Hyper-parameter optimization

Xuedong Shang XUEDONG.SHANG@INRIA.FR

Supervisor & Co-advisor:

Abstract

1. Introduction

Modern machine learning algorithms often contain many nuisance parameters that cannot be learned through the learning process, but instead, need to be manually specified. It is thus appealing to design algorithms that require fewer such so-called *hyper-parameters* (not always feasible though).

An alternative is to perform the *hyper-parameter optimization* (HPO) that can be viewed as a *black-box/global optimization* problem where function evaluations are supposed to be very expensive. Here a typical function evaluation involves running the primary machine learning algorithm to completion on a large and high-dimensional dataset, which often takes a considerable amount of time or resources. This limits vastly the number of evaluations that could be carried out, which makes it desirable to design efficient high-level algorithms that automate this tuning procedure.

Related literature Several naïve but traditional ways of performing the hyper-parameter search exist, such as `GridSearch` and `RandomSearch`.

`GridSearch` is an old-fashioned but commonly used method in a lot of research papers before 2010s (LeCun et al., 1998), as well as in many machine learning softwares such as `LIBSVM` (Chang and Lin, 2011) and `scikit-learn` (Pedregosa et al., 2011). It simply carries out an *exhaustive* searching of parameters through a manually specified *finite* subset of the hyper-parameter search space. `GridSearch` clearly suffers from the *curse of dimensionality*, which makes it undesirable in real applications.

`RandomSearch` overcomes this problem by randomly picking hyper-paramter configurations from the search space, and can be easily generalized to continuous space as well. It is shown to outperform `GridSearch` especially in the case that the *intrinsic dimensionality* of the optimization problem is low (Bergstra and Bengio, 2012). Hence we use `RandomSearch` as a baseline method here in this section. Note that both methods are *embarrasingly parallel*, which can be a very strong point in some situations.

Recent trending solutions for HPO are mostly Bayesian-based (Bergstra et al., 2011; Hutter et al., 2011; Snoek et al., 2012, 2015). *Bayesian optimization* (BO) depends on a prior belief, typically a Gaussian process (GP), on the target function that we can update to a posterior distribution over the function space where lies the target function given a sequence of observations. It then decides where to sample next with the help of an *utility/acquisition function* by maximizing/minimizing the utilitly w.r.t. the posterior distribution. Typical acquisition functions include `EI`, `PI` (Mockus et al., 1978) and `GP-UCB` (Srinivas et al., 2010)

(see Brochu et al. 2010; Shahriari et al. 2016 for a survey). Note that classic BO methods with GP prior and typical acquisition functions can be applied straightforwardly to hyper-parameter tuning (Snoek et al. 2012 provides a Python package called `Spearmint` to perform the task), some variants like `TPE` (Bergstra et al., 2011) and `SMAC` (Hutter et al., 2011) are more commonly used though.

Remark 1 *One may notice that a new optimization task emerges in a Bayesian optimization procedure, that is to optimize the acquisition function. It may seem to be a bit artificial, but this acquisition function is usually much more regular compared to the target function, thus easier to optimize.*

Bayesian optimization focuses on adaptively choosing different parameter configurations based on previous observations, but always run the primary machine learning classifier/regressor into completion given a set of hyper-parameters. In a bandit point of view, this setting can be considered as a *stochastic infinitely-armed bandit* (SIAB) problem.

Another take on this problem is to adaptively allocate *resources* to more promising configurations. Resources here can be time, dataset subsampling, feature subsampling, etc. In such a setting, the classifier is not always trained into completion given a parameter configuration, but rather is stopped early if it is shown to be bad so that we can allocate more resources to other configurations. This idea of early stopping is proposed by Li et al. (2016), which states the HPO problem as a *non-stochastic infinitely-armed bandit* (NIAB) problem.

In this report, we treat both the two settings mentioned previously. Before that, we need to mention that totally different types of approaches also exist, for example *evolutionary optimization* (EO). EO follows a process inspired by the biological concept of *evolution*, which repeatedly replaces the worst-performing hyper-parameter configurations from a randomly initialized population of solutions.

Finally, hierarchical bandit algorithms like `HOO` (Bubeck et al., 2010), `HCT` (Azar et al., 2014) could be potential candidates for HPO as well. And to the best of our knowledge, these methods have never been investigated in the HPO literature.

2. `Hyperband` coupled with a `ThompsonSampling`-like algorithm

We propose a new heuristic based on the Top-Two Thompson Sampling (`TTTS`, see Russo 2016. The idea is similar to `Hyperband`. By running several brackets of `TTTS` with different number of configurations n and the same budget, we try to trade off between the number of configurations and the number of resources allocated to each configuration. Unlike `Hyperband` where the number of resources is fixed for each configuration in every bracket, here the number of resources is decided by the underlying `ThompsonSampling`.

This heuristic requires three inputs β, γ, B, and $s_{\max}$: (1) γ characterizes how fast does the number of configurations we want to evaluate in each bracket shrink, (2) B is the total budget, and (3) $s_{\max}$ represents the index of the largest bracket (containing the largest number of configurations).

The subroutine used here is `TTTS` where we make use of a prior distribution Π_1 over a set of parameters Θ, where for each $\theta \in \Theta$, θ_i caraterizes configuration i for $i \in \{1 \dots n\}$. Based on a sequence of observations, we can update our beliefs to attain a posterior distribution

Π_t. At each time step t, the subroutine samples θ from Π_t, then with probability β, the subroutine evaluates the configuration I with the best paramter $\theta_I = \max_{c \in C} \widehat{\theta}_c$. Otherwise it samples a new θ from Π_t until we obtain a different configuration $J \neq I$ such that $\theta_J = \max_{c \in C} \widehat{\theta}_c$.

