Programming Assignment 4: Bayesian Optimization

CS4787/5777 — Principles of Large-Scale ML — Fall 2023

Project Due: Wednesday, November 29, 2023 at 11:59pm

Late Policy: Up to two slip days can be used for the final submission.

Please submit all required documents to **Gradescope**. There are two separate assignments there, one for the code and one for the report. You need to submit **both**.

This is a group project. You can either work alone, or work **ONLY** with the other members of your group, which may contain AT MOST three people in total. Failure to adhere to this rule (by e.g. copying code) may result in an Academic Integrity Violation.

Overview: In this project, you will be learning how to use Bayesian optimization to automatically tune the hyperparameters of a machine learning algorithm. This project involves a combination of the skills you've previously developed from working with PyTorch and the technical skills you've learned in class about Bayesian optimization. Note that while Project 4 explored PyTorch's high-level support for deep learning with `torch.nn`, here we'll see some more low-level aspects of PyTorch as we use it to compute gradients for the inner optimization problem of Bayesian optimization.

Background: In class, we discussed the Gaussian process prior for Bayesian optimization. The Gaussian process we described in class used a prior that was determined by a kernel, such as the RBF kernel, for which $K(x, x) = 1$. This implicitly assumes that if we evaluate the objective function multiple times with the same parameters x, then the value will be exactly the same (i.e. the variance of the difference between different evaluations is 0). This also implicitly assumes that for values of x and z that are very close together $K(x, z)$ becomes arbitrarily close to 1, and so, the variance of $f(x) - f(z)$ also becomes small, implying that they are very close together as well. More explicitly,

$$\begin{aligned}
\mathbf{E}\left[(f(x) - f(z))^2\right] &= \mathbf{E}\left[f(x)^2\right] - 2\mathbf{E}\left[f(x)f(z)\right] + \mathbf{E}\left[f(z)^2\right] \\
&= K(x, x) - 2K(x, z) + K(z, z) \\
&= 2 - 2K(x, z) \\
&= 2 - 2 \cdot \exp\left(-\gamma\|x - z\|^2\right)
\end{aligned}$$

which comes arbitrarily close to zero as x approaches z. But for a randomized process like training a ML model, this is not necessarily the case, and the objective measurements $f(x)$ can be noisy or discontinuous (which is particularly the case for metrics like error that take on discrete values). To better handle this situation, it is common to model the observations of the objective $f(x)$ as *noisy* with some noise parameter σ^2: this is discussed in more detail in the "Additive Gaussian Noise" section of this old CS4780 lecture on Gaussian processes. Using this noisy observations assumption, the predicted value of the objective observation at a test point x_* is distributed according to

$$f(x_*) \sim \mathcal{N}(\mathbf{k}_*^T(\Sigma + \sigma^2 I)^{-1}y, K(x_*, x_*) + \sigma^2 - \mathbf{k}_*^T(\Sigma + \sigma^2 I)^{-1}\mathbf{k}_*).$$

Using this assumption can also improve the behavior of Bayesian optimization by making the matrix inverses in this expression better-behaved (since nothing ensures that Σ is invertible but $\Sigma + \sigma^2 I$ certainly will be if $\sigma^2 > 0$). So we will be using this formulation with a nonzero σ^2 in this assignment.

There are many ways to solve the inner optimization problem to choose the next point x_* at which we are going to evaluate the objective. In this assignment, we are going to the following a simple heuristic. First, choose a random starting point from some distribution and run gradient descent with a fixed learning rate for some fixed number of steps. Then choose another random starting point and repeat some number of times. Finally, return the parameter vector x_* that has the smallest value for the acquisition function among all the parameter values arrived at at the end of gradient descent.

Please do not wait until the last minute to do this assignment! Just as in Programming Assignment 4, actually training the models that we will be exploring can take some time, depending on the machine you run on.

