IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

SPECIAL SECTION ON SCALABLE DEEP LEARNING FOR BIG DATA

Received January 1, 2020, accepted March 5, 2020, date of publication March 16, 2020, date of current version March 25, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.2981072

Basic Enhancement Strategies When Using Bayesian Optimization for Hyperparameter Tuning of Deep Neural Networks

HYUNGHUN CHO[1], YONGJIN KIM[2], EUNJUNG LEE[1], DAEYOUNG CHOI[1], YONGJAE LEE[3], AND WONJONG RHEE[1], (Fellow, IEEE)

[1]Department of Transdisciplinary Studies, Seoul National University, Seoul 08826, South Korea
[2]College of Liberal Studies, Seoul National University, Seoul 08826, South Korea
[3]School of Management Engineering, UNIST, Ulsan 44919, South Korea

Corresponding author: Wonjong Rhee (wrhee@snu.ac.kr)

This work was supported in part by the National Research Foundation of Korea (NRF) Grant funded by the Korean Government (MSIT) under Grant NRF-2017R1E1A1A03070560, and in part by the Electronics and Telecommunications Research Institute (ETRI) Grant funded by the Korean Government (Distributed Intelligence Core Technology of Hyper-Connected Space) under Grant 19ZH1100.

ABSTRACT Compared to the traditional machine learning models, deep neural networks (DNN) are known to be highly sensitive to the choice of hyperparameters. While the required time and effort for manual tuning has been rapidly decreasing for the well developed and commonly used DNN architectures, undoubtedly DNN hyperparameter optimization will continue to be a major burden whenever a new DNN architecture needs to be designed, a new task needs to be solved, a new dataset needs to be addressed, or an existing DNN needs to be improved further. For hyperparameter optimization of general machine learning problems, numerous automated solutions have been developed where some of the most popular solutions are based on Bayesian Optimization (BO). In this work, we analyze four fundamental strategies for enhancing BO when it is used for DNN hyperparameter optimization. Specifically, diversification, early termination, parallelization, and cost function transformation are investigated. Based on the analysis, we provide a simple yet robust algorithm for DNN hyperparameter optimization - DEEP-BO (Diversified, Early-termination-Enabled, and Parallel Bayesian Optimization). When evaluated over six DNN benchmarks, DEEP-BO mostly outperformed well-known solutions including GP-Hedge, BOHB, and the speed-up variants that use Median Stopping Rule or Learning Curve Extrapolation. In fact, DEEP-BO consistently provided the top, or at least close to the top, performance over all the benchmark types that we have tested. This indicates that DEEP-BO is a robust solution compared to the existing solutions. The DEEP-BO code is publicly available at https://github.com/snu-adsl/DEEP-BO.

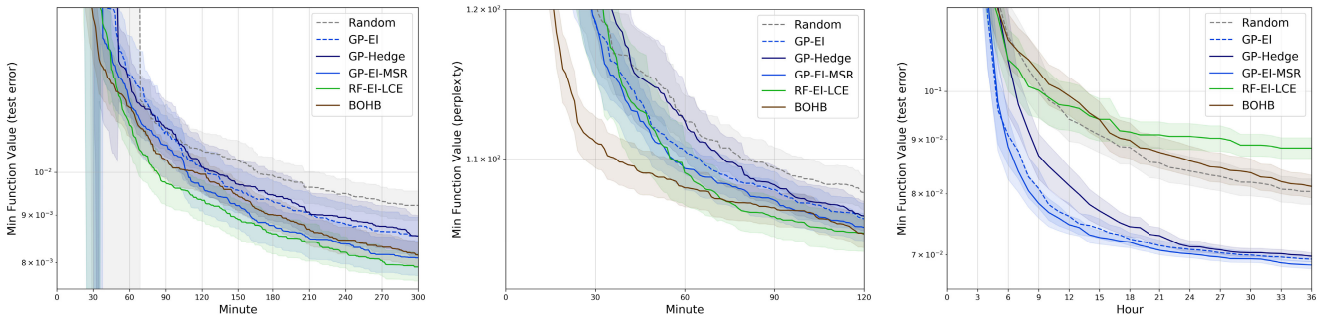
INDEX TERMS Deep neural networks, hyperparameter optimization, Bayesian optimization, diversification, early termination, parallelization, cost function transformation.