Remark 2 *Note that this algorithm does not require computing or approximating the optimal action probabilities, which could be computationally heavy. However in practice, sometimes it could be ridiculously long to sample a J that is different from I, especially when the best configuration is far better than the others. To avoid this issue, we can either explicitly compute the optimal action probabilities, or just play the second best one when this kind of situation occurs.*

Algorithm 1: Heuristic (based on `TTTS`)

Input : β; γ; B; $s_{\max}$
Initialize: `budget` $= \lfloor B/s_{\max} \rfloor$; $L = \emptyset$; $t = 0$

1 **for** $s \leftarrow s_{\max}$ **to** 0 **do**
2 $\quad n = \left\lceil \frac{s_{\max}+1}{s+1} \gamma^s \right\rceil$;
3 $\quad C =$ get_hyperparameter_configurations(n);
4 $\quad$ **for** $c \in C$ **do**
5 $\quad\quad L = L \cup \{\text{run\_then\_return\_val\_loss}(c)\}$;
6 $\quad$ **end**
7 $\quad t = n$;
8 $\quad$ // Begin `TTTS`;
9 $\quad$ **while** $t <$ *budget* **do**
10 $\quad\quad$ Sample $\widehat{\theta} \sim \Pi_t$; $I \leftarrow \arg\max_{c \in C} \widehat{\theta}_c$;
11 $\quad\quad$ Sample $b \sim$ Bernoulli(β);
12 $\quad\quad$ **if** $b = 1$ **then**
13 $\quad\quad\quad L = L \cup \{\text{run\_then\_return\_val\_loss}(I)\}$;
14 $\quad\quad$ **else**
15 $\quad\quad\quad$ Repeat sample $\widehat{\theta} \sim \Pi_t$; $J \leftarrow \arg\max_{c \in C} \widehat{\theta}_c$ until $I \neq J$;
16 $\quad\quad\quad L = L \cup \{\text{run\_then\_return\_val\_loss}(J)\}$;
17 $\quad\quad$ **end**
18 $\quad\quad$ Update Π_t;
19 $\quad\quad t = t + 1$;
20 $\quad$ **end**
21 **end**
22 **return** *Configuration with the smallest intermediate loss seen so far*

The pseudo-code is shown in Algorithm 1. Let us give some more details on how to update the posterior belief (Line. 17 in Algorithm 1). Currently, we assume a Beta prior which is usually associated with Bernoulli bandits. Here in our case, however, the reward (or more precisely, the loss) lies in $[0, 1]$, we thus need to adapt the Thompson Sampling process to the general stochastic bandits case. One way to tackle this is the binarization trick shown in Algorithm 2 inspired by **?**.

Algorithm 2: Detailed TTTS with Beta prior for general stochastic bandits

Input : $n \leftarrow |C|$(number of arms), α_0, β_0, β
Initialize: $\forall c \in C, S_c = 0$(number of successes), $F_c = 0$(number of failures)
for $t \leftarrow 1$ **to** B **do**
  $\forall c \in C$, sample $\widehat{\theta}_c \sim \text{Beta}(S_c + \alpha_0, F_c + \beta_0)$; $I \leftarrow \arg\max_{c \in C} \widehat{\theta}_c$;
  Sample $b \sim \text{Bernoulli}(\beta)$;
  if $b = 1$ **then**
    Evaluate configuration I, and observe loss $\widetilde{l}_t$;
  else
    Repeat $\forall c \in C$, sample $\widehat{\theta}_c \sim \text{Beta}(S_c + \alpha_0, F_c + \beta_0)$; $J \leftarrow \arg\max_{c \in C} \widehat{\theta}_c$ until $I \neq J$;
    Set $I \leftarrow J$;
    Evaluate configuration I, and observe loss $\widetilde{l}_t$;
  end
  Sample $r_t \sim \text{Bernoulli}(1 - \widetilde{l}_t)$;
  if $r_t = 1$ **then**
    $S_I \leftarrow S_I + 1$;
  else
    $F_I \leftarrow F_I + 1$;
  end
  $t = t + 1$;
end

3. Hyper-parameter tuning as NIAB problem

We consider in this section the non-stochastic setting where several classifiers such as logistic regression, Multi-Layer Perceptron (MLP) and Convolutional Neural Networks (CNN) are trained on the MNIST dataset using mini-batch *stochastic gradient descent* (SGD). This part of code (code for classifers with eventual usage of GPU) is based on code available at http://deeplearning.net/.

Dataset The MNIST dataset is pre-split into three parts: training set D_{train}, validation set D_{valid} and test set D_{test}.

Hyper-parameters The hyper-parameters to be optimized are listed below in Table 1, 2 and 3. For logistic regression, the hyper-parameters to be considered are learning rate and mini-batch size (since we are doing mini-batch SGD). For MLP, we take into account an additional hyper-parameter which is the l_2 regularization factor. For CNN, we take into account the number of kernels used in the two convolutional-pooling layers.

Resource Allocation The type of resource considered here is the number of epochs, where one epoch means a pass of training through the whole training set using SGD. Note that this is similar to the original Hyperband paper where one unit of resources corresponds to 100 mini-batch iterations for example. One epoch may contain a various number of mini-batch iterations depending on the mini-batch size.

Hyper-parameter	Type	Bounds
learning_rate	$\mathbb{R}^+$	$\left[10^{-3}, 10^{-1}\right]$ (log-scaled)
batch_size	$\mathbb{N}^+$	$[1, 1000]$

Table 1: Hyper-parameters to be optimized for logistic regression with `SGD`.

Hyper-parameter	Type	Bounds
learning_rate	$\mathbb{R}^+$	$\left[10^{-3}, 10^{-1}\right]$ (log-scaled)
batch_size	$\mathbb{N}^+$	$[1, 1000]$
l_2_reg	$\mathbb{R}^+$	$\left[10^{-4}, 10^{-2}\right]$ (log-scaled)

Table 2: Hyper-parameters to be optimized for `MLP` with `SGD`.

