)^{n/2}|\boldsymbol{\Sigma}|^{1/2}} e^{-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1}(\boldsymbol{x}-\boldsymbol{\mu})} \tag{3.2}$$

Where $\boldsymbol{\mu}$ is the mean vector and $\boldsymbol{\Sigma}$ is the *covariance matrix*. The covariance matrix describes the relationship between the different output variables. So, the standard deviation in the 1-dimensional case describes the relationship x has on itself whereas in the n-dimensional case $\boldsymbol{\Sigma}$ describes the relationship each element in $\boldsymbol{x}$ has on each other element in $\boldsymbol{x}$, including itself. The n-dimensional case also extends the idea that $\sigma > 0$ by requiring the covariance matrix $\boldsymbol{\Sigma}$ to be symmetric ($\boldsymbol{\Sigma}^T = \boldsymbol{\Sigma}$) and positive semi-definite ($\boldsymbol{x}^T\boldsymbol{\Sigma}\boldsymbol{x} \geq 0$ for all $\boldsymbol{x} \in \mathbb{R}^n$).

How does one go about generating samples from a MVN? First, we assume that we can generate samples from a standard normal distribution, that is, a equation (3.1) with $\mu = 0$ and $\sigma = 1$. Then, create an n by s matrix (called z) where each element is sampled from a standard Gaussian, effectively sampling from $\mathcal{N}(\boldsymbol{0}, \boldsymbol{1})$. n is the dimension of the MVN and s is the number of points we want to sample. We can then compute the Cholesky decomposition $\boldsymbol{\Sigma} = \boldsymbol{L}\boldsymbol{L}^{\boldsymbol{T}}$. The generated points are then described by $\mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma}) \sim \boldsymbol{\mu} + \boldsymbol{L} \times z$. Intuitively, this method is like sampling from a standard normal distribution and then moving the mean and scaling the standard deviation, $\mathcal{N}(\mu, \sigma) \sim \mu + \sigma\mathcal{N}(0, 1)$, except in n-dimensions: $\mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma}) \sim \boldsymbol{\mu} + \boldsymbol{L} \times \mathcal{N}(\boldsymbol{0}, \boldsymbol{1})$. In two-dimensions, what this would do is give us random locations of points

on a grid with a probability described by the MVN.

3.2. Gaussian Processes.

DEFINITION 3.2 (Gaussian Process). "A Gaussian Process *is a collection of random variables, any finite number of which have a joint Gaussian Distribution.*"

A Gaussian Process is completely described by a mean function $\boldsymbol{\mu}(x)$ and a kernel $K(x, x')$. The kernel is used to generate a covariance matrix. Let's assume that we have some training data $X = \{\mathbf{x}_1, ..., \mathbf{x}_n\}$ and $Y = \{\mathbf{y}_1, ..., \mathbf{y}_n\}$. To recreate Figure 3.1 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 given by:

$$\begin{bmatrix} \boldsymbol{f} \\ \boldsymbol{f^*} \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} \boldsymbol{\mu}(X) \\ \boldsymbol{\mu}(X_*) \end{bmatrix}, \begin{bmatrix} K(X,X) & K(X,X_*) \\ K(X,X_*)^T & K(X_*,X_*) \end{bmatrix} \right) \tag{3.3}$$

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

$$\begin{aligned} (\boldsymbol{f^*}|X_*, X, \boldsymbol{f}) \sim & \mathcal{N}(\boldsymbol{\mu}(X_*) + K(X,X)^T K(X,X)^{-1}(\boldsymbol{f} - \boldsymbol{\mu}(X)), \\ & K(X_*,X_*) - K(X,X_*)^T K(X,X)^{-1} K(X,X_*)) \end{aligned} \tag{3.4}$$

where

$$K(X, X_*) = \begin{bmatrix} K(x_1, x_1^*) & K(x_1, x_2^*) & \dots & K(x_1, x_m^*) \\ K(x_2, x_1^*) & K(x_2, x_2^*) & \dots & K(x_2, x_m^*) \\ \vdots & \vdots & \ddots & \vdots \\ K(x_n, x_1^*) & K(x_n, x_2^*) & \dots & K(x_n, x_m^*) \end{bmatrix}$$

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 $K(x, x')$ is.