I. INTRODUCTION

Hyperparameter Optimization (HPO) aims to find the global optimum $\mathbf{x}^*$ of an unknown black-box function f where $f(\mathbf{x})$ can be evaluated for any arbitrary $\mathbf{x} \in \mathcal{X}$. That is, $\mathbf{x}^* = \arg\max_{\mathbf{x}\in\mathcal{X}} f(\mathbf{x})$, where $\mathcal{X}$ is a hyperparameter space that can contain categorical, discrete, and continuous variables. To automatically solve HPO problems, a variety of approaches have been adopted. Among them, random search [1] and Bayesian Optimization (BO) [2] are the most basic and popular solutions, and they have been intensively studied mainly for traditional Machine Learning (ML) problems [3], [4], [20].

Recently, HPO of Deep Neural Network (DNN) has emerged as an important topic. Compared to the traditional machine learning, deep learning with human tuning has dramatically reduced the required level of human effort by shifting the central work from feature engineering based on domain knowledge to DNN architecture design and tuning based on data and task types. DNN performance, however, is known to be highly sensitive to the hyperparameter setting, and deep learning researchers often spend long hours trying

The associate editor coordinating the review of this manuscript and approving it for publication was Liangxiu Han.



(a) Digit classification (b) Language modeling (c) Image classification

FIGURE 1. HPO performance comparison for three different DNN tasks (the lower, the better): (a) LeNet model on MNIST (b) LSTM model on PTB (c) ResNet model on CIFAR-10. Each plot compares the minimum function value of six HPO algorithms as the optimization time t increases. The shaded areas show 0.25σ, where the variance is σ^2. Clearly, none of the algorithms universally perform well over the three tasks, and the best performance is achieved by RF-EI-LCE, BOHB, and GP-EI-MSR, respectively.

to tune the hyperparameters. Therefore, it would be natural to use an automatic procedure for configuration tuning. Deep learning with HPO is capable of further reducing the required level of human effort by shifting the essential work from DNN architecture design and tuning to the selection of base architecture and $\mathcal{X}$. It is important to recognize that the use of HPO does not imply a complete elimination of human effort at the cost of computation power, but rather it means yet another level of reduction in the required level of human effort.

There are numerous examples of improving DNN performance with HPO. For instance, it was reported in [4] that their HPO outperformed human experts by tuning seven hyperparameters of a CNN model. Deep learning researchers, however, are often reluctant to use such an automated procedure for a few reasons. A well-known reason is the utilization of domain knowledge and experience. Researchers typically prefer to perform manual tuning and utilize one's knowledge for deciding what to try next. Most of the existing automated procedures are not capable of utilizing the previously learned knowledge, and there is a significant on-going effort for absorbing such previously learned knowledge into the hyperparameter optimization framework. Some of the recent works are [6], [7] and [8] where transfer learning and meta learning are considered. While this subject is an important topic, it is not in the scope of this work. Another reason is the possibility of a disastrous failure. When running an automated procedure on DNN, even a single sequence of hyperparameter optimization can take hours or days before an exit condition is met. Therefore, it is important to minimize the risk of a disastrous failure and to improve the worst-case performance (or equivalently the chance of a successful hyperparameter optimization). This is an important topic to address in the field of DNN hyperparameter optimization, and the main subject of this work.

While many new HPO algorithms have been introduced recently, none seems to be universally effective over DNN benchmark problems. In Fig. 1, we show performance for three different DNN benchmark tasks using the following well-known HPO algorithms: random search [1], Gaussian Process (GP-EI) [4], an adaptive GP algorithm GP-Hedge [9], two speed-up BOs using Median Stopping Rule (GP-EI-MSR) [10] and Learning Curve Extrapolation (RF-EI-LCE) [11], and a hybrid bandit based approach BOHB [12]. The minimum function values are considered to be the standard and conventional metric in the hyperparameter optimization literature. The results show that an HPO algorithm's relative performance can be highly dependent on the specific choice of task, and none of the algorithms robustly work well over the three tasks. In this regard, no free lunch theorem in [13] provides insightful proof that there exists no optimization algorithm that universally performs best. Also, [14] shows a similar result with practical ML benchmarks.