Experimental Design In this section, we focus on neural network-typed classifiers, we will thus be maximizing the likelihood of the training set D_{train} under the model parameterized by θ (in this section, θ corresponds to (W, b) where W is the weight matrix and b is the bias vector):

$$\mathcal{L}(\theta, D_{\text{train}}) = \frac{1}{|D_{\text{train}}|} \sum_{i=1}^{|D_{\text{train}}|} \log(\mathbb{P}(Y = y^{(i)}|x^{(i)}, \theta),$$

which is equivalent to minimize the loss function:

$$\ell(\theta, D_{\text{train}}) = -\mathcal{L}(\theta, D_{\text{train}}).$$

At each time step t, we give one unit of resources (one epoch of training here) to the current algorithm, who will run one epoch of training on the training set D_{train}. The trained model is then used to predict output values $\widehat{y}_{\text{pred},t}$ and $\widetilde{y}_{\text{pred},t}$ respectively over validation set D_{valid} and test set D_{test}. We then compute the number that were misclassified by the model, a.k.a. the zero-one loss on the validation and test set:

$$\ell_t(\widehat{\theta}_t, D_{\text{valid}}) = \frac{1}{|D_{\text{valid}}|} \sum_{i=1}^{|D_{\text{valid}}|} \mathbb{1}_{\{\widehat{y}^{(i)}_{\text{pred},t} \neq y^{(i)}\}},$$

$$\ell_t(\widehat{\theta}_t, D_{\text{test}}) = \frac{1}{|D_{\text{test}}|} \sum_{i=1}^{|D_{\text{test}}|} \mathbb{1}_{\{\widetilde{y}^{(i)}_{\text{pred},t} \neq y^{(i)}\}}.$$

During the experiment, we keep track of the best validation error and its associated test error. At each time step t, if the new validation error is smaller than the current best validation error, then we update the best validation error, and report the new test error. Otherwise we just report the test error associated with the previous best validation error.

Note that there is a very important 'keep training' notion here, which means if we are running the classifier with a same hyper-parameter configuration, then we do not restart the training from scratch. In contrary, we always keep track of previously trained weight

Hyper-parameter	Type	Bounds
learning_rate	$\mathbb{R}^+$	$[10^{-3}, 10^{-1}]$ (log-scaled)
batch_size	$\mathbb{N}^+$	$[1, 1000]$
k_2	$\mathbb{N}^+$	$[10, 60]$
k_1	$\mathbb{N}^+$	$[5, k_2]$

Table 3: Hyper-parameters to be optimized for `CNN` with `SGD`.

matrix and bias vector with respect to the current hyper-parameter configuration, and train the model from these pre-trained parameters.

The total budget for Hyperband and the heuristic would be $B = R \times s_{\max}$, and each configuration can be evaluted for $R_{\max}$ times (this $R_{\max}$ depends only on R and $s_{\max}$). For HCT, we just need to feed the algorithm the total budget B, and the number of times that each configuration is evaluated will be decided by the algorithm itself. While for TPE, HOO and Random Search, we evaluate each configuration for $R_{\max}$ times in order to make a fair comparison, which means $B/R_{\max}$ configurations will be evaluted.

Comparison All plots here (from Fig. 1 to Fig. 3) are averaged on 10 trials of experiments.

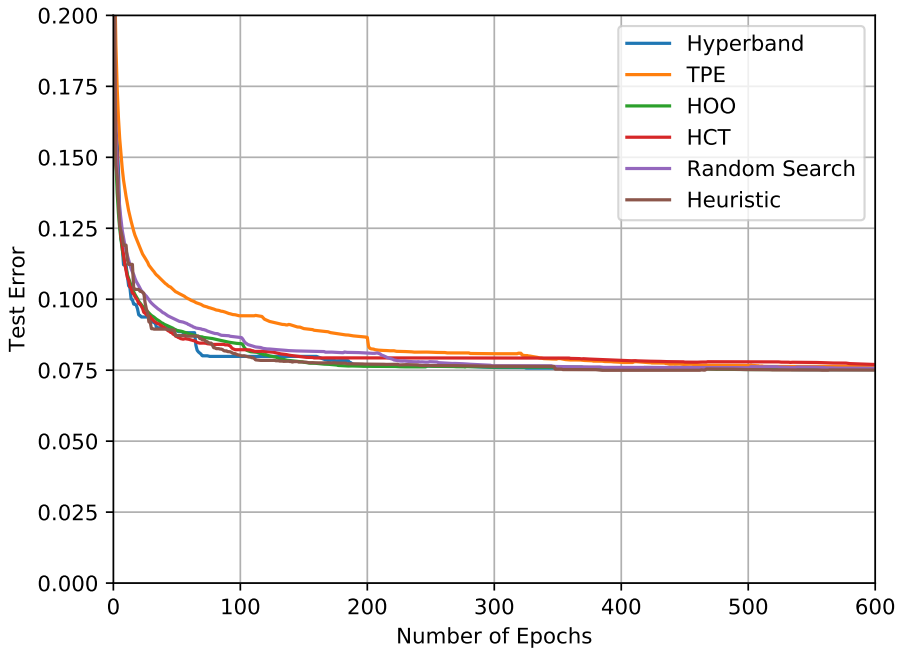

Figure 1: Comparing different hyper-parameter optimization algorithms on Logistic Regression, trained on `MNIST` Dataset.

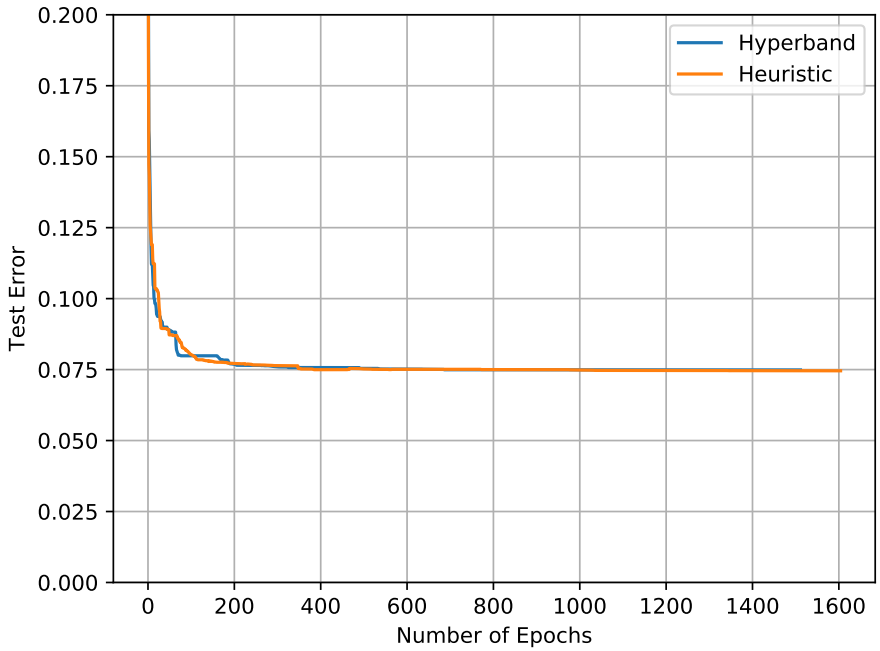

Figure 2: Comparing `Hyperband` and the heuristic on Logistic Regression, trained on `MNIST` Dataset.

