CS 6787 Final Report: A Comparison of Hyperparameter Optimization Methods: Bayesian Optimization vs Stochastic Line Search

Sophie Zheng [1] **Matthew Xu** [1]

Abstract

Hyperparameter tuning continues to be a sought after subject in machine learning. For any machine learning algorithm, it is desirable to find good hyperparameters for a given model: ones that help the model achieve high accuracy while also minimizing convergence time. We present a comparison of two methods of hyperparameter optimization on Stochastic Gradient Descent based models. The two methods of hyperparameter tuning we attempt to explore include the Stochastic Line Search and Bayesian Optimization. SLS adjusts the learning rate as the SGD algorithm steps, and eventually converges on a learning rate; BO runs SGD multiple times, each time on a different learning rate to eventually find the best one. These two methods are based on distinctly different philosophies, which leads us to wonder: which method performs better? We will test these two optimization methods on both standard binary classification problems and multiclass classification problems. In a recently published paper, *a Painless Stochastic Gradient: Interpolation, Line-Search, and Convergence Rates* (Vaswani et al., 2019), the authors of the paper demonstrated promising results of SLS on a variety of machine learning tasks: MNIST, CIFAR-10, mushrooms, etc. Building on the results found in this paper, we attempt to compare the performance of SLS on the same datasets against the Bayesian Optimization methods both in model performance and model convergence time.

1. Introduction

When it comes to solving classification problems, SGD is a standard choice. But what do we set the learning rate to? Or the batch size, or the momentum? And thus, the hyperparameter optimization problem resurfaces. Although simple and straightforward methods for hyperparameter optimization include handtuning, as well as Grid Search, other attempts to optimize this problem have become more popular in the machine learning community. In this paper, we are interested in exploring Bayesian Optimization with Gaussian Priors, as well as Stochastic Line Search, proposed by a recent paper. These two methods differ distinctly in terms of methodology: BO is more similar to grid search, in that it tries a plethora of parameters and finds the best one; while SLS only evaluates a model once, adjusting the learning rate as the model steps.

In this paper, we attempt to compare the two methods, and evaluate them based on efficiency and accuracy. We are also focusing on simple models with an SGD optimizer, and we will be using the two methods mentioned above to find the optimal learning rate for SGD.

2. Bayesian Optimization

Bayesian Optimization is a hyperparameter optimization method that aims to model parameter tuning as a function, that maps from the parameter space to $\mathbb{R}$, representing the error rate of the model with the given parameters, and find the value that minimizes the function (Snoek et al., 2012). This implies that numerous evaluations of the model must be conducted. However, this is still different from Grid Search, because BO assumes a gaussian prior and uses a function to predict the next point to evaluate.

Thus, we believe that with a good number of epochs, we will be able to converge on a good set of parameters that achieve good accuracy. However, this is expected to result in a large run time. We will discuss this more in Section 5. In this section, we will provide background information on Bayesian Optimization with Gaussian Process Priors.

2.1. Gaussian Process Priors

As mentioned before, BO attempts to model the relationship between the parameter space and the performance of the model. By assuming that this relationship is gaussian as a prior, we can assume to find the next minimum with an acquisition function (see section 2.2) as a posterior.

In addition, Gaussian Processes are multivariate normal

Algorithm 1 Bayesian Optimization

Input: model m, data X, label y, acquisition function a, best x_b, best value v
repeat
  Initialize $x = a(x_b)$
  Set target parameter in m to x
  Set e to the evaluation of m on data X and label y
  if $e > v$ **then**
    Set $x_b = x$
    Set $v = e$
  end if
until good number of epochs

distributions: every finite subset of the distributions are also a multivariate normal distribution. Thus, even though the evaluated points of the BO is constantly expanding, we can still assume the Gaussian Process prior.

2.2. Expected Improvement

BO is a series of evaluations of the model. Thus, the natural question to ask is: which point do we evaluate next? We could reformat this question mathematically:

$$x_{best} = \text{argmin}_x \, a(x)$$

where a is the acquisition function. There are some popular acquisition functions, including Probability of Improvement (PI), Expected Improvement (EI), and GP Upper Confidence Bound (UCB). Since EI and UCB are generally known for their performance, and EI is simpler in nature due to the additional required hyperparameter for UCB, our experiments will be focusing on EI as the acquisition function.