In addition to the minimum function value that is used in Fig. 1, we also use two new metrics for evaluating HPO algorithms. Because it is important to design an HPO algorithm that can achieve a target goal within the given time budget of t, *success rate* is defined as the first metric. Assuming that an HPO algorithm takes time τ, a random variable, to achieve a target performance (e.g. accuracy) c, the algorithm's *success rate* at time t can be defined as successfully finding a $\hat{\mathbf{x}}$ such that $f(\hat{\mathbf{x}}) > c$ before reaching time t. It can also be simply expressed as $\mathbf{P}(\tau \leq t)$. An alternative metric is *expected time* to achieve a target performance (e.g. accuracy), and it can be expressed as $\mathbb{E}[\tau]$. While both are not the standard metrics in the literature yet, they provide easily interpretable values because they are based on a fixed-target approach [15].

Our goal is to empirically identify the basic BO strategies that work well specifically for DNN tuning. HPO of DNN has important and distinct aspects when compared to the traditional ML. Firstly, DNN's architecture design can significantly affect the performance and it has many hyperparameters that can be considered for the design. For instance, the number of neurons in each layer can be a hyperparameter.

Therefore, DNN problems tend to have a much larger $\mathcal{X}$ than the traditional ML problems. We will address this issue further in Section III. Secondly, the performance curve from the first epoch to the last epoch of DNN training can be drastically affected as new techniques are introduced. For instance, the introduction of batch normalization [16] has made it more difficult to use an early epoch performance to predict a later epoch performance. We will address this issue further in Section V. With these aspects in mind, we focus on automated procedures that are based on BO and show how basic enhancement strategies can be used altogether to improve the chance of a successful hyperparameter optimization. To be specific, we mainly focus on developing useful insights on diversification, early termination, parallelization, and cost function transformation. Then, we propose a simple yet robust BO algorithm based on the basic strategies. The algorithm is called DEEP-BO (*Diversified, Early-termination Enabled, Parallel Bayesian Optimization*), and we empirically show that DEEP-BO can consistently and robustly achieve relatively high performance over six representative DNN benchmarks.

II. BAYESIAN OPTIMIZATION

Bayesian Optimization (BO) is a sequential approach for parameter optimization of any black-box function $f(\mathbf{x})$. BO was shown to be useful for algorithm configuration first in [17], and now it has become a popular state-of-the-art solution. Introduction to BO can be found, for instance, in [18] and [2]. BO incorporates prior belief for estimating a response surface function $\hat{f}(\mathbf{x})$, uses $\hat{f}(\mathbf{x})$ for selecting the next configuration $\mathbf{x}_n$ to try, evaluates $f(\mathbf{x}_n)$ using the true black-box function, calculates posterior belief using the evaluated performance $f(\mathbf{x}_n)$, and repeats the process in sequence until a stopping criterion is met. A pseudocode of BO is shown in Algorithm 1.

Algorithm 1 Bayesian Optimization

Inputs: black-box function f, BO algorithm $\hat{f}$, parameter space $\mathcal{X}$
$\mathcal{H} \leftarrow \emptyset$
for $n = 1, 2, \ldots$ **do**
  Select $\mathbf{x}_n \in \arg\max_{\mathbf{x}\in\mathcal{X}} \hat{f}(\mathbf{x}; \mathcal{H})$
  Evaluate $y_n \leftarrow f(\mathbf{x}_n)$
  Update $\mathcal{H} \leftarrow \mathcal{H} \cup (\mathbf{x}_n, y_n)$
  Check for exit criteria
end for

For building the response surface of $\hat{f}(\mathbf{x})$, the three most popular choices for ML problems are Tree-structured Parzen Estimator, Gaussian Process [19], and Random Forest regressor. They were used in [3], [4], [20], respectively. The algorithms utilize acquisition functions that can provide a tradeoff between *exploration* and *exploitation*, and the three most popular choices for the acquisition functions are Probability of Improvement (PI) [21], Expected Improvement (EI) [17], and Upper Confidence Bound (UCB) [22].

In our work, the black-box function $f(\mathbf{x})$, which represents the performance (usually prediction accuracy or error) of a DNN with a model configuration $\mathbf{x}$, is highly non-convex. Also, $f(\mathbf{x})$ can be evaluated at an arbitrary point $\mathbf{x}$, but even a single evaluation can take a considerable amount of time because the evaluation of $f(\mathbf{x})$ requires performing the entire training procedure of the DNN.