There are two typically used kernels for GPs. The first is the Squared Exponential/ Radial Basis Function (RBF):

$$K(x, x') = e^{\left(-\frac{(x-x')^2}{2l^2}\right)} \tag{3.5}$$

The other is the Matern 5/2 Kernel:

$$K_{5/2}(x, x') = \left(1 + \frac{\sqrt{5}(x-x')^2}{2l^2} + \frac{5(x-x')^4}{12l^4}\right) e^{\left(-\frac{\sqrt{5}(x-x')^2}{2l^2}\right)} \tag{3.6}$$

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. The value l represents the lengthscale of the kernel and is itself a hyperparameter, and is learnable with Maximum Likelihood Estimation (MLE).

Accounting for noise in a GP is simply adding to the diagonal of the covariance matrix:

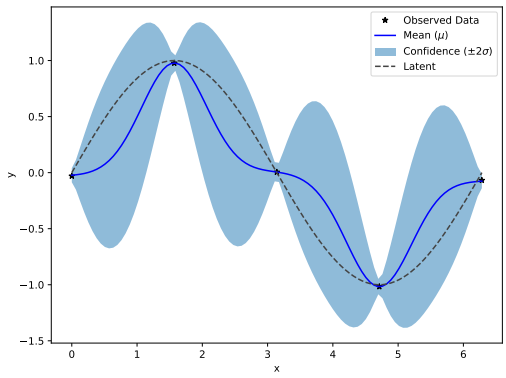
$$\begin{bmatrix} \boldsymbol{y} \\ \boldsymbol{y^*} \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} \boldsymbol{\mu}(X) \\ \boldsymbol{\mu}(X_*) \end{bmatrix}, \begin{bmatrix} K(X,X) + \sigma^2 I & K(X, X_*) \\ K(X, X_*)^T & K(X_*, X_*) \end{bmatrix}\right) \tag{3.7}$$

This then accounts for gaussian noise in the function evaluations: $\boldsymbol{y} = \boldsymbol{f} + \boldsymbol{\epsilon}$ where $\epsilon \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$.

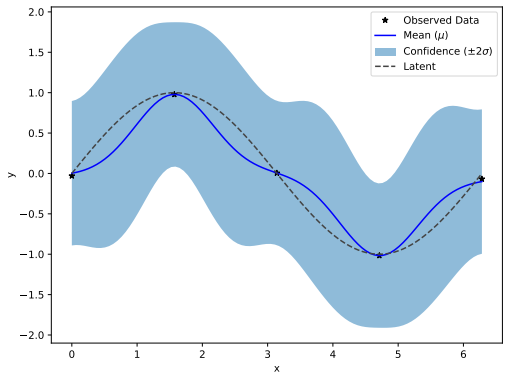
The posterior distribution is almost identical to equation (3.4) we saw previously:

$$\begin{aligned}(\boldsymbol{y^*}|X_*, X, \boldsymbol{y}) \sim &\mathcal{N}(\boldsymbol{\mu}(X_*) + K(X,X)^T(K(X,X) + \sigma^2 I)^{-1}(\boldsymbol{y} - \boldsymbol{\mu}(X)), \\ &K(X_*, X_*) - K(X, X_*)^T(K(X,X) + \sigma^2 I)^{-1} K(X, X_*))\end{aligned} \tag{3.8}$$

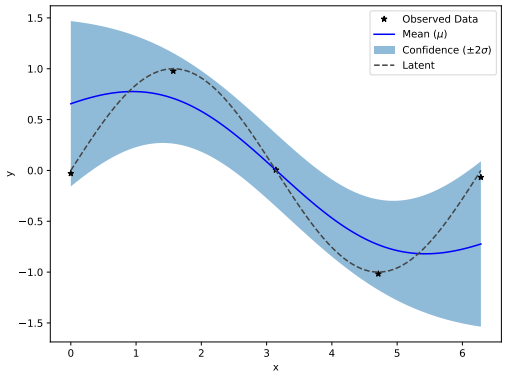
So what is σ? This is another hyperparameter! Choosing the hyperparameters of our GP l (the lengthscale) and σ (signal noise) manually would defeat the purpose of using Bayesian Optimization for AutoML. Fortunately, there is a method called Maximum Likelihood Estimation which determines the best values for l and σ automatically. 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 3.2 below shows the effect that varying the signal noise and lengthscale parameters has on the resulting GP:



(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. 3.2: 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 $[0, 2\pi]$.

