Hyperparameter Tuning using Gaussian Process Multi-Arm Bandits

Arec Jamgochian, Bernard Lange

***Abstract*—Learning useful models from data generally requires fixing hyperparameters to either define model class or optimization procedure. Choice of hyperparameter can have a huge impact on model performance, but hyperparameter tuning is often labor-intensive, costly, and sub-optimal. Hyperparameter tuning can be automated by using a surrogate model to regress the choice of parameter to model score, then searching using some heuristic. In this paper, we use Gaussian Process guided by expected improvement exploration to efficiently perform Bayesian hyperparameter optimization. Our framework[1] can be easily integrated with a wide range of problems requiring high-dimensional hyperparameter optimization, including parameters that are discrete. We showcase our framework on a number of examples, including choosing optimal regularization coefficients for regression and optimizing neural network architecture for image classification.**

I. Introduction

When trying to learn a good function to model data, there are usually parameters that one must specify before learning takes place. These hyperparameters generally either parametrize a family of models to be learned, or optimization methods to learn them. Choosing the correct hyperparameters determines the success of the model and generally involves manual tuning which is often the source of much frustration. The standard practice of hyperparameter tuning usually alternates grid search with coordinate descent, as the tuner will typically focus on adjusting one parameter at a time.

One can automate the tuning of hyperparameters by building a surrogate model to regress the joint hyperparameter space to model score, then select hyperparameters using surrogate optimization. Since model training and evaluation can be quite expensive, we must use an efficient method to select candidate hyperparameters for evaluation. This can be viewed analogously to a multi-armed bandit problem, where we have to balance the *exploration* of choosing hyperparameters in a multi-dimensional design space to gather information with the *exploitation* of choosing the best possible hyperparameters. Gaussian Processes are a type of surrogate model that allow one to quantify uncertainty about a prediction across the design space [1]. Due to their incorporation of uncertainty, they lend themselves particularly well to surrogate optimization.

One can optimize exploitation in the hyperparameter search by selecting points that minimize the predicted mean of the Gaussian Process. One can optimize for exploration by selecting points where uncertainty about the predicted mean is maximal. A simple yet powerful heuristic that has proved particularly effective in the bandit setting is upper-confidence bound (UCB) exploration, in which new points are selected by minimizing a weighted sum of the two which represents some confidence bound quantile. Srinivas et al. derive regret bounds for the associated Gaussian Process upper confidence bounds (GP-UCB) algorithm, implying quick convergence [2].

An alternative heuristic for surrogate optimization using Gaussian Processes is the expected improvement (EI), or the amount by which the best score is expected to improve [3]. Maximizing expected improvement weighs the likelihood that a design point improves a model's score with the amount of improvement (Eq. 3). Expected improvement exploration has been shown empirically to converge more quickly to areas of the design space with higher fitness. An example of expected improvement exploration using Gaussian Processes can be seen in Figure 1.

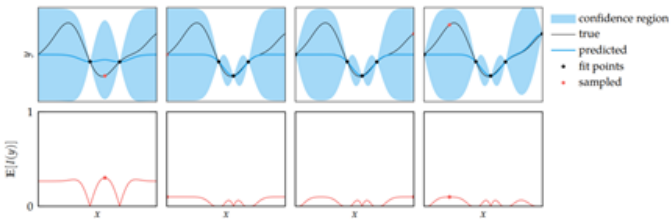

Fig. 1. Expected improvement exploration using Gaussian Processes [1].

Google's black-box hyperparameter tuning service, Vizier [4] uses Batched Gaussian Process Bandits [5] with a Matern kernel with automatic relevance determination [6] and the expected improvement acquisition function [7]. One can implement discrete parameters by embedding them as real numbers and represented categorical parameters via one-hot encoding [4]. In either scenarios, the Gaussian Process regressor provides a continuous and differentiable function which can be traversed using various optimization methods and when the convergence is achieved, the result is rounded to the nearest feasible point. A visualization of some choices for and convergence of hyperparameters can be seen in Figure 2.