III. SIX DNN BENCHAMARKS AND PRE-EVALUATION

For evaluating HPO algorithms, a variety of tasks have been used as benchmarks. For the ML domain, [14] collected twelve traditional and popular benchmarks and created a library. Modern DNN benchmarks were also considered in a few works including [4] and [23]. The DNN benchmarks, however, were often limited in one way or another. Some benchmarks had only four hyperparameters (which can be considered to be too small for fully tuning a modern DNN), some benchmarks excluded categorical hyperparameters, and some benchmarks used old architectures such as Deep Belief Network (DBN).

A. SIX DNN BENCHMARKS

To overcome the limitations, we have created six DNN benchmarks using currently popular deep learning datasets (MNIST, PTB (Penn Treebank), CIFAR-10, and CIFAR-100) and commonly used DNN architectures (simple CNN, LeNet [24], ResNet [25], and LSTM [26]). The six benchmarks are summarized in Table 1.

TABLE 1. Six DNN benchmarks.

Name	Number of hyperparameters			Max epoch (E)
	discrete	continuous	categorical	
MNIST-LeNet1	6	3	-	15
MNIST-LeNet2	3	3	3	15
PTB-LSTM	3	5	1	15
CIFAR10-CNN	6	2	2	50
CIFAR10-ResNet	2	4	1	100
CIFAR100-CNN	6	2	2	50

Practically, the number, type, and range of hyperparameters are important factors, too. Except for the CIFAR10-ResNet benchmark, ten hyperparameters were chosen such that a sufficiently large $\mathcal{X}$ could be used as the search space. For CIFAR10-ResNet, we had to limit the number of hyperparameters to seven because the training time was an order of magnitude larger than the other benchmarks. The hyperparameters considered were related to architecture, optimization, and regularization, and all of discrete, continuous, and categorical types were included. We note that the learning rate, that is most important for tuning DNN, is included in all the benchmarks and its range and sampling scheme were carefully designed. We discuss this further in Section XI. The details of the six benchmarks can be found in Table 2–7. Even though not included in this work, we have also experimented

TABLE 2. MNIST-LeNet1 benchmark.

Hyperparameter	Range
number of first convolution kernels	[1, 350]
first pooling size	[2, 3]
number of second convolution kernels	[1, 350]
second pooling size	[2, 3]
number of neurons in fully connected layer	[1, 1024]
square length of the convolution kernel	[2, 10]
learning rate	[0.0001, 0.4] (log scale)
L2 regularization factor	[0.0, 1.0]
dropout rate	[0.0, 0.9]

TABLE 3. MNIST-LeNet2 benchmark.

Hyperparameter	Range
number of first convolution kernels	[1, 350]
number of second convolution kernels	[1, 350]
number of neurons in a fully connected layer	[1, 1024]
learning rate	[0.0001, 0.4] (log scale)
L2 regularization factor	[0.0, 1.0]
dropout rate	[0.0, 1.0]
activation function type	ReLU, tanh,sigmoid, eLU, leaky ReLU
optimizer type	AdaDelta, AdaGrad, Adam, GD, Momentum, RMSProp
batch normalization	Enable, Disable

TABLE 4. PTB-LSTM benchmark.

Hyperparameter	Range
number of neurons in a hidden layer	[10, 200]
number of hidden layers	[1, 2]
number of training steps	[10, 20]
initial value of uniform random scale	[0.01, 0.1]
dropout rate	[0.0, 0.9]
learning rate	[0.1, 1.0]
learning rate decay	[0.5, 1.0]
gradient clipping by global normalization	[5.0, 10.0]
RNN training module	Basic, Block, cuDNN

TABLE 5. CIFAR10-CNN benchmark.

Hyperparameter	Range
number of first convolution kernels	[8, 32]
number of second convolution kernels	[32, 64]
number of third convolution kernels	[64, 128]
number of forth convolution kernels	[64, 128]
number of neurons in a fully connected layer	[10, 1000]
square length of convolution kernel	[2, 3]
learning rate	[0.0001, 0.4] (log scale)
L2 regularization factor	[0.0, 1.0]
activation function type	ReLU, tanh, eLU
regularization method	None, Dropout, BatchNorm

with smaller and larger numbers of hyperparameters (four to seventeen) and the findings were similar.