From Figure 3.2 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's 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. FOBO requires that the mean $\mu(x)$ function and kernel function $K(x, x')$ are adjusted:

$$\begin{aligned} \tilde{\boldsymbol{\mu}}(x) &= (\mu(x), \nabla\mu(x))^T \\ \tilde{\boldsymbol{K}}(x, x') &= \left(\begin{bmatrix} K(x, x') & J(x, x') \\ J(x', x)^T & H(x, x') \end{bmatrix} \right) \end{aligned} \tag{3.9}$$

where $J(x, x')$ is the gradient of $K(x, x')$ with respect to x' and $H(x, x')$ is the Hessian of $K(x, x')$. Then, conditioning on the prior distribution gives us a posterior similar to equation (3.8), assuming we are dealing with noisy observations:

$$\begin{aligned} \hat{\boldsymbol{\mu}}(x) &= \tilde{\boldsymbol{\mu}}(x) + \tilde{K}(x, X)(\tilde{K}(X, X) + diag(\sigma^2(x^{(1)}), \ldots, \sigma^2(x^{(n)})))^{-1}((y, \nabla y)^{1:n} - \tilde{\boldsymbol{\mu}}(x)) \\ \hat{K}(x, x') &= \tilde{K}(x, x') - \tilde{K}(x, X)(\tilde{K}(X, X) + diag(\sigma^2(x^{(1)}), \ldots, \sigma^2(x^{(n)})))^{-1}\tilde{K}(X, x') \end{aligned} \tag{3.10}$$

Once again, we can play around with the noise and lengthscale to obtain an intuition of what those parameters represent:

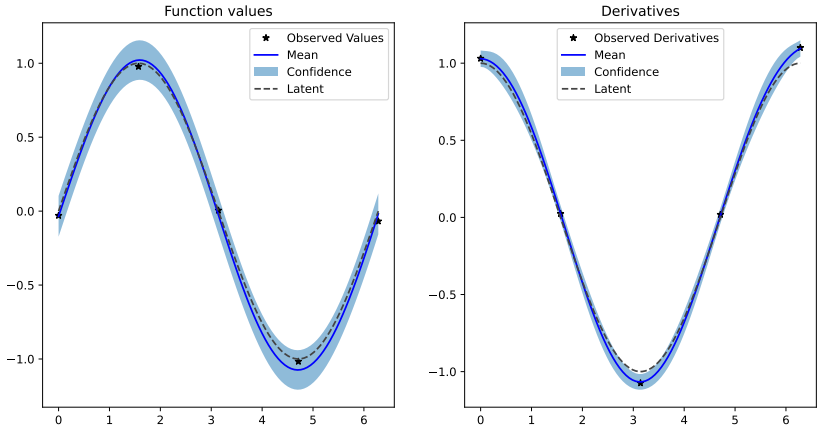


Fig. 3.3: Gaussian Process with small signal noise and a small lengthscale. The training data is noisy function evaluations of $sin(x)$ on the domain $[0, 2\pi]$.

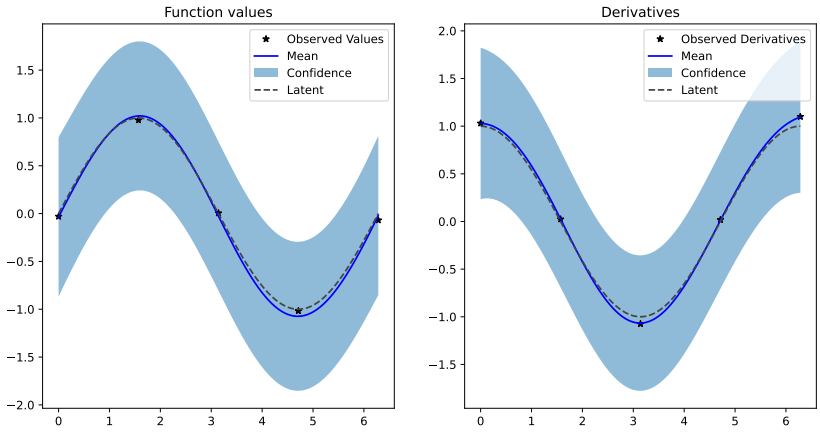


Fig. 3.4: Gaussian Process with large signal noise and small lengthscale. The training data is noisy function evaluations of $sin(x)$ on the domain $[0, 2\pi]$.