In this project, we implement our own version of Gaussian Process Bandits in order to help with our future hyperparameter tuning needs. Our implementation is described in Section II. To test out our Gaussian Process Bandits implementation, we learn the best set of hyperparameters for models targeting toy problems, described in Section III. Our problems range from choosing the best regularization parameters for regression, to choosing the best neural network architectures for image classification. We conclude in Section IV.

[1]Available at:
`https://github.com/sisl/GaussianProcessBandits.py`

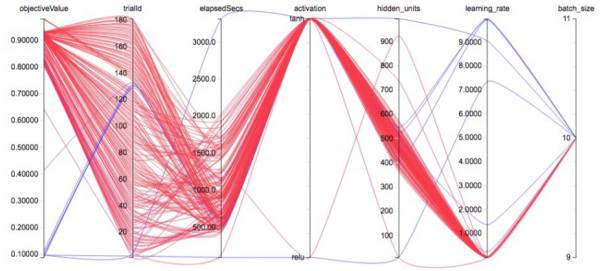

Fig. 2. The Parallel Coordinates visualization.

II. Implementation

The Gaussian Process Bandits algorithm works by attempting to regress hyperparameters in the design space to model scores. As different models are evaluated at different hyperparameter locations, the Gaussian Process is collapsed at the points in design space associated with those hyperparameter locations. For any model problem class, the user must specify functions to map between design space and hyperparameter space. The user must also specify a training and testing function to return a model score. The algorithm then iterates on choosing which hyperparameters to try next, training a model, and returning a model score.

To allow for our implementation to generalize to many classes of models, we perform the hyperparameter search in the design space of $[0,1)^n$ where n is the number of hyperparameters in the problem. We separate the problem of choosing hyperparameters from evaluating the model by requiring that each model class implement three functions:

- *decode(arr)* to set model hyperparameters appropriately when passed in an array in $[0,1)^n$,
- *encode()* to encode and return the model's set hyperparameters as an array in $[0,1)^n$,
- *train_test_cv(data)* to evaluate the model on a dataset (with whatever validation scheme is appropriate) and return a score to be minimized.

Note that the decoding scheme does not need to map to a continuous hyperparameter space. One can incorporate discrete hyperparameters by simply using *decode* to separate the design space into discrete bins (e.g. if setting a binary option). As a result, an encoding of a decoding does not necessarily map to the originally chosen design point.

With these functions implemented on each model class, the unbatched Gaussian Process Bandit implementation follows the following workflow, which is visualized in Figure 3:

1) Initial model evaluation: perform model evaluation using the default hyperparameters for the model. Add the (design point, model score) tuple to the Gaussian Process

2) Sample candidate design points: sample m candidate points randomly in $[0,1)^n$. The number of candidate points m can be extremely large since evaluating their expected mean and variance using the Gaussian Process requires inverting a matrix independent of the number of candidate points. Currently a random sampling plan is used, but one can implement a uniform sampling plan to spread samples better across the design space.

3) Estimate mean and variance of score at candidate design points: marginalize the Gaussian Process to form the posterior at the m candidates. With a zero mean function, kernel function K, and Gaussian Process noise parameter ν, the predicted mean $\hat{\mu}$ and variance of the predicted mean $\hat{\sigma}^2$ at the candidate points can be found using

$$\hat{\mu}(\mathbf{x}) = \mathbf{K}(\mathbf{x}, X)\mathbf{K}(X, X) + \nu I)^{-1} y \quad (1)$$

$$\hat{\sigma}^2(\mathbf{x}) = \mathbf{K}(\mathbf{x}, \mathbf{x}) - \mathbf{K}(\mathbf{x}, X)(\mathbf{K}(X, X) + \nu I)^{-1}\mathbf{K}(X, \mathbf{x}), \quad (2)$$

where $\mathbf{x}$ is a vector of candidate points, X is a vector of previously evaluated design points, and y is a vector of their associated model scores. The noise parameter ν is included to overcome variability in model score given differences in scores of models trained with the same hyperparameters. In our experiments, we use the squared exponential kernel, with a single characteristic length scale, $K(x, x') = exp(\frac{-||x-x'||_2^2}{2l^2})$.