Instructions: This project is split into three parts: the implementation of Bayesian optimization for general objective functions, the exploration of its performance for a synthetic objective function, and the investigation of how Bayesian optimization can be used to automatically set the hyperparameters of models you've previously explored.

Part 1: Bayesian Optimization.

1. Implement a function `rbf_kernel_matrix` that computes a Gaussian RBF kernel matrix. The function should take in two matrix inputs $X \in \mathbb{R}^{d \times m}$ and $Z \in \mathbb{R}^{d \times n}$ and one scalar input $\gamma \in \mathbb{R}$, and it should return a matrix $\Sigma \in \mathbb{R}^{m \times n}$ such that $\Sigma_{ij} = K(X_i, Z_j) = \exp(-\gamma \cdot \|X_i - Z_j\|^2)$ where X_i denotes the ith column of X and similarly for Z_j.
    - Importantly, this function should use PyTorch operations, since we're later on going to need to **backpropagate** through it to compute derivatives. Specifically, it should take PyTorch tensors as inputs (except for γ which can just be a constant scalar) and output a PyTorch tensor.
    - You may find it useful to first develop a version of the function in pure numpy, and then convert it to PyTorch later.
2. Now implement a function `gp_prediction` that computes the distribution predicted by the Gaussian process using an RBF kernel. This function should take as input the observations you've already made, the parameter γ of the RBF kernel, the parameter σ^2 of the noise as described above, and the point x_* at which we want to make a prediction. Importantly, we will want to make predictions for many different points x_*, and there is a lot of computation (e.g. the forming of the covariance matrix Σ) that be shared among these many predictions. To enable this sharing, your function `gp_prediction` should use *currying* in the sense that it takes as input the observations already made and the kernel parameters γ and σ^2, performs any precomputation that is necessary on those parameters, and then returns a second function that takes as input x_* and returns the prediction made by the GP.
    - The prediction should be returned as a tuple of (mean, variance).
    - Just as before, this second function (the one that takes only x_* as input) should use PyTorch operations, since we're going to need to backpropagate through it later to compute a derivative with respect to x_*.
    - Since we don't need to compute derivatives with respect to the inputs to the first function, you don't *need* to use PyTorch operations in the precomputation step (you could use numpy instead), but you may as well do so. If you aren't tracking gradients, PyTorch has very little overhead compared to numpy.
3. Now implement the three acquisition functions we discussed in class.
    - Probability of improvement.
    - Expected improvement.
    - Lower confidence bound. Note that for this acquisition function, since there is an extra parameter κ I have set up the starter code to also use currying, so that you first pass the parameter κ to the function `lcb_acquisition` and then it returns a function that has the same signature as the other acquisition functions.
4. Finally, implement a function `bayes_opt` that runs the Bayesian optimization algorithm. This function should take as input
    - `objective`: the objective function we want to optimize using Bayesian optimization.
    - `d`: the dimension of the parameter vector we are optimizing over.

  - `gamma` and `sigma2_noise`: the RBF kernel and noise parameters for the Gaussian process.
  - `acquisition`: the acquisition function to use. Takes three parameters (y_{best}, the mean, and the standard deviation of the prediction, and returns a scalar.
  - `random_x`: a function that takes no inputs and returns a random parameter vector suitable to pass to the objective function. To be used in the initialization and minimization steps.
  - Parameters for gradient descent for the inner argmin step:
    - `gd_nruns`: number of independent random initializations we should use for gradient descent.
    - `gd_alpha`: learning rate for gradient descent.
    - `gd_niters`: number of steps of gradient descent to run.
  - `n_warmup`: number of warmup iterations to run.
  - `num_iters`: total number of outer iterations to run (equivalently, the total number of evaluations of the objective function to use).

You may find the provided `gradient_descent` function helpful for solving the inner minimization problem. You could also implement your own version in PyTorch if you prefer.

Part 2: Evaluating Bayesian Optimization on a Synthetic Objective.