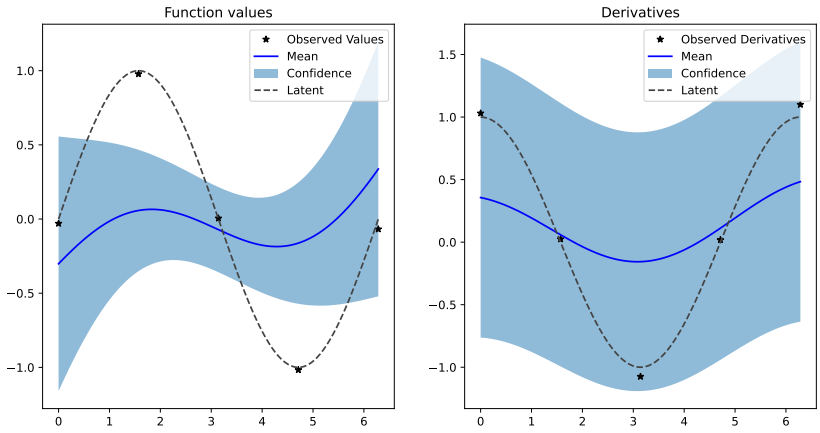


Fig. 3.5: Gaussian Process with a moderate signal noise and large lengthscale. The training data is noisy function evaluations of $sin(x)$ on the domain $[0, 2\pi]$.

From Figure 3.3 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.

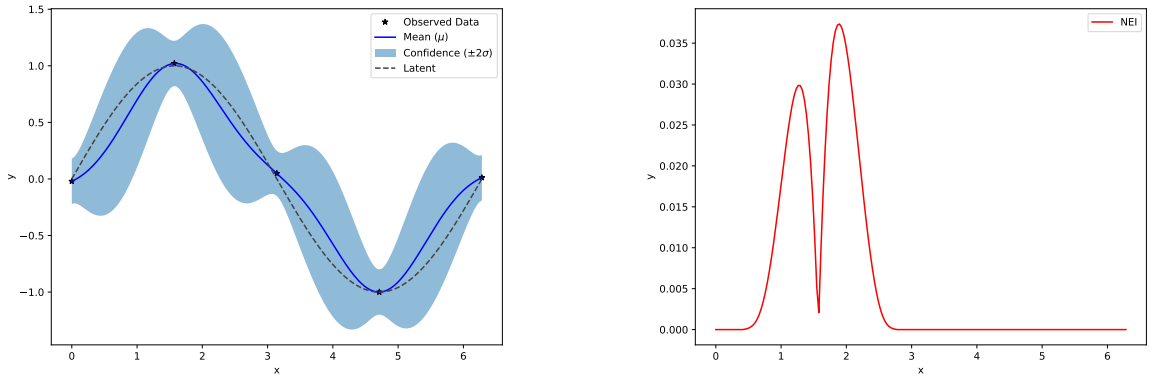
From Figure 3.3, Figure 3.4, and Figure 3.5 we see that the signal noise increases the confidence bound at each point in the Gaussian Process. The lengthscale also has a similar effect as seen for standard Gaussian Processes, where a larger value causes the training data to have a stronger effect on the mean of the GP around them.

Recall that Bayesian Optimization has two steps that are in a loop: fitting the Gaussian Process (primarily with MLE) 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.

3.3. Acquisition Functions.

Definition 3.3 (Acquisition Function). "Acquisition functions are heuristics employed to evaluate the usefulness of one of more design points for achieving the objective of maximizing the underlying black box function."