EI, as its name suggests, aims to maximize the expected improvement over the current best. We can model this function in terms of the cumulative distribution function of the standard normal distribution ($\Phi(\cdot)$) and the probability density function ($\phi(\cdot)$), as Snoek et al mentions:

$$\begin{aligned} x_{t+1} &= \text{argmax}_x \, \mathbb{E}(\max\{0, f(x_{best}) - \mu(x)\}) \\ &= \sigma(x)(\gamma(x)\Phi(\gamma(x)) + \phi(\gamma(x))) \end{aligned}$$

3. Stochastic Line Search

The goal of classic line search in deterministic optimization, is to tune the step taken by a optimization algorithm along a search direction (Mahsereci & Hennig, 2015). Stochastic Line Search is essentially classical SGD with a variant of Armijo Line Search (Vaswani et al., 2019). Instead of trying different learning rates and evaluating the model many times like BO, SLS adjusts the learning rate as it trains, reducing the need to evaluate the model to only once. For this reason, we hypothesize that SLS will faster than BO.

Algorithm 2 Stochastic Line Search

repeat
  Input: current objective f, current batch X, current weights w_k
  Compute the gradients $\nabla f_k(w_k)$ for the current batch X
  Search for learning rate η_k that satisfies the Armijo condition:

$$f_k(w_k - \eta_k \nabla f_k(w_k)) \leq f_k(w_k) - c\eta_k ||\nabla f_k(w_k)||^2$$

  Use η_k to update the model parameters with SGD:

$$w_{k+1} = w_k - \eta_k \nabla f_k(w_k)$$

until convergence

3.1. Armijo Line Search

As mentioned in (Vaswani et al., 2019), Armijo Line Search is a standard method for setting the learning rate for Gradient Descent. The algorithm is straightforward: begin with maximum learning rate $\eta_0 > 0$, search control parameters $\tau \in (0, 1)$, $c \in (0, 1)$,

1. Set $t = -cm$
2. Until the following condition is satisfied, set $\eta_{j+1} = \tau\eta_j$:
$$f(x) - f(x + \eta_j p) \geq \eta_j t$$
where p is the search direction.
3. Return η_j as the solution.

In our case, we must adapt the Armijo Condition to the stochastic setting, as we are running SGD instead of standard Gradient Descent. Thus, we have that we run the Armijo Line Search algorithm until the following condition is met:

$$f_j(w_j - \eta_j \nabla f_j(w_j)) \leq f_j(w_j) - c\eta_j ||\nabla f_j(w_j)||^2$$

3.2. Convergence Rates

We first discuss the strongly convex case. Assuming interpolation, L_i smoothness, convexity of each f_i's, and μ strong-convexity of f, let the hyperparameter $c = \frac{1}{2}$ in Armijo Line Search, we have that SGD with Armijo achieves the following rate:

$$\begin{aligned} &\mathbb{E}\left[||w_T - w^*||^2\right] \\ &\leq \max\left\{\left(1 - \frac{\bar{\mu}}{L_{max}}\right), (1 - \bar{\mu}\eta_{max})\right\}^T ||w_0 - w^*||^2 \end{aligned}$$

We then discuss the convex case. Assuming interpolation, L_i smoothness, convexity of each f_i's, we have that SGD with Armijo for all $c > \frac{1}{2}$ achieves the following rate:

$$\mathbb{E}\big[f(\bar{w}_T - f(w^*))\big]$$

$$\leq \frac{c \cdot \max\{\frac{L_{max}}{2(1-c)}, \frac{1}{\eta_{max}}\}}{(2c-1)T}||w_0 - w^*||^2$$

(proof provided in (Vaswani et al., 2019))

4. Implementation

We decided to use Python3 to develop **Algorithm 1** and **Algorithm 2**. In addition, we chose to use scikit (including sklearn and skopt) for the implementation of Bayesian Optimization and PyTorch for Stochastic Line Search. We did not use scikit for SLS because PyTorch offered more freedom to customize optimizers. Although we used different frameworks on the two methodologies, we made sure that the structures of the models used in the experiments were the same for a good comparison.