4) Choose the best candidate design point: choose the design point that maximizes expected improvement. Once the predicted mean and variance of the predicted mean are calculated, expected improvement can be calculated at each candidate design point as

$$\begin{aligned}\mathbb{E}_y(I(\mathbf{x}, y)) =& (y_{min} - \hat{\mu}(\mathbf{x}))P(y \leq y_{min} \mid \hat{\mu}(\mathbf{x}), \hat{\sigma}^2(\mathbf{x})) \\ &+ \hat{\sigma}(\mathbf{x})\mathcal{N}(y_{min} \mid \hat{\mu}(\mathbf{x}), \hat{\sigma}^2(\mathbf{x})),\end{aligned} \quad (3)$$

where y_{min} is the minimum score observed thus far.

5) Set the model hyperparameters and evalulate the model: set the hyperparameters by decoding the best candidate design point. Evaluate the model using *train_test_cv(data)* and return the model score. Add the (design point, model score) tuple to the Gaussian Process. It is important to add the original design point rather than an encoded version of the decoded design point to the Gaussian Process to avoid continually exploring the same space in discrete problems.

6) Iterate: return to step 2 while iterations remain. An additional intermediary step before iterating could include fitting Gaussian Process parameters (i.e. characteristic length scales and noise variance) to best fit observed data. This could be done by using a gradient-based method (e.g. L-BFGS) to minimize the negative log likelihood of observed model scores given evaluated design points and Gaussian Process parameters. We could use this to better learn dependencies between hyperparameter dimensions (in the form of a non-uniform kernel characteristic length matrix), and ultimately speed up convergence.

III. Experiments

Our optimization framework is evaluated on benchmark tasks with known optimal solutions discussed in the following subsections.

A. Ridge Regression

The task, also known as linear regression with Tikhonov regularization, is to find the optimal regularization constant $\lambda \in \mathbb{R}^{++}$ which minimizes model mean squared error on a holdout set. The optimal regularization constant reduces variance in the conventional linear regression approach by