B. PRE-EVALUATED CONFIGURATIONS

The performance of an HPO algorithm for a specific task is not deterministic. That is, an HPO algorithm would output a different result for each trial of hyperparameter optimization. Therefore, repeated experiments are necessary to estimate the *success rate* $\mathbf{P}(\tau \leq t)$ and *expected time* $\mathbb{E}[\tau]$ in a credible way. However, each trial of a single HPO run requires a considerable number of DNN training (typically a few hundreds of DNN training), and thus it would be extremely time-consuming to repeat the experiments over multiple algorithms. In this regard, [27], [28] suggest that *surrogate benchmarks* are cheaper to evaluate and closely follow real-world benchmarks.

TABLE 6. CIFAR10-ResNet benchmark.

Hyperparameter	Range
number of layers	38, 44, 50, 56, 62, 70, 78
training data batch size	45, 90, 180, 360, 450
data augmentation	True, False
learning rate	[0.0001, 0.1] (log scale)
momentum	[0.1, 0.9]
weight decay	[0.00001, 0.001] (log scale)
batch normalization decay	[0.9, 0.999]

TABLE 7. CIFAR100-CNN benchmark.

Hyperparameter	Range
number of first convolution kernels	[8, 32]
number of second convolution kernels	[32, 64]
number of third convolution kernels	[64, 128]
number of forth convolution kernels	[64, 128]
number of neurons in a fully connected layer	[10, 1000]
square length of convolution kernel	[2, 3]
learning rate	[0.0001, 0.4] (log scale)
L2 regularization factor	[0.0, 1.0]
activation function type	ReLU, tanh,sigmoid, eLU
regularization method	None, Dropout, BatchNorm

To utilize surrogate benchmarks, we pre-selected a representative set of model configurations, pre-evaluated all the selected configurations, and saved their DNN training results (including the evaluated test performance and the consumed time) into a database. This process takes a considerable amount of time to complete, because we need to complete DNN training of all the configurations. Therefore we carefully chose the number of maximum epoch for each benchmark, where it was chosen to be sufficiently large but not unnecessarily large. We had 20,000 configurations for five benchmarks and 7,000 for CIFAR10-ResNet, and thus a total of 107,000 DNN training had to be completed for the six benchmarks. We utilized eight computing machines that have multiple NVIDIA GPUs. The information on training time can be found in Table 8. Though the performance of PTB-LSTM is originally provided as perplexity, the values were transformed to be in the same range as accuracy. Because a lower perplexity means a better performance,

TABLE 8. Statistics of pre-evaluated configurations.

Benchmark	MNIST-LeNet1	MNIST-LeNet2	PTB-LSTM	CIFAR10-CNN	CIFAR10-ResNet	CIFAR100-CNN
Number of configurations	20,000	20,000	20,000	20,000	7,000	20,000
Total evaluation time (day)	88.1	55.5	87.3	75.2	385	82.1
Mean evaluation time (minute)	6.3	4	6.3	5.4	79	5.9
Stdev. evaluation time (minute)	3.6	2.1	2.9	1.0	19	1.6
Best accuracy	0.9939	0.9941	0.9008	0.8052	0.9368	0.5067
Difficult target accuracy	0.9933	0.9922	0.8993	0.7836	0.9333	0.4596
Easy target accuracy	0.9903	0.9861	0.8935	0.6185	0.9092	0.0350

we transformed the value of perplexity as below.

$$y = 1.0 - perplexity/1000.0 \tag{1}$$

In the tables, the single best accuracy is shown together with two carefully chosen accuracy targets - *difficult target accuracy* and *easy target accuracy*. When evaluating HPO algorithms, we repeated the experiments over the two targets such that we can understand the algorithm behavior for both easy and difficult accuracy targets. The difficult target was chosen as the 10th best performance from the 20,000 pre-evaluated configurations, and the easy target was chosen as the 500th best performance from the 20,000 pre-evaluated configurations (7,000 for CIFAR10-ResNet). Figure 2 shows the distribution of the pre-evaluated test accuracy values sorted from the best to the worst.