1. To get a sense of how your method performs, run Bayesian optimization on the provided one-dimensional synthetic objective `test_objective`

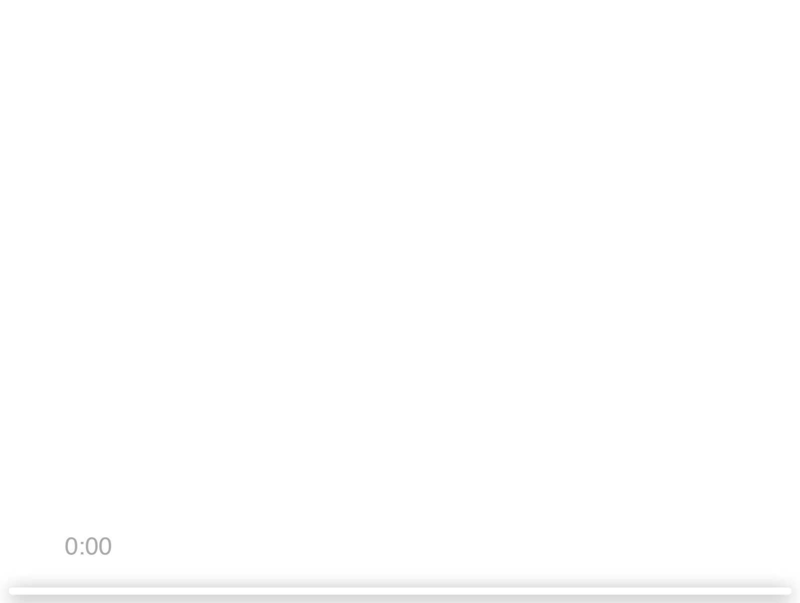
$$f(x) = \cos(8x) - 0.3 + (x - 0.5)^2.$$

   Use the following parameters:

   - RBF kernel parameter $\gamma = 10$
   - Noise variance $\sigma^2 = 0.001$
   - Random initialization uniform on $[0, 1]$ i.e. the `rand` function.
   - Run gradient descent from 20 random initializations, each for 20 steps, with a learning rate of 0.01.
   - Use 20 total iterations of Bayesian optimization, 3 of which are warmup iterations.

   Run Bayesian optimization for each of the three acquisition functions you wrote. For the lower confidence bound acquisition function, use parameter $\kappa = 2$. **Report** the best parameter value and objective value returned by Bayesian optimization for each acquisition function.

2. For at least one of your acquisition functions, use the provided `animate_predictions` function to visualize what your Gaussian process is predicting at each step of Bayesian optimization. (If you can't get this conversion to video to work on your computer, that's fine...you can fall back on some other backend for `matplotlib.animate`.) This function should take in the outputs of your Bayesian optimization function, and generate an animation. Here's an example of the output of this function, plotted for the EI acquisition function.



In this video, the blue curve represents the true objective function, the red curve represents the mean predicted by Bayesian optimization, the light blue region is a 2σ-confidence interval about the mean, and the blue dots are the parameters explored by Bayesian optimization. Briefly **report** what you observed in this video, and include some representitive screenshots from the video in your submission. (Warning: the video may take some time to render, so start early!)

- When you pass a filename to the `animate_predictions` function, **be sure to include** a `.mp4` extension or other supported video extension. If you don't include an appropriate extension, the animation library may throw a cryptic error.

3. Now explore how the performance of Bayesian optimization changes as we change the kernel hyperparameter γ. Choose one of your three aquisition functions, and observe how the loss changes when γ is changed to values that are much larger or much smaller. **Report** your conclusions, and justify them by reporting your observations for at least two runs of Bayesian optimization, one with a larger value of γ and one with a smaller value of γ.
4. Now explore how the performance of Bayesian optimization changes as we change the LCB hyperparameter κ. Using the LCB aquisition function, observe how the loss changes when κ is changed to values that are much larger or much smaller. **Report** your conclusions, and justify them by reporting your observations for at least two runs of Bayesian optimization, one with a larger value of κ and one with a smaller value of κ.