Discussion In the current setting, Hyperband seems to be a plausible choice since it may explore more configurations compared to other algorithms. And comparing Hyperband and the heuristic, we can see that they almost performs as well as each other. There are two major observations here. First, the heuristic seems to be converging more smoothly than Hyperband, this is because Hyperband focuses on evaluating one of the configurations after its round-robin tour among all the configurations in the bracket, which leads to a straight drop of the output loss. The second observation is that the heuristic seems to have a slightly better output at the end. This could be due to the fact that`TTTS` is likely to evaluate the possible best configurations more times than Sequential Halving.

4. Hyper-parameter tuning as SIAB problem

We consider here Adaptive Boosting (AdaBoost), Gradient Boosting Machine (GBM), k-Nearest Neighbors (KNN), Multi-Layer Perceptron (`MLP`), Support Vector Machine (SVM), Decision Tree and Random Forest from Scikit-learn.

Dataset Several datasets on UCI dataset archive (e.g. Wine, Breast Cancer, etc) are being used. They are all pre-split into a training set D_{train} and a test set D_{test}.

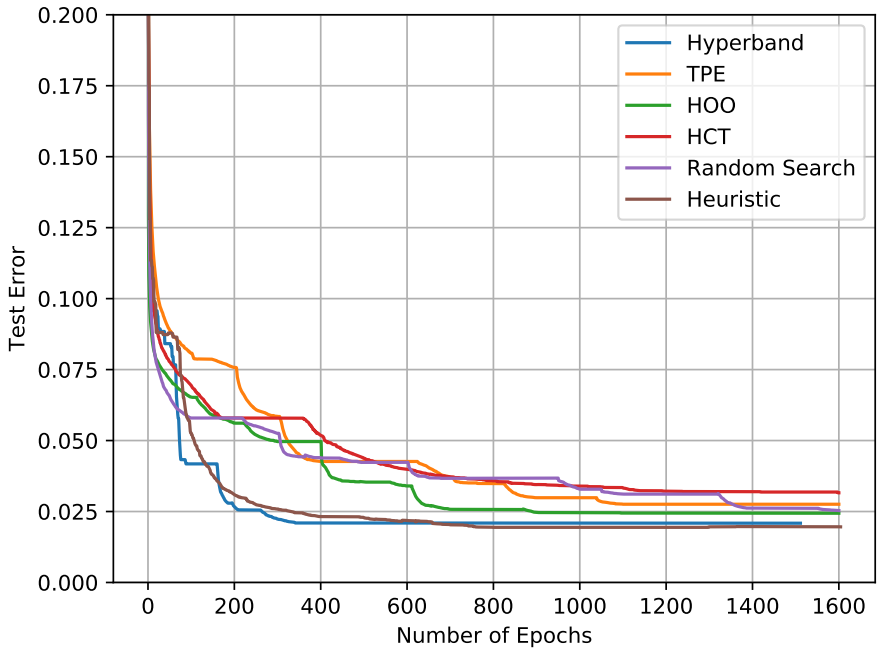

Figure 3: Comparing different hyper-parameter optimization algorithms on `MLP`, trained on `MNIST` Dataset.

Hyper-parameters The hyper-parameters to be optimized are listed below in Table 4, 5, 6, 7, 8, 9 and 10.

Parameter	Type	Bounds
`learning_rate`	$\mathbb{R}^+$	$[10^{-5}, 10^{-1}]$
`n_estimators`	Integer	$\{5, \ldots, 200\}$

Table 4: Hyper-parameters to be optimized for AdaBoost models.

Parameter	Type	Bounds
`learning_rate`	$\mathbb{R}^+$	$[10^{-5}, 10^{-2}]$
`n_estimators`	Integer	$\{10, \ldots, 100\}$
`max_depth`	Integer	$\{2, \ldots, 100\}$
`min_samples_split`	Integer	$\{2, \ldots, 100\}$

Table 5: Hyper-parameters to be optimized for GBM models.

Parameter	Type	Bounds
k	Integer	$\{10, \ldots, 50\}$

Table 6: Hyper-parameters to be optimized for KNN models.

Parameter	Type	Bounds
`hidden_layer_size`	Integer	$[5, 50]$
`alpha`	$\mathbb{R}^+$	$[0, 0.9]$

Table 7: Hyper-parameters to be optimized for `MLP` models.

Resource Allocation One unit of resources in this setting is one iteration of training, which means one complete training of each classifier/regressor over the whole training set.

Experimental Design In this section we use the logarithmic loss, also known as cross-entropy for classification problem, defined by:

$$\ell(\theta, D) = -\frac{1}{|D|} \sum_{i=1}^{|D|} \sum_{j=1}^{m} y_j^{(i)} \log(\widehat{p}_j^{(i)}),$$

where $\widehat{p}_{ij}$ is the predicted probability of a sample i belonging to class j, and m is the number of classes considered. And for regression problems, the loss that we use is the typical mean squared error, defined by:

$$\ell(\theta, D) = -\frac{1}{|D|} \sum_{i=1}^{|D|} \left(y^{(i)} - \widehat{y}_{\text{pred}}^{(i)} \right)^2.$$