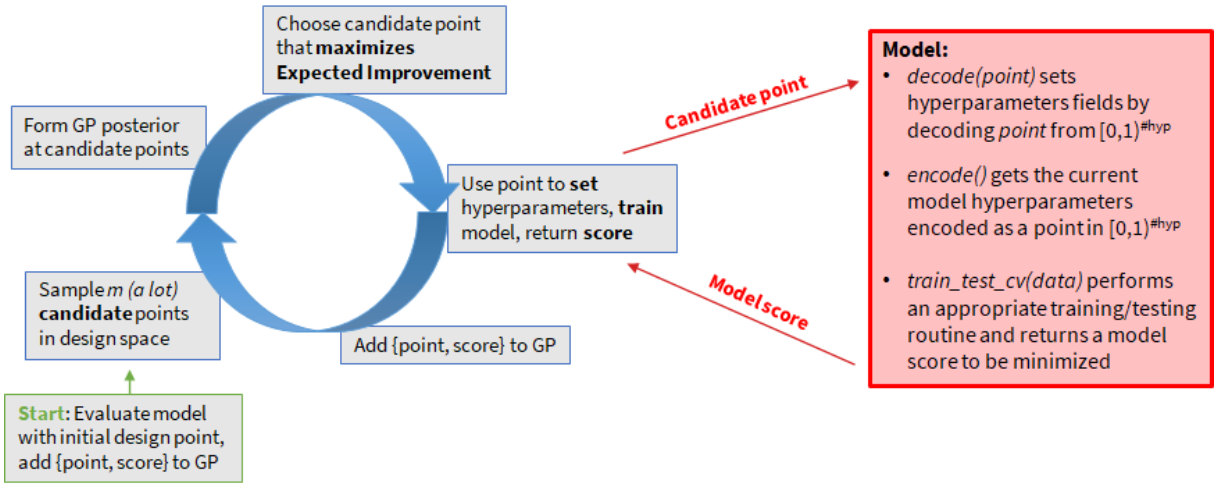

Fig. 3. Hyperparameter tuning using Gaussian Processes guided by expected improvement exploration.

introducing a Gaussian prior on the model parameters $\mathbf{w}$. This is equivalent to adding an L2-norm on the model parameters to the linear regression cost function J, as shown in Equation 4 [8].

$$J(\mathbf{w}) = \frac{1}{N}\sum_{i=1}^{N}(y_i - \mathbf{w}^T\mathbf{x}_i)^2 + \lambda||\mathbf{w}||_2^2 \tag{4}$$

To use with our hyperparameter optimization framework, the decoding scheme maps to values of L2-norm coefficient λ in natural log space. For any given value of λ, the optimal model parameters are determined explicitly using the closed form solution $\mathbf{w} = (X^TX + \lambda I)^{-1}X^Ty$, where X is a matrix of data point and y are the values to regress to. We perform k-fold cross-validation and minimize the mean squared error as a function of λ, which is provided as the score for our optimization framework to minimize. We use this toy problem to compare Gaussian Process Bandits to a random search. The result, shown in Figure 4, visualizes quicker convergence to the optimal score value.

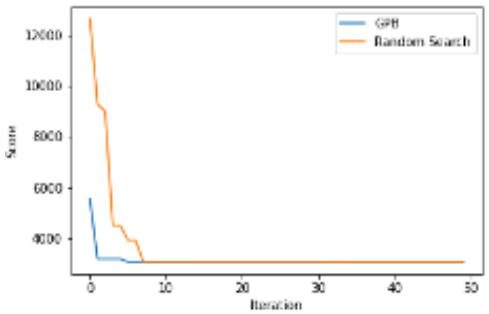

Fig. 4. **Best performing model score vs. iterations of hyperparameter search for Ridge regression**. Comparison between Gaussian Process Bandits algorithm and a random search.

B. Elastic Net Regression

Elastic Net regression is an extension of the previous problem, where we also add an L1-norm regularization term to our loss function. L1 regularization has the added benefit of not only reducing model variance, but also encouraging sparsity in the model parameter vector $\mathbf{w}$. Now we must search over a 2D space for optimal L1- and L2-norm weighting parameters λ_1 and λ_2 as shown in Equation 5 [9].

$$J(\mathbf{w}) = \frac{1}{N}\sum_{i=1}^{N}(y_i - \mathbf{w}^T\mathbf{x}_i)^2 + \lambda_1||\mathbf{w}||_1 + \lambda_2||\mathbf{w}||_2^2 \tag{5}$$

The hyperparameter decoding now maps to values of both L1 and L2 regression weights λ_1 and λ_2 in natural log space, respectively. The model parameters $\mathbf{w}$ are identified by performing gradient descent using Scikit-Learn package [10]. The rest of the implementation is equivalent to that of Ridge regression. The results are shown in Figure 5 with similar convergence properties as in Ridge regression.

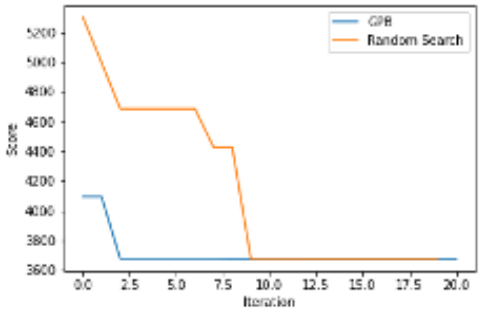

Fig. 5. **Best performing model score vs. iterations of hyperparameter search for Elastic Net regression**. Comparison between Gaussian Process Bandits algorithm and a random search.