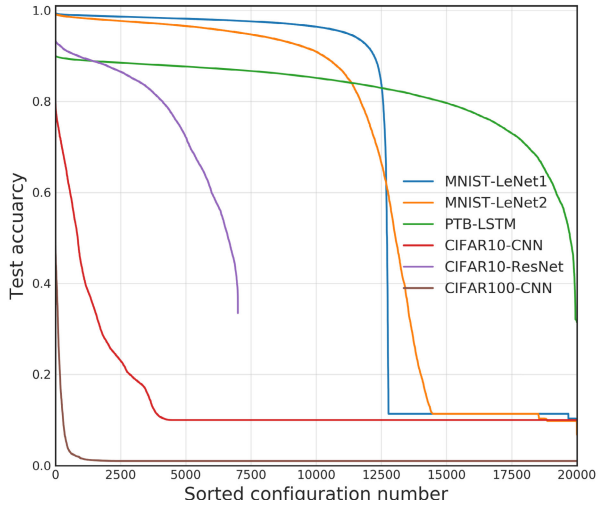


FIGURE 2. Test accuracy distribution of the pre-evaluated configurations. Note that the performance of PTB-LSTM was scaled to match the range of accuracy. Performance was sorted before plotting. The leftmost configuration number corresponds to the largest test accuracy.

After completing the pre-evaluations, we were able to repeatedly perform HPO experiments using the database without requiring actual DNN training. With the pre-generated database, proper book-keeping was performed to expedite the experiments. Using this methodology, we were able to evaluate each HPO algorithm **100 times** for each benchmark task. All the performance results presented in this work were created by repeating each task 100 times.

For selecting '*a representative set of configurations*' from the given $\mathcal{X}$ (i.e. 20,000 or 7,000 configurations from $\mathcal{X}$), we have used the Sobol sequence [29]. Sobol is a quasi-random low-discrepancy sequence in a unit hypercube, and its samples are known to be more evenly distributed in a given space than uniform random sampling.

IV. DIVERSIFICATION STRATEGY

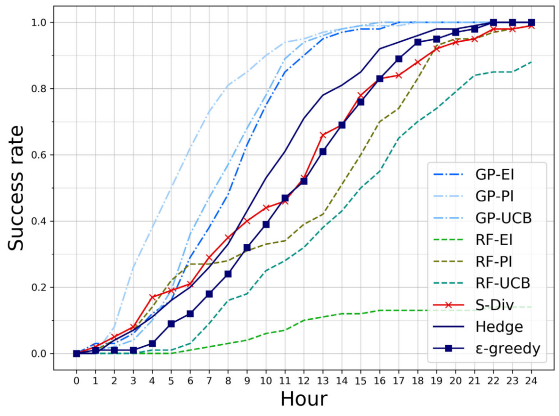
Diversity is a widely used concept in the algorithm studies, and ensemble might be the most relevant diversification example for this work. Ensemble is a popular diversification technique in machine learning [30]. Ensemble employs a set of learned models instead of using only a single model, and the inferences from the individual models are combined to generate the overall inference. While a homogeneous ensemble produces all the individual models using a single type of base learner algorithm (e.g., random forest), a heterogeneous ensemble uses many different types of learner algorithms (e.g., stacking). In both cases, the diversity among the individual models can greatly contribute to the performance improvement.

Typically, a single model is repeatedly used over the entire sequence of HPO. Such a single model HPO is inherently vulnerable because the chosen HPO algorithm might work well for one task but then perform very poorly for another as was demonstrated in Fig. 1. This vulnerability can be a serious limitation for the hyperparameter tuning of deep neural networks because we cannot afford to try many models one by one where each trial can take days or even months. As an essential strategy to overcome this limitation, we propose to adopt diversification at the model level just as in the ensemble. In this section, we study simple and basic implementations of diversification using BO models. Thanks to the inherently sequential nature of BO, we can simply use many different models in turn. In addition to the simple strategy, adaptive strategies are studied, too.