Acquisition functions are easy to understand visually. Let's assume we have a Gaussian Process like the one shown in Figure 3.6 (a). The *Noisy Expected Improvement* (NEI) acquisition function will assign a usefulness value to each point in the domain, like in Figure 3.6 (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. 3.6: 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 3.6 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 I focus on in my 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):

(3.11) $$EI_n(x') := \mathbb{E}_n\left(max(f(x') - f_n^*, 0)|x_{1:n}, y_{1:n}\right)$$

Where $(x_{1:n}, y_{1:n})$ are the training data and f_n^* is the current best point (that is, the maximum value in $y_{1:n}$). 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, $EI_n(x')$ is the integral of the posterior at x' from f_n^* to ∞. Thus, both the confidence region and the mean can contribute to a higher usefulness score at a point x'. However, this acquisition function doesn't 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:

(3.12) $$NEI_n(x') := \mathbb{E}_n\left(max(max(Y) - max(Y_{base}), 0)|x_{1:n}, y_{1:n}\right)$$

Where $(x_{1:n}, y_{1:n})$ are the training data, Y_{base} are the *random variables* at the points $x_{1:n}$, and Y is the random variable from x'. This modification takes into account the variability of the training data. So, what NEI does is assigns a point x' its usefulness based on the probability that the resulting y-value from sampling at x' is, on average, higher than the average best y-value from the sampled points $x_{1:n}$.

The final acquisition function I'll introduce 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 return only a point that has been sampled. 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:

$$KG_n(x') := \mathbb{E}_n\left(\mu_{n+1}^* - \mu_n^*|x' = y', x_{1:n}, y_{1:n}\right)$$

Where $(x_{1:n}, y_{1:n})$ are the training data, μ_n^* is the maximum mean of the posterior of the GP, μ_{n+1}^* is the maximum mean of the posterior of the GP *if x'=y' in sampled*. So, KG tries to find the point 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 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 $n+1$ dimensional vectors, so the function value that needs to be maximized is the one corresponding to function evaluations, not the gradient.

4. Software Development.

So, with the background and theory established, I began the hunt for a software package implementing Bayesian Optimization. A preliminary search led me to the `fmfn/BayesianOptimization` python library. It appeared to be regularly maintained and used by many people. However, in trying it out it became clear to me that it is a bit too sensitive to the hyperparameters that were being used, some of which I have not heard of when studying the theory of BayesOpt. This resulted in some strange behavior during hyperparameter tuning, seen in Figure 4.1:

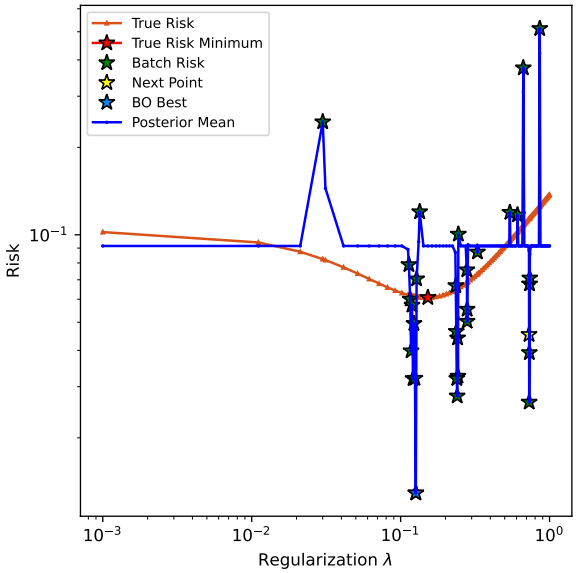


Fig. 4.1: The Python library fmfn used to find the minimum of the true loss from the example scenario developed in section 2.

Trying out different things, like scaling the training data to be logarithmic (as seen in Figure 4.1) didn't seem to result in consistent BayesOpt, since different seeds drastically changed the results. Furthermore, I later came to realize that the package did 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:

- Hypermapper
  - Owned by Dr. Stephen Becker's colleague Dr. Luigi Nardi's research group
  - Small group maintaining it

  - Doesn't have dGP support
  - I would be supported in its development by direct contacts
- Vizier
  - Owned by Google
  - Primarily for industry-level usage
  - Doesn't have dGP support
- Cornell-MOE
  - 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
- Ax
  - Owned by Meta Open Source
  - Actively maintained
  - Primarily for industry-level usage
  - Uses BoTorch under the hood
- BoTorch
  - Owned by Meta Open Source
  - Actively maintained
  - Doesn't have native dGP support
  - Used heavily by the research community

So, the choice is to either use a package filled with tech debt (Cornell-MOE) that supports derivative-based GPs, or to choose an existing maintained package that does not and then to implement dGP support myself. I first chose BoTorch, since it made more sense to me to contribute to the open source community and provide support to a widely used library. However, I switched to Ax since the tutorials mention to use Ax instead, which is equivalently open-source and maintained. I found it was very easy to use, however I quickly learned that to implement dGP support it would be better to just use BoTorch.