C. Gaussian Mixture Model

In this unsupervised learning example, the task is to determine the optimal number of clusters $k \in \mathbb{N}$ to use to fit a Gaussian Mixure Model (GMM) to a set of data. This is done by picking k, fitting a GMM using the Expectation Maximization algorithm (EM), and trying to maximize the log likelihood of a holdout set of data [8].

The decoding maps $[0, 1)$ to a discrete number of clusters k between 1 and k_{max}. During training and testing, we fit 70% of a dataset to a GMM with k centers, then evaluate the average negative log-likelihood on the holdout data. We repeat this process m times choosing the holdout data at random, and return a single averaged negative log-likelihood to be minimized as a function of k. The model converges quicker to the optimal number of clusters than a conventional grid search approach.

D. Neural Network Architecture Design

In our final example, the task is to design the optimal neural network architecture to perform the well-studied problem of digit classification using the MNIST dataset of handwritten digits [11]. The general architecture design, as shown in Figure 6, consists of flattened input layer, a series of hidden layers with activation function and dropout probability determined by the optimizer, and the output classification layer.

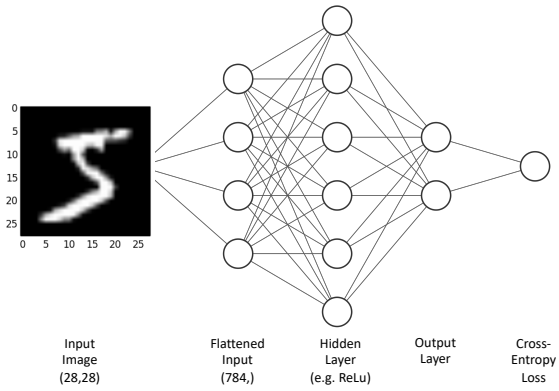

Fig. 6. General fully-connected neural network architecture.

The score to be minimized is the cross-entropy loss for network output $\mathbf{y}$ and correct *class* as shown in Equation 6.

$$J(\mathbf{y}, class) = -\log\left(\frac{\exp(\mathbf{y}[class])}{\sum_j \exp(\mathbf{y}[j])}\right) \tag{6}$$

The hyperparameter search domain and the decodings from a point in design space (denoted *point*) to hyperparameter space are as follows:

1) Hidden layers: the total number of hidden layers. The decoding to hyperparameter fields with a predefined maximum number of hidden layers (5) is:

$$HidLayers = int(point[0] * MaxHidLayers) + 1$$

2) Units per hidden layer: the number of hidden units in each hidden layer, which remains fixed in this architecture. The decoding to hyperparameter field with a predefined maximum number of hidden units (64) is:

$$HidUnits = int(point[1] * MaxHidUnits) + 1$$

3) Activation function type: : whether to use the Rectified Linear Unit (ReLU) or the Hyperbolic Tangent (Tanh) activation function. The same activation function is applied after all layers except the output layer. The decoding is implemented using the Near Integer function (Nint):

$$Activation = \begin{cases} ReLU(), & \text{if } Nint(point[2]) = 0 \\ Tanh(), & \text{otherwise} \end{cases}$$

4) Dropout probability between layers: the probability of any given node being zeroed out. The value is the same for all hidden layers. The decoding from design space to hyperparameter space is:

$$Pr_{drop} = point[3]$$

Each model is tuned using the Adam optimizer with default parameters as implemented in PyTorch. Training is done with a batch size of 32 for 200 epochs [12], [13]. In future work, the optimization hyperparameters, batch size and convergence criteria can be added without increased complexity of the implementation. The training and testing datasets are predefined according to [11] to enable valid comparisons. The training set size and test set size are 60,000 and 10,000 samples, respectively.