4.1. Algorithm 1

Because we wanted to test BO and SLS on both binary classification and multiclass classification, we had to adapt our Bayesian Optimizer to accept different models. In our case, the scikit-optimize, or skopt, package was perfect. The skopt library provides the `gp_minimize` function, which is a Bayesian Optimizer with Gaussian Prior that takes in an evaluation method for the model, a search space for the parameter(s), an acquisition function, and number of epochs to run.

Because `gp_minimize` passes the updated parameters through the evaluation function, we must be able to update the model with the new parameters. Thus, the model must have method `set_params` to achieve so. Furthermore, the model must be able to fit to a data given the batch during the execution of the evaluation function, so it must have method `fit` that takes in data X and label y. Thus, the sklearn library provides models perfect for this case: they implement both `fit` and `set_params`. Finally, to provide a value representing the error of the model, we decided to use the `cross_val_score` from the sklearn library, since it is perfectly compatible with the sklearn models.

For binary classification, we chose the simple logistic regression model provided by the sklearn library: `SGDClassifier`. In additon to its compatibility with the frameworks we chose to use, it also makes it easier to reproduce this model in PyTorch for **Algorithm 2**.

For multiclass classification, we chose a more complex Multi-Layer Perceptron (MLP) model provided by the sklearn library: `MLPClassifier`. Our MLP model has in addition to input and output layers two hidden layers of size 512 and 256 and utilized the logistic activation function. This was described by (Vaswani et al., 2019) to perform greatly with the help of the SGD-Armijo Optimizer. Since SLS will also be using this model for multiclass classification, we believe that this would be a great choice for our implementation for BO as well.

4.2. Algorithm 2

Based off of the implementation done by (Vaswani et al., 2019), we created a SLS Optimizer in PyTorch by writing a custom implementation of `torch.optim.Optimizer`. Initially, we wanted to use the scikit library as mentioned in **Algorithm 1**, since it worked so well with Bayesian Optimization. However, the scikit libraries, including the Tensorflow kera libraries, did not allow custom lambda learning rate schedules. The closest thing we found was in PyTorch, where it provided the `torch.optim.LambdaLR` that takes in a lambda function as its learning rate scheduler, but that only allows the learning rate to change depending on the number of epochs, which still does not fit our case. Thus, we decided to implement our own optimizer that implements the `torch.optim.Optimizer` class so that we can update the learning rate with the Armijo Line Search algorithm.

In the `step` function of our Optimizer, we require a closure, which is the criterion for which we evaluate the loss of our model. In the case of MNIST, closure is `softmax_loss` and in the case of binary classification, closure is `logistic_loss`. This closure is used to determine the step size our algorithm will use next in accordance with the Armijo Line Search condition.

In terms of Model choice, we utilized the same models as in the BO experiments to focus our experiment on the algorithms themselves rather than model differences; for our binary classifications, we utilized a simple Linear Regression Model and for our multiclass classification task, we used a MLP model with the same number of hidden layers and activation functions.

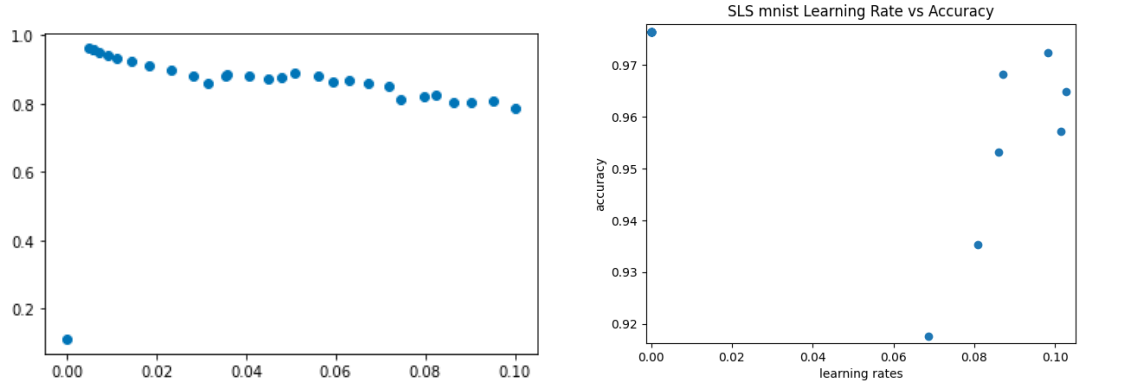
4.3. Difficulties

As mentioned before, it was very confusing for us to use different frameworks for the two algorithms we planned to implement. Because we implemented BO first, we did not realize that SLS was not compatible with sklearn until after we already finished the BO implementation for both binary and multiclass classification.