Part 3: Bayesian Optimization for Learning Hyperparameters.

1. Now, let's use Bayesian optimization to optimize the hyperparameters for a machine learning task from a previous assignment. Specifically, we're going to look at stochastic gradient descent with minibatching, sequential sampling, and momentum on MNIST, which you explored in Programming Assignment 3. For this assignment, we'll need a validation set, so the provided `load_MNIST_dataset_with_validation_split` splits the MNIST training set into a new training set of 50000 examples and a validation set of 10000 examples. The hyperparameters we want to set are α, the learning rate, β, the momentum parameter, and γ, the ℓ_2-regularization constant (not to be confused with the RBF kernel hyperparameter). Since these hyperparameters are all of different scales, we need to rescale them somehow to be on a similar scale. When you're using Bayesian optimization in practice, this is something that you'll need to do based on your intuition about the problem, but for this assignment I've suggested the following parameterization. Let $x \in \mathbb{R}^3$ be the parameter optimized by Bayesian optimization, and define

$$\gamma = 10^{-8 \cdot x_1},\ \alpha = 0.5 \cdot x_2,\ \beta = x_3.$$

This scales all the parameters such that randomly sampling each coordinate of x from $[0, 1]$ will result in a reasonable setting of the hyperparameters. Implement the function `mnist_sgd_mss_with_momentum`, which provides an objective function for Bayesian optimization. Specifically, this function trains the classifier using

SGD+Momentum with the specified parameters, and then returns the validation error minus 0.9. (This is the validation error minus what we expect the validation error would be for random guessing).

2. Run Bayesian optimization on your objective, using the following parameters:
    - For the SGD+Momentum algorithm, use a batch size of $B = 500$ and run for 5 epochs.
    - RBF kernel parameter $\gamma = 10$
    - Noise variance $\sigma^2 = 0.001$
    - LCB acquisition function with $\kappa = 2.0$.
    - Random initialization uniform on $[0, 1]$ for each coordinate i.e. the `rand` function.
    - Run gradient descent from 20 random initializations, each for 20 steps, with a learning rate of 0.01.
    - Use 20 total iterations of Bayesian optimization, 3 of which are warmup iterations.
3. **Report** the validation error and test error that result from the best set of hyperparameters returned from Bayesian optimization. This should be your only use of the MNIST test set in this assignment. How does the performance of Bayesian optimization compare to the performance of the other hyperparameter optimization methods you tried in Programming Assignment 3? Briefly explain.
4. Now design and perform some experiment to measure the fraction of overall wall clock time in your program that is spent on the evaluation of the objective function `mnist_sgd_mss_with_momentum` (as opposed to the other computations internal to Bayesian optimization). **Report** your methodology, your observations, and your conclusions about what the fraction is. What does this say about the overhead of Bayesian optimization for this task?

What to submit:

1. An implementation of the functions in main.py.
2. A lab report containing:
    - A one-paragraph summary of what you observed during this programming assignment.
    - The best parameter value and objective value returned by Bayesian optimization for each acquisition function, as described in Part 2.1.
    - A brief summary of what you observed in the video, as described in Part 2.2.
    - A report of your conclusions about the impact of the RBF parameter γ, as described in Part 2.3.
    - A report of your conclusions about the impact of the LCB κ, as described in Part 2.4.
    - The validation error and test error that result from the best set of hyperparameters returned from Bayesian optimization in Part 3.3, along with a brief explanation of how the performance of Bayesian optimization compares to the performance of the other hyperparameter optimization methods you tried in Programming Assignment 3.
    - Your methodology, observations, and conclusions about the fraction of time spent on evaluating the objective, as described in Part 3.4.
3. The video you produced in Part 2.2.

Setup:

1. Run `pip3 install -r requirements.txt` to install the required python packages