Table I shows the top five architectures found during 200 iterations of running the optimization algorithm. As can be seen, the best-performing architecture on the test set is similar, given the constraints such as maximum number of hidden layers and hidden units, to the architecture found in [14]. The best architecture contain one hidden layer, 62 hidden units, ReLU activation function and the dropout probability of 0.149.

TABLE I
TOP 5 BEST-PERFORMING ARCHITECTURES.

Hidden Layers	Units/Layer	Activation	Dropout Pr
1	62	ReLU	0.149
3	43	Tanh	0.076
1	38	Tanh	0.082
3	33	ReLU	0.100
3	62	ReLU	0.240

IV. CONCLUSIONS

In this paper, we explored ways to automate the costly, time-intensive, and sub-optimal process of hyperparameter tuning. Specifically, we implement a Bayesian hyperparameter optimization framework using Gaussian Process Bandits. We use a Gaussian Process as a surrogate model to regress hyperparameter design space to model score, and expected improvement exploration to efficiently guide hyperparameter search. We generalize our framework by mandating a decoding scheme to decode points from a common design space to a continuous or discrete hyperparameter space specific to many classes of problems.

We validate our implementation with four different examples - ridge regression, elastic net regression, clustering using

Gaussian Mixture Models, and neural network architecture design for digit classification. In all cases, we efficiently converge to well-performing solutions. Our framework is available at `github.com/sisl/GaussianProcessBandits.py`.

In the future, we will make improvements to our Gaussian Process Bandits implementation to better utilize CPU and GPU capabilities by a) vectorizing kernel matrix formation and b) batching the design evaluations to enable algorithm parallelization. Additionally, we will continually adapt our choices of Gaussian Process parameters (characteristic length matrix, noise variance, and kernel function) to better fit observed data and quicken convergence. We will also investigate the variation of possible neural network designs by adding more operators (e.g. convolution) and by adding a capability to tune optimizer's hyperparameters (e.g. learning rate, momentum).

REFERENCES

[1] M. J. Kochenderfer and T. A. Wheeler, *Algorithms for optimization*. Mit Press, 2019.

[2] N. Srinivas, A. Krause, S. M. Kakade, and M. Seeger, "Gaussian process optimization in the bandit setting: No regret and experimental design," *arXiv preprint arXiv:0912.3995*, 2009.

[3] D. R. Jones, M. Schonlau, and W. J. Welch, "Efficient global optimization of expensive black-box functions," *Journal of Global optimization*, vol. 13, no. 4, pp. 455–492, 1998.

[4] D. Golovin, B. Solnik, S. Moitra, G. Kochanski, J. Karro, and D. Sculley, "Google vizier: A service for black-box optimization," in *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining*, 2017, pp. 1487–1495.

[5] T. Desautels, A. Krause, and J. W. Burdick, "Parallelizing exploration-exploitation tradeoffs in gaussian process bandit optimization," *Journal of Machine Learning Research*, vol. 15, pp. 3873–3923, 2014.

[6] C. Williams, "Gaussian processes for machine learning," University Lecture, 2007.

[7] R. Benassi, J. Bect, and E. Vazquez, "Robust gaussian process-based global optimization using a fully bayesian expected improvement criterion," in *International Conference on Learning and Intelligent Optimization*. Springer, 2011, pp. 176–190.

[8] K. P. Murphy, *Machine learning: a probabilistic perspective*. MIT press, 2012.

[9] H. Zou and T. Hastie, "Regularization and variable selection via the elastic net," *Journal of the royal statistical society: series B (statistical methodology)*, vol. 67, no. 2, pp. 301–320, 2005.

[10] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[11] Y. LeCun, C. Cortes, and C. Burges, "Mnist handwritten digit database," *ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist*, vol. 2.

[12] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[13] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems*, 2019, pp. 8024–8035.

[14] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.