In addition, because of the lack of computing power, our Bayesian Optimization implementation took days to train on only 10 epochs. At first, it was extremely frustrating to

Table 1. Classification statistics for Bayesian Optimization and Stochastic Line Search on various data sets.

DATA SET	BEST ACCURACY BO	BEST ACCURACY SLS	BEST LR BO	BEST LR SLS	TIME BO	TIME SLS
MNIST	$96.5 \pm 1\%$	$95.754 \pm 1\%$	5.63e−03	2.41e−09	∼24 HOURS	534 SECONDS
MUSHROOMS	$89.3 \pm 1\%$	$96.8 \pm 0.5\%$	6.91e−02	6.72e−04	126 SECONDS	53 SECONDS



(a) Accuracy (y) v. Learning Rate (x) for BO MLP MNIST

(b) Accuracy (y) v. Learning Rate (x) for SLS MLP MNIST

Figure 1. Accuracy v. Learning Rate for BO (left) and SLS (right) with MLP model on MNIST dataset

find out that our implementation did not work after running the code for two or three days. Eventually, we decided to use Google Colab to train our models, since it is more stable and has more computing power, but that still took us about a day to get results from our implementation. This prevented us from experimenting with large datasets like CIFAR-10 and CIFAR-100, which would presumably take weeks of training with the limited hardware we had access to.

5. Experiments

We ran Bayesian Optimization and Stochastic Line Search on the Mushroom dataset from LibSVM, a binary classification problem, and MNIST, a classical multiclass classification problem. For Bayesian Optimization, due to the long run time in nature and lack of computing power, we ran 30 epochs for both experiments; for Stochastic Line Search, we ran 200 epochs for both experiments. In our experiments, we keep Track of the Accuracy of our model and the wall clock time our algorithms ran for.

5.1. Accuracy

Hypothesis: Bayesian Optimization will eventually achieve higher accuracy than Stochastic Line Search.

Method: As mentioned before, because of lack of computing power, we could only afford to run BO for 30 epochs. Since SLS ran quicker, we ran SLS for 200 epochs.

Results: For MNIST, BO and SLS seems to achieve around the same accuracy. In one of our experiments, Bayesian Optimization was able to achieve an accuracy of 97.6%. This was greater than any of the results produced by the SLS experiment. However, due to the stochastic nature of BO, the average accuracy is usually $\sim 95\%$, while the average accuracy for SLS is usually $\sim 97\%$, making SLS perform better than BO in general (see Figure 1). Thus, we conclude that our hypothesis has been verified for MNIST that if we run BO enough times, eventually it will achieve better than SLS for MNIST.

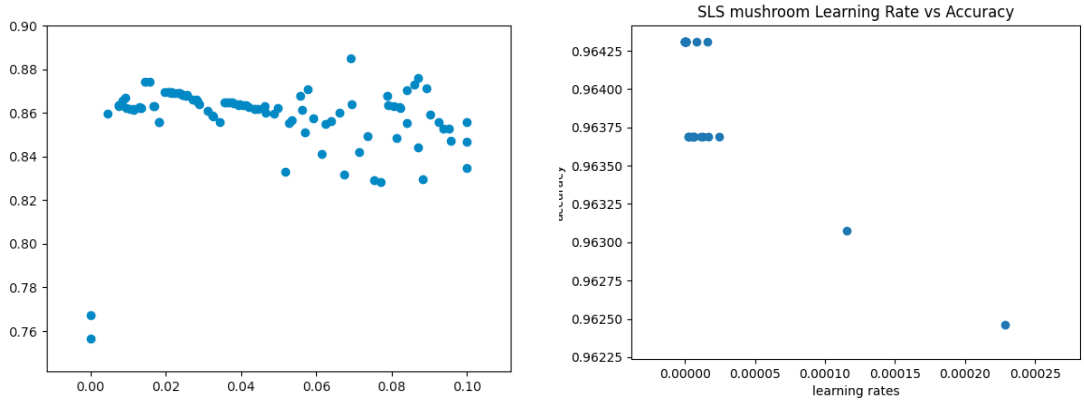
For Mushrooms, however, SLS converges quickly on an accuracy of 96.8%, while BO remains around 88% (see Figure 2). We see that our hypothesis has not been verified in this case.