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. Fortunately, GPyTorch *does* support dGPs, and can be used within BoTorch! The figures that have been included in this thesis were from BoTorch and the dGPs are from GPyTorch. Figure 4.2 below shows the results of fitting a 2D dGP to noisy data:

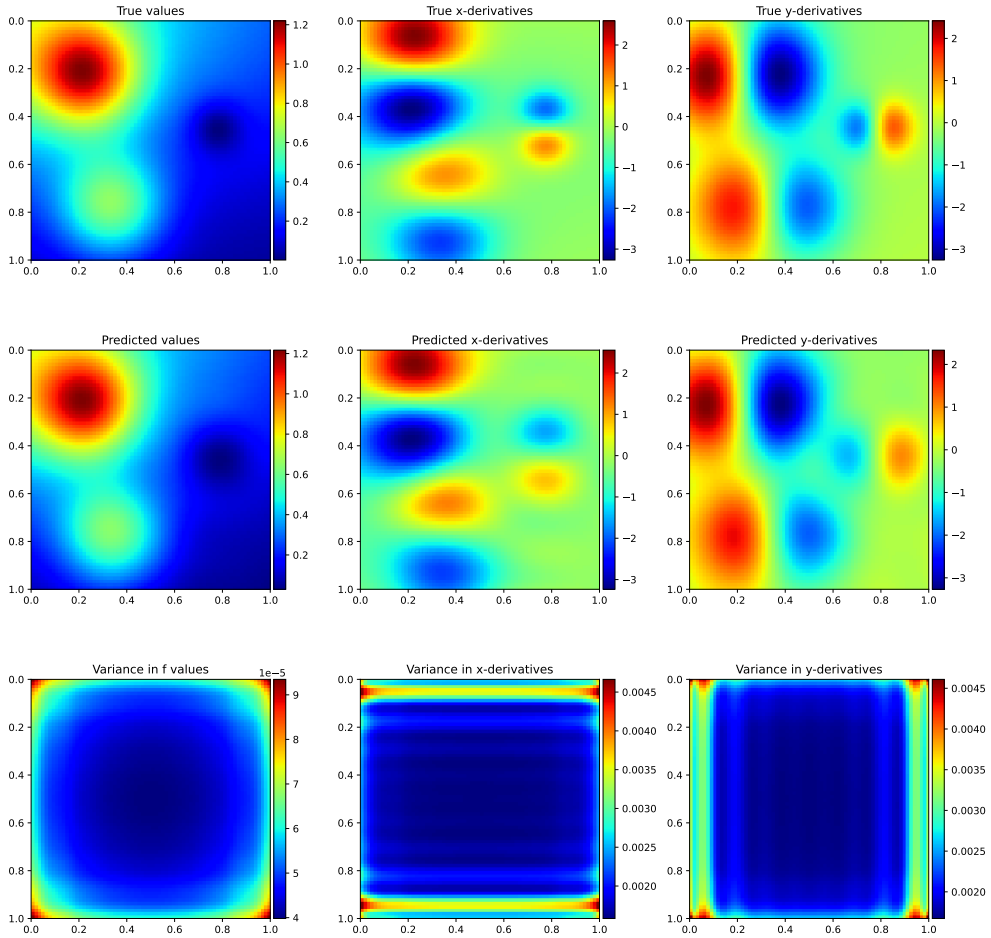


Fig. 4.2: A 2D dGP fitted to noisy data from Frankes function on the domain $[0, 1] \times [0, 1]$.

Unfortunately, dGPs are handled differently in software with respect to GPs since their means and covariance matrices are different sizes than their non-derivative-based counterparts. This means that using the NEI and KG acquisition functions resulted in various errors in BoTorch.