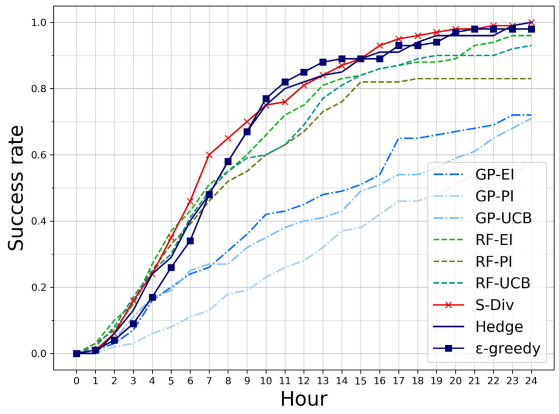
A. SIMPLE DIVERSIFICATION

While a typical BO uses a single model $\hat{f}$ to approximate the true black-box function f, diversification makes use of N different models $(\hat{f}_1, \ldots, \hat{f}_N)$. The N models can be chosen in many different ways, for instance, in a homogeneous way (all are based on a single base model but with minor variations) or in a heterogeneous way. In fact, any state-of-the-art HPO models can be used as the individual models. In this study, however, we restrict our focus to the most popular BO models. Once the N algorithms are chosen, we can repeatedly rotate over the N algorithms. We name the sequential diversification strategy as *S-Div*. Obviously, there are many different ways to implement simple diversification other than S-Div. For instance, it is also possible to randomly choose one of the N models in each turn. When evaluated over the benchmarks, the randomization strategy performed

comparably well as S-Div. Nonetheless, we have chosen to limit our study to S-Div because it makes fair and deterministic use of N individual models and thus, elimination variance from random selection.



(a) PTB-LSTM



(b) MNIST-LeNet2

FIGURE 3. Effects of diversification on two DNN benchmarks. Success rate P($\tau \leq t$) for difficult target are shown for (a) PTB-LSTM and (b) MNIST-LeNet2. Six individual BO models, S-Div, Hedge, and ϵ-greedy are shown. The plots were generated by repeating HPO 100 times for each model.

The effects of model diversification are shown in Fig. 3. For discussing the benefits of simple diversification, we focus on six individual models and S-Div only. Hedge and ϵ-greedy will be addressed in the next subsection. $N = 6$ individual models were used for S-Div (GP-EI, GP-PI, GP-UCB, RF-EI, RF-PI, and RF-UCB), and the single model performance is also shown for each of the six individual models. In Fig. 3a (PTB-LSTM benchmark), it can be seen that the three GP models perform much better than the three RF models. In particular, the performance of RF-UCB turns out to be disastrous. While the performances of individual models show large variations, S-Div was able to avoid the worst cases simply by adopting diversification. Typically, a disastrous case occurs when a model fails to predict promising hyperparameter spaces for a long time. S-Div is able to escape from this problem as long as one of the individual models predict well, and hence it is important to combine individual models that can predict well for a variety of deep neural network tasks. In Fig. 3b (MNIST-LeNet2 benchmark), now the three RF models perform much better than the three GP models, and the previously disastrous RF-EI is the best performing single model for this task. For this benchmark, S-Div was able to perform better than the GP models as expected. In fact, S-Div was able to do much more than avoiding the poor performance, and it actually performed even better than the best performing single model. For this benchmark, each better performing single model not only made sure to choose a promising hyperparameter candidate $\mathbf{x}_n$ during its turn, but it also constructed a better dataset $\mathcal{H}$ to use for poorly performing models. When a good dataset is provided to a single model, the model performance can be significantly improved. We will provide an additional explanation on diversity gain in Section VI.

As we will show in Section X, empirical results show that what has been observed in Fig. 3b is more typical than what has been observed in Fig. 3a. Then, it can be seen that trying many HPO models one by one is not necessary as long as S-Div is used and at least some of the chosen individual models happen to perform well for the given task. As the end result, the productivity of the DNN hyperparameter optimization process can be dramatically improved with diversification. For this reason, we believe diversification is one of the most basic and important enhancement strategies that should be considered all the time.

B. ADAPTIVE DIVERSIFICATION

A natural extension of S-Div is to adapt the model selection of the n'th iteration. While such an adaptation has been often treated as a Multi-Armed Bandit (MAB) problem in the previous HPO works, we would like to clarify that it is not a simple stationary MAB because of $\mathcal{H}$. Each model selection affects how $\mathcal{H}$ will be expanded, and subsequently, the performance of all individual models for the $n+1$'th iteration is affected through the dataset $\mathcal{H}$. Thus, the problem can be understood as a contextual bandit problem [31].

To perform basic investigation on the effects of adaptation, we have considered two adaptation schemes - *Hedge* and *ϵ-greedy*. As in S-Div, the same six models were used as the individual arms, and negative log error was chosen as the reward after trying several options. For ϵ-greedy, ϵ was linearly decreased from 1.0 to 0.1 as the evaluation step n increased. The results are shown in Fig. 3. To our surprise, neither Hedge nor ϵ-greedy showed a meaningful difference from S-Div for both benchmarks. Therefore, a further analysis on arm selection ratio was performed. In Table 9, the arm selection ratio over the six individual models are shown for S-Div, Hedge, and ϵ-greedy. As expected, S-Div equally selected from the six models. Hedge that selects the individual model with the best average reward during iterations 1 to $n-1$ also ended up selecting the six models almost equally. This indicates that none of the six models clearly and consistently outperformed due to the sharing of $\mathcal{H}$.