Thus, our hypothesis was partially verified.

5.2. Time

Hypothesis: Stochastic Line Search will run faster than Bayesian Optimization.

Method: We use the time package in Python3 and the `time` method to obtain the wall clock time before and after the execution, then we subtract to find the wall clock time actually used for computation.



(a) Accuracy (y) v. Learning Rate (x) for BO Mushrooms (b) Accuracy (y) v. Learning Rate (x) for SLS Mushrooms

Figure 2. Accuracy v. Learning Rate for BO (left) and SLS (right) with Linear model on Mushrooms dataset

Results: For MNIST, BO ran for days for only 30 epochs. After running it on Google Colab, it still took around 24 hours. However, SLS terminated in 534 seconds for 200 epochs (see Table 1).

For Mushrooms, BO ran for 126 seconds, while SLS ran for 53 seconds. Because BO took significantly less time for this dataset, we ran both BO and SLS for 200 epochs (see Table 1).

Thus, our hypothesis was verified that SLS will run faster than BO.

6. Conclusion

While our initial hypothesis that Bayesian Optimization would eventually reach a higher accuracy than SLS was indeed valid for MNIST, the performance difference in terms of Time was drastically greater than what we expected. In most real world settings, the 0.8% accuracy increase does not justify the over one hundred fold increase in hyper parameter tuning time. While wall clock time was not an issue for the Mushrooms dataset, the accuracy of BO on the mushroom dataset was in fact lower than that of SLS despite our hypothesis. We believe SLS may be able to perform better than BO on this binary classification task because of its variable learning rate across one training period. We conjecture that BO uses a singular learning rate during each of its iterations which makes the model converge either on a local minima or diverge as a result of a too large of a learning rate. We speculate that SLS achieves the same benefits one might find for SGD with momentum, allowing for faster gradient changes in the beginning of training (by allowing for larger learning rates) but eventually tightening these changes by lowering the learning rate.

Ultimately, our experiments show that while Bayesian Optimization provides theoretically sound methodologies for finding good hyperparameters, this is at the cost of long convergence times and possibly expensive hardware. We find that Stochastic Line Search as described by (Vaswani et al., 2019) achieves an accuracy similar to or better than that of Bayesian Optimization at a fraction of the time taken.

References

Buitinck, L., Louppe, G., Blondel, M., Pedregosa, F., Mueller, A., Grisel, O., Niculae, V., Prettenhofer, P., Gramfort, A., Grobler, J., Layton, R., VanderPlas, J., Joly, A., Holt, B., and Varoquaux, G. API design for machine learning software: experiences from the scikit-learn project. In *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, pp. 108–122, 2013.

Mahsereci, M. and Hennig, P. Probabilistic line searches for stochastic optimization. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'15, pp. 181–189, Cambridge, MA, USA, 2015. MIT Press.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. Pytorch: An imperative style, high-performance deep learning library. In Wallach, H., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems 32*, pp. 8024–8035. Curran Associates, Inc., 2019. URL http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

Snoek, J., Larochelle, H., and Adams, R. P. Practical bayesian optimization of machine learning algorithms. In Pereira, F., Burges, C. J. C., Bottou, L., and Weinberger, K. Q. (eds.), *Advances in Neural Information Processing Systems*, volume 25, pp. 2951–2959. Curran Associates, Inc., 2012. URL https://proceedings.neurips.cc/paper/2012/file/05311655a15b75fab86956663e1819cd-Paper.pdf.

Vaswani, S., Mishkin, A., Laradji, I., Schmidt, M., Gidel, G., and Lacoste-Julien, S. Painless stochastic gradient: Interpolation, line-search, and convergence rates. In *Advances in Neural Information Processing Systems*, pp. 3727–3740, 2019.