And for this part of experiments, we choose to perform a shuffled $k = 5$ cross-validation scheme on D_{train} at each time step t. In practice, this means that we fit 5 models with the same architecture to different train/validation splits and average the loss results in each. More precisely, for every cross-validation split $\texttt{cv}_j, j = 1 \ldots 5$, we get a loss $\ell_{j,t}(\widehat{\theta}_{j,t}, D_{\text{valid},j,t}) = \frac{1}{n} \sum_{i=1}^{n} \left(y_j^{(i)} - \widehat{y}_{\text{pred},j,t}^{(i)} \right)^2$, where $n = |D_{\text{valid}}|$ (here we take MSE as an example, it's the same for log-loss). Thus the validation loss at time t is

$$\frac{1}{5} \sum_{j=1}^{5} \ell_{j,t}(\widehat{\theta}_{j,t}, D_{\text{valid},j,t}) = \frac{1}{5n} \sum_{j=1}^{5} \sum_{i=1}^{n} \left(y_j^{(i)} - \widehat{y}_{\text{pred},j,t}^{(i)} \right)^2.$$

Just like in the previous section, we can then compute and report the test error on the holdout test set D_{test}:

$$\ell_t(\widehat{\theta}_t, D_{\text{test}}) = \frac{1}{|D_{\text{test}}|} \sum_{i=1}^{|D_{\text{test}}|} \left(y^{(i)} - \widetilde{y}_{\text{pred},t}^{(i)} \right)^2.$$

Note that under this experimental environment, 'keep training' does not make sense anymore. Thus for HOO, TPE and Random Search, we only need to evaluate each configuration once, contrarily to what we did in the previous setting. While for Hyperband, we still need to evaluate each configuration for a certain times based on R and $s_{\max}$.

Parameter	Type	Bounds
C	$\mathbb{R}^+$	$[10^{-5}, 10^5]$ (log-scaled)
γ	$\mathbb{R}^+$	$[10^{-5}, 10^5]$ (log-scaled)

Table 8: Hyper-parameters to be optimized for SVM models.

Parameter	Type	Bounds
`max_features`	$\mathbb{R}^+$	$[0.01, 0.99]$
`max_depth`	Integer	$\{4, \ldots, 30\}$
`min_samples_split`	$\mathbb{R}^+$	$[0.01, 0.99]$

Table 9: Hyper-parameters to be optimized for Decision Tree models.

Comparison Each plot here is averaged on 20 runs of experiments. Fig. 4 to 8 display the results for Wine Dataset.

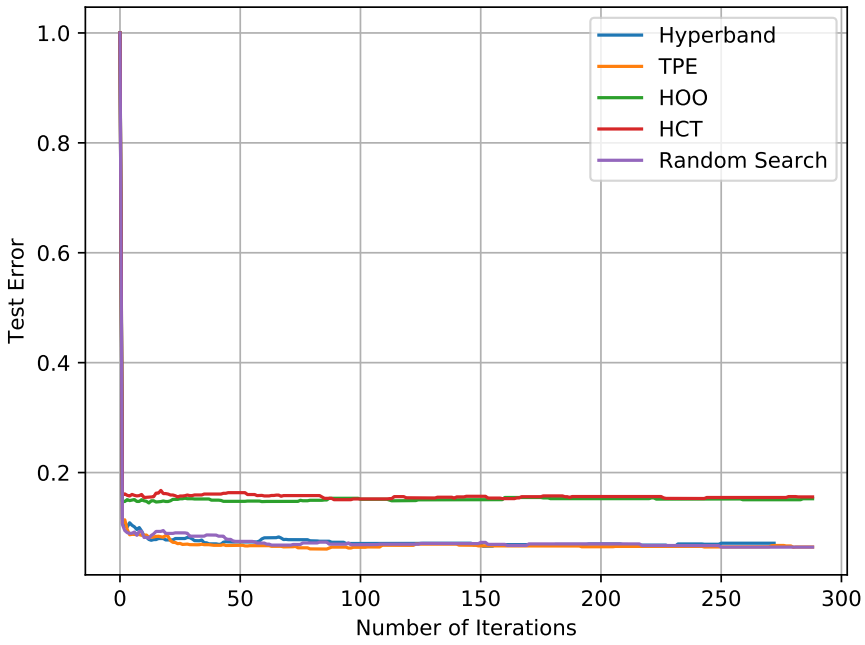

Figure 4: Performance comparison of different hyper-parameter optimization algorithms on AdaBoost, trained on Wine Dataset.

Fig. 9 to 13 display the results for Breast Cancer Dataset.

Parameter	Type	Bounds
`max_features`	$\mathbb{R}^+$	$[0.1, 0.5]$
`n_estimators`	Integer	$\{10, \ldots, 50\}$
`min_samples_split`	$\mathbb{R}^+$	$[0.1, 0.5]$

Table 10: Hyper-parameters to be optimized for Random Forest models.

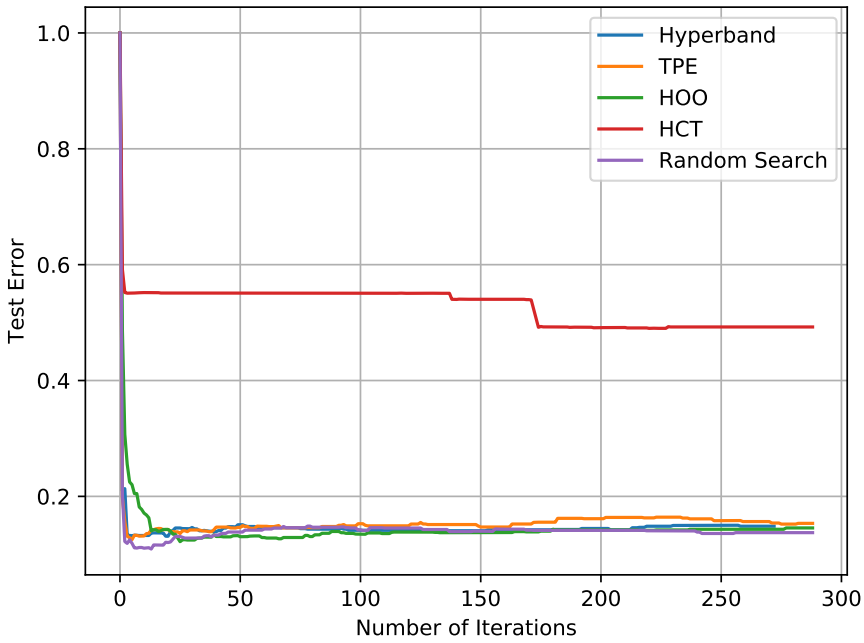

Figure 5: Performance comparison of different hyper-parameter optimization algorithms on GBM, trained on Wine Dataset.