First, I wanted to get NEI to work. Fortunately, there is a Github Issue from January 2023 which included a demo Jupyter notebook using a GPyTorch dGP with the EI acquisition function. Unfortunately, the data they are using is noisy, which does not align with EI in theory. In swapping to NEI, I ran into 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 semi-definite:

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

The solution to this is to add to the diagonal of the covariance matrix until it is positive semi-definite. This is done by converting the covariance matrix from a PyTorch tensor to a tensor from the linear algebra package LinearOperator, which already implemented a function for adding to the diagonal of a matrix if it is not positive semi-definite. I then created a Pull Request (PR) which is now merged, officially making me a contributor to BoTorch! 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 training data that is used to fit the GP. Then, the lengthscale values were no longer incredibly small, resulting in NEI working for dGPs. Another solution to this is 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 didn't work right away. This time, I got the following error:

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

Great! So it appears that it is being actively developed, however, it turned out that the last time someone tried to resolve the issue was four years ago. However, the Pull Request gave me a head start on resolving the issue and gave me a good idea of where to look. I managed to get everything working, wrote a unit test, and submitted a PR which is still open at the time of writing this thesis.

4.1. Evaluating FOBO. Now that I 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. I decided to use the Rosenbrock function since it allows for changing the dimensionality N of the problem and has a global minimum of 0 at $(1, ..., 1)$:

$$f(\boldsymbol{x}) = \sum_{i=1}^{N-1} \left[100(x_{i+1} - x_i^2) + (1 - x_i)^2\right], \quad \boldsymbol{x} \in \mathbb{R}^N \tag{4.1}$$

I then created 12 different configurations of BO to see which method would perform the best. The methods were compared with respect to which configuration would find the lowest value of the Rosenbrock function after 50 iterations. I modified whether or not the function evaluations and/or the gradient evalua-

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 4.1: ZOBO and FOBO configurations.

tions had gaussian noise. I also modified whether or not the domain and range were standardized. Lastly, I modified whether or not the lengthscale is bounded. The configurations were as follows:

To generate the results, I ran 50 iterations of BayesOpt for 154 different seeds. Each seed provided each configuration above with the same five initialization points which were obtained from randomly sampling the domain $[-2, 2] \in \mathbb{R}^3$. Note that this means I tested BayesOpt in three dimensions, which is equivalent to optimizing three hyperparameters at the same time. The equivalent to the loss function would be the evaluations of the Rosenbrock function.

The different methods can be compared visually through a performance profile. The performance of a solver s (in my case, each configuration is a different solver) for a performance ratio α is defined as:

$$\rho_s(\alpha) = \frac{1}{|P|} size\{p \in P : r_{p,s} \leq \alpha\} \tag{4.2}$$

Where α is the domain of the plot for the performance profile and $r_{p,s} = \frac{t_{p,s}}{min\{t_{p,s}:s\in S\}}$ is the performance ratio, $t_{p,s}$ is the performance measure (which in my case is the minimum function evaluation), and p is the problem (in my case each seed is a different problem).

To make the plot, a tolerance $\tau \in [0, 1]$ needs to be chosen, which represents the accuracy for each solver as an approximation to the best solver in the performance profile. A larger value for τ represents a smaller accuracy and smaller value for τ represents a larger accuracy.

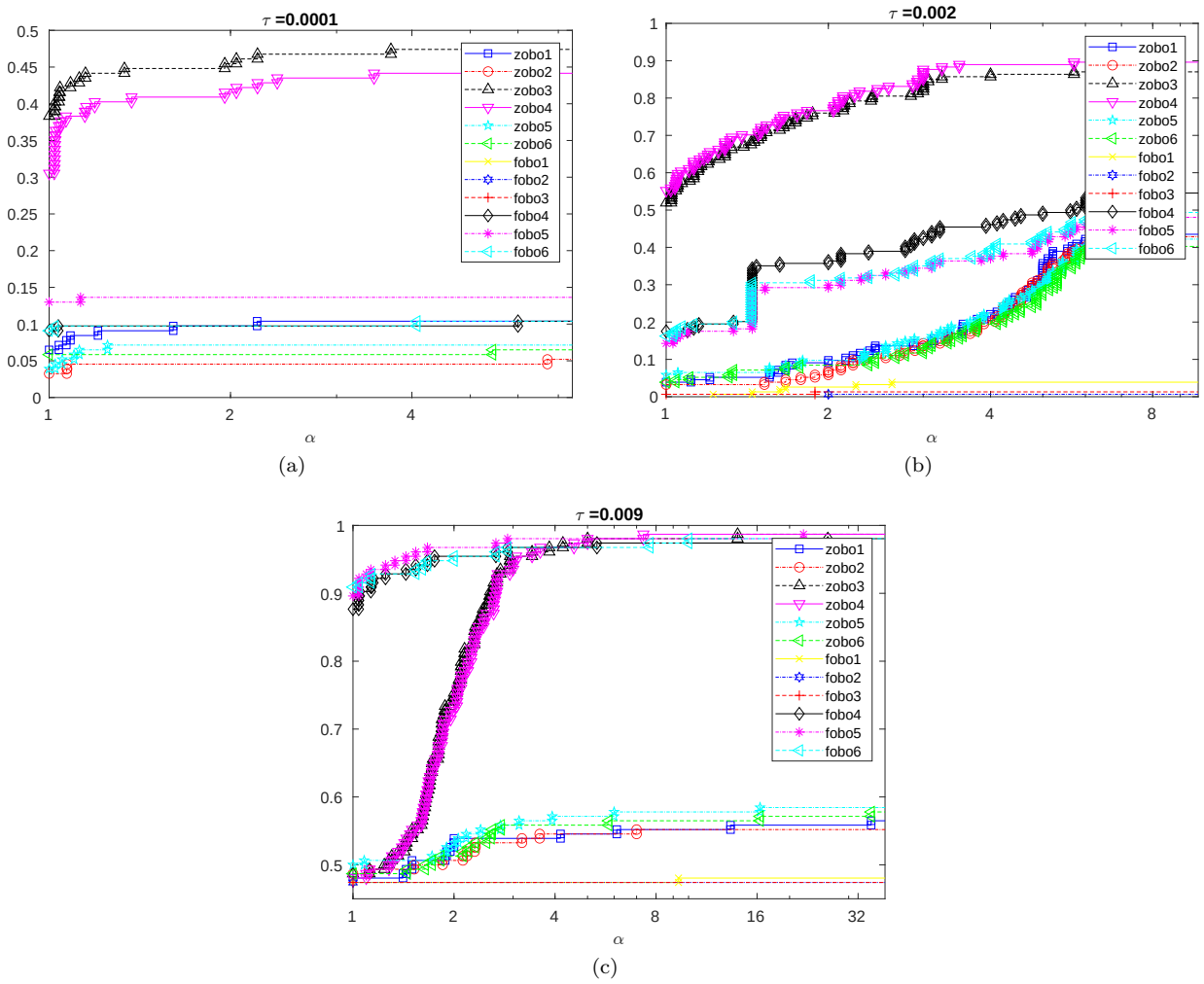


Fig. 4.3: Performance profiles for 154 seeds of BayesOpt on 12 different configurations from Table 4.1 with varying values of τ.

From Figure 4.3 (a) we see that the best BO configurations were zobo3 followed by zobo4 respectively for high-accuracy problems. From Figure 4.3 (b) We see zobo3 followed by zobo4 for slightly less accurate problems. From Figure 4.3 (c) we see zobo5, fobo5, and fobo6 perform the best for small α, meaning other methods need more function evaluations to achieve similar accuracy a significant amount of the time. However, for larger values of α, zobo3 and zobo4 catch up and are on-par with the former three methods.

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

5. Conclusion.

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, I covered the theory before attempting to tackle the motivating example. To my surprise, no maintained BayesOpt software was available that incorporated gradient information. This led me down a rabbit hole of exploring different libraries before realizing I would need to contribute to an existing library. As a result, I ended up becoming a contributor to the BoTorch library, allowing other researchers the opportunity to use dGPs with many acquisition functions.

Unfortunately, the results I obtained were not supportive of the conclusion that first-order Bayesian Optimization 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 first issue that can be improved is that there is a single signal noise parameter for GPyTorch's implementation of dGPs. 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 reality is that this method of incorporating gradient information to Gaussian Processes is flawed. From Frazier's paper (cite), using derivative information wasn't always beneficial. Other methods for incorporating gradients into BayesOpt have been researched (cite), and other unexplored implementations are bound to exist.

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. The experiment I performed was not comprehensive and may in fact work better for ZOBO at the end of the day.

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.

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.