Discussion In this setting, Hyperband seems to loose its advantage of exploring more configurations. It appears to me that there is no reason for HOO, TPE and Random Search to evaluate one point several times as does Hyperband.

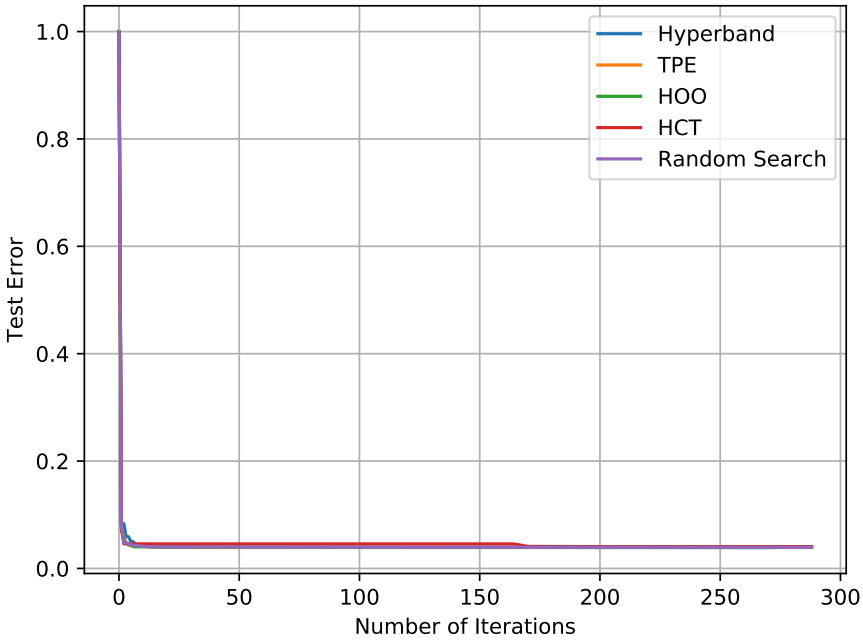

Figure 6: Performance comparison of different hyper-parameter optimization algorithms on KNN, trained on Wine Dataset.

References

Mohammad Gheshlaghi Azar, Alessandro Lazaric, and Emma Brunskill. Online stochastic optimization under correlated bandit feedback. In *Proceedings of the 31st International Conference on Machine Learning (ICML)*, pages 1557–1565, 2014.

James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13:281–305, 2012.

James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. In *Advances in Neural Information Processing Systems 24 (NIPS)*, pages 2546–2554, 2011.

Eric Brochu, Vlad M. Cora, and Nando de Freitas. A tutorial on Bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. 2010.

Sébastien Bubeck, Rémi Munos, Gilles Stoltz, and Csaba Szepesvari. X-armed bandits. *Journal of Machine Learning Research*, 12:1587–1627, 2010.

Chih-Chun Chang and Chih-Jen Lin. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 2(3):1–27, 2011.

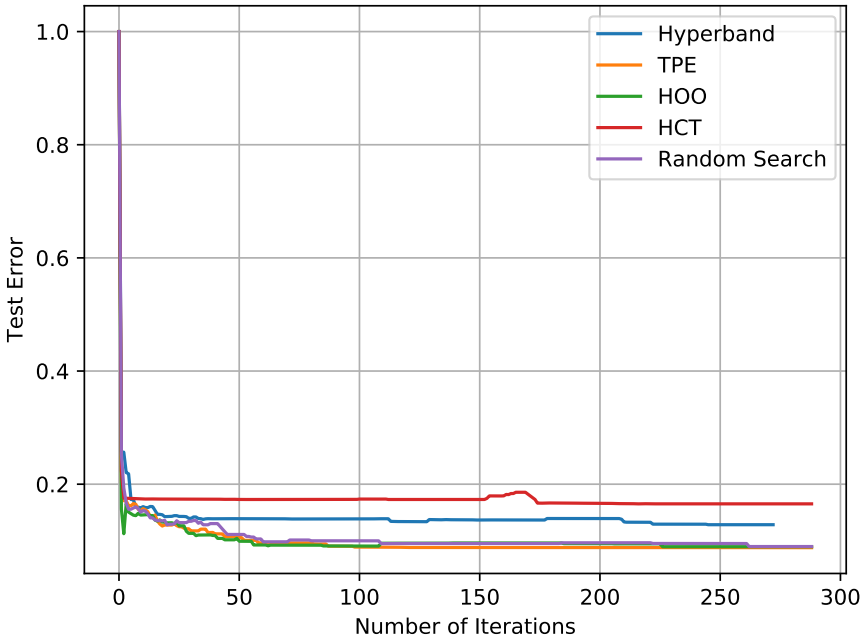

Figure 7: Performance comparison of different hyper-parameter optimization algorithms on `MLP`, trained on Wine Dataset.

Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *Proceedings of the 5th International Conference on Learning and Intelligent Optimization (LION)*, pages 507–523, 2011.

Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. Technical report, 2016.

Jonas Mockus, Vytautas Tiešis, and Antanas Žilinskas. The application of Bayesian methods for seeking the extremum. *Towards Global Optimisation 2*, pages 117–129, 1978.

Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Edouard Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

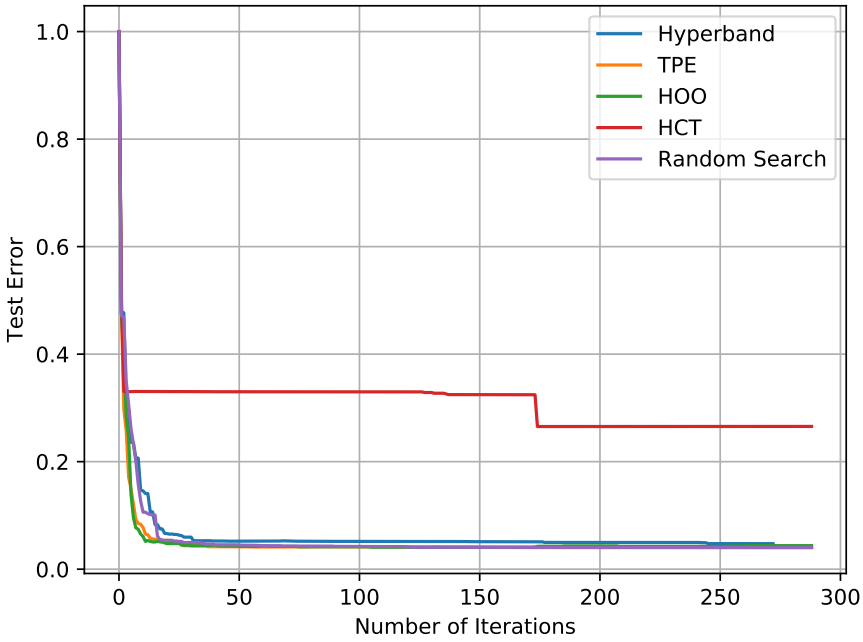

Figure 8: Performance comparison of different hyper-parameter optimization algorithms on SVM, trained on Wine Dataset.

Daniel Russo. Simple Bayesian algorithms for best arm identification. In *Proceeding of the 29th Conference on Learning Theory (CoLT)*, 2016.

Bobak Shahriari, Kevin Swersky, Ziyu Wang, Ryan P. Adams, and Nando de Freitas. Taking the human out of the loop: A review of Bayesian optimization. *Proceedings of the IEEE*, 104(1):148–175, 2016.

Jasper Snoek, Hugo Larochelle, and Ryan P. Adams. Practical bayesian optimization of machine learning algorithms. In *Advances in Neural Information Processing Systems 25 (NIPS)*, pages 2951–2959, 2012.

Jasper Snoek, Oren Rippel, Kevin Swersky, Ryan Kiros, Nadathur Satish, Narayanan Sundaram, Mostofa A. Patwary, Prabhat, and Ryan P. Adams. Scalable Bayesian optimization using deep neural networks. In *Proceedings of the 32nd International conference on Machine Learning (ICML)*, 2015.

Niranjan Srinivas, Andreas Krause, Sham M. Kakade, and Matthias Seeger. Gaussian process optimization in the bandit setting: No regret and experimental design. In *Proceedings of the 27th International conference on Machine Learning (ICML)*, pages 1015–1022, 2010.

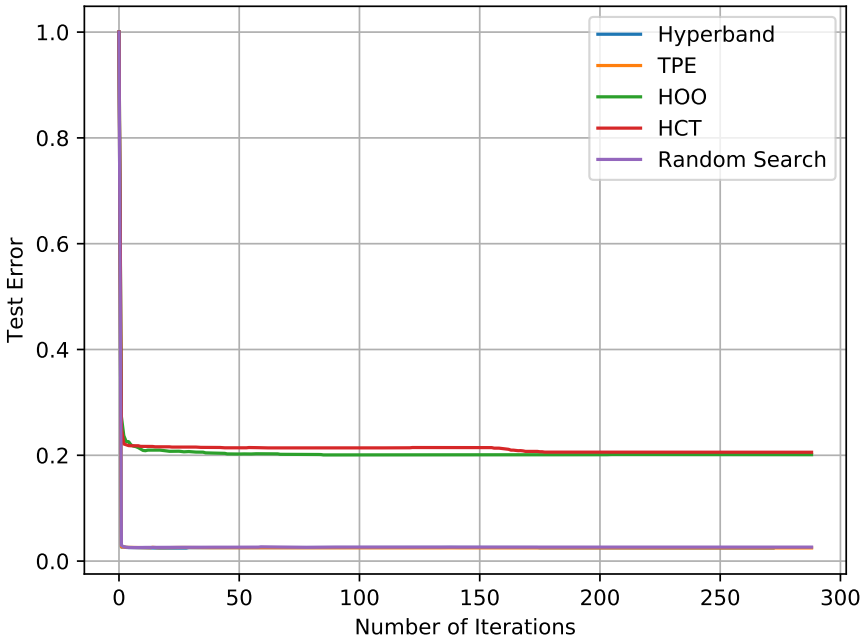

Figure 9: Performance comparison of different hyper-parameter optimization algorithms on AdaBoost, trained on Breast Cancer Dataset.

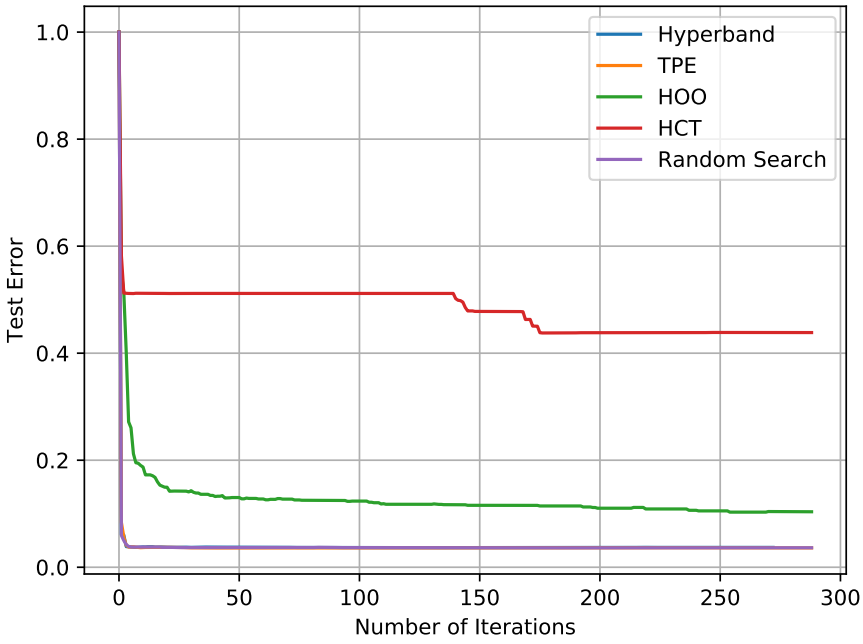

Figure 10: Performance comparison of different hyper-parameter optimization algorithms on GBM, trained on Breast Cancer Dataset.

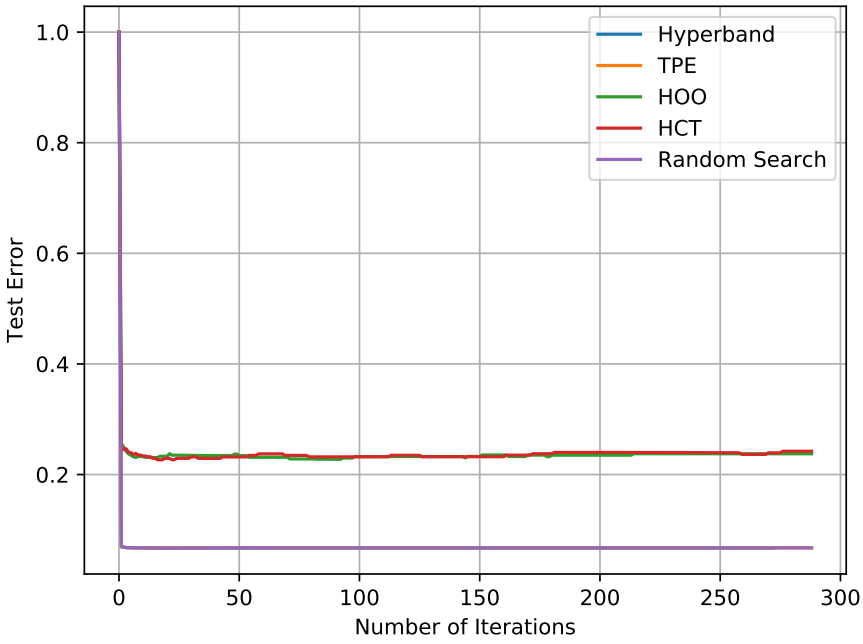

Figure 11: Performance comparison of different hyper-parameter optimization algorithms on KNN, trained on Breast Cancer Dataset.

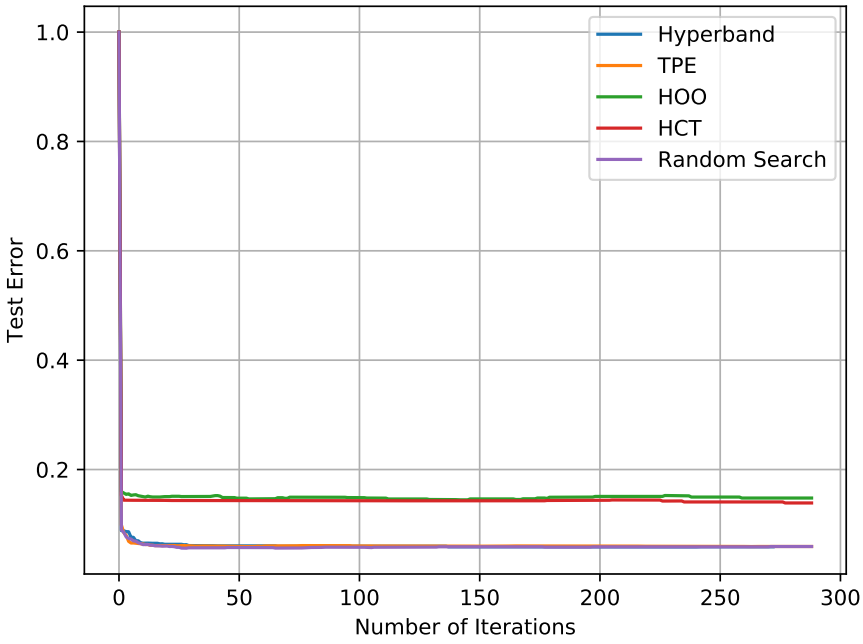

Figure 12: Performance comparison of different hyper-parameter optimization algorithms on `MLP`, trained on Breast Cancer Dataset.

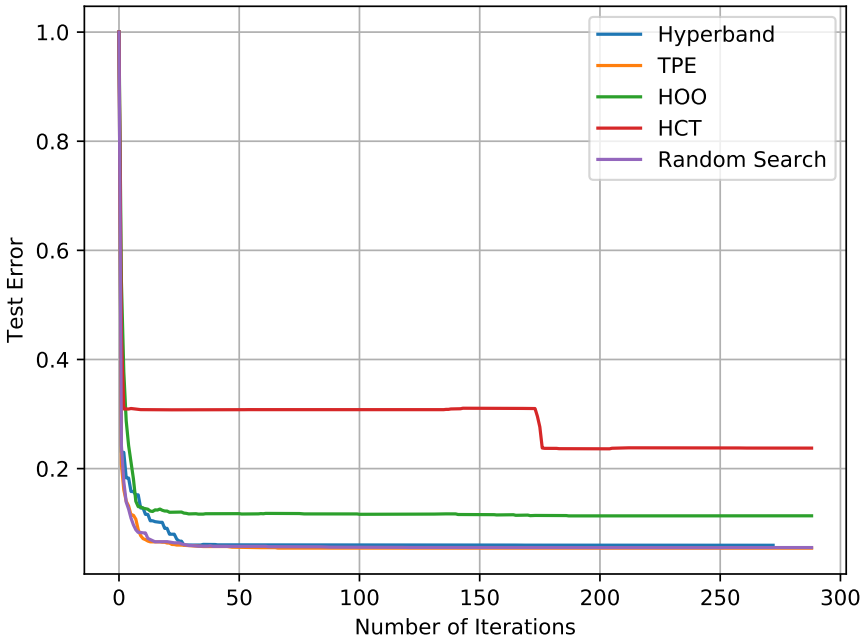

Figure 13: Performance comparison of different hyper-parameter optimization algorithms on SVM, trained on Breast Cancer Dataset.