TABLE 9. The individual model selection ratio in % ($\mu \pm \sigma$). In the second column, the best performing strategy is marked in bold. In the third column, the most frequently selected individual models are marked in bold and the best-performing individual model is highlighted in green.

Benchmark	Diversification strategy	GP-EI	GP-PI	GP-UCB	RF-EI	RF-PI	RF-UCB
PTB-LSTM	S-Div	$\mathbf{16.9 \pm 0.1}$	$\mathbf{16.8 \pm 0.1}$	$\mathbf{16.6 \pm 0.0}$	$\mathbf{16.6 \pm 0.0}$	$\mathbf{16.6 \pm 0.0}$	$\mathbf{16.6 \pm 0.0}$
	Hedge	16.4 ± 2.8	16.4 ± 2.5	16.4 ± 2.9	$\mathbf{17.0 \pm 2.7}$	16.7 ± 2.8	$\mathbf{17.0 \pm 2.8}$
	ϵ-greedy	14.3 ± 3.1	$\mathbf{30.4 \pm 4.5}$	13.9 ± 2.9	13.6 ± 2.7	13.9 ± 3.1	13.8 ± 2.5
MNIST-LeNet2	**S-Div**	$\mathbf{16.8 \pm 0.1}$	$\mathbf{16.7 \pm 0.1}$	$\mathbf{16.6 \pm 0.1}$	$\mathbf{16.6 \pm 0.1}$	$\mathbf{16.6 \pm 0.1}$	$\mathbf{16.6 \pm 0.1}$
	Hedge	16.2 ± 2.0	15.7 ± 2.0	16.2 ± 2.0	$\mathbf{18.4 \pm 2.3}$	16.0 ± 1.8	17.5 ± 2.0
	ϵ-greedy	10.5 ± 2.0	11.3 ± 5.7	9.5 ± 1.6	10.6 ± 4.5	$\mathbf{46.7 \pm 10.6}$	11.3 ± 5.6

Unlike Hedge, ϵ-greedy ended up choosing GP-PI and RF-PI most frequently for PTB-LSTM and MNIST-LeNet2 benchmarks, respectively. For the first benchmark, GP-PI turns out to be the best performing single model. For MNIST-LeNet2, however, RF-PI turns out to be one of the best performing single models but not exactly the best one. The best performing one is RF-EI, and it was selected only 10.6% of the time on average. Overall, the results in Table 9 indicate that naïve adaption schemes fail to outperform S-Div because the main benefit comes from the sharing of $\mathcal{H}$ and not from the adaptation itself. More sophisticated adaptation schemes might be able to outperform S-Div, but we adopt S-Div as the diversification choice when designing DEEP-BO in Section VIII.

C. EXISTING WORKS

There are existing works where multiple models are considered together. GP-Hedge algorithm in [9] focuses on adaptive selection from a portfolio of acquisition functions governed by an online multi-armed bandit strategy where the base model is restricted to GP. The basic Hedge model in our experiment was based on the ideas of GP-Hedge with the concept extended to the model level. In [4], it is shown that certain choices for the nature of the GP (such as the type of kernel and the treatment of its hyperparameters) can play a crucial role for optimizing machine learning algorithms, and MCMC is adopted to blend acquisition functions arising from samples from the posteriors. Reference [32] explicitly focuses on constructing a novel kernel between models to explain a given dataset. Reference [33] introduces an automated BO approach that dynamically selects promising models for explaining the observed data using BO in model space.

While all of the works above are relevant, all of them have the viewpoint of choosing or approaching a single best BO model to explain the data. For instance, the single model might be an integration of multiple models where selection, sampling, or adaptive blending of acquisition function and/or kernel is performed according to the observed data. Our viewpoint is that model level diversity should be recognized in a way analogous to the ensemble technique where the N models bring different characteristics and strengths of their own. Unlike ensemble, however, the N models interact with one another through the dataset $\mathcal{H}$. This interaction can often make even a poorly performing model to be as useful as the best performing model for the given task. With our best knowledge, we are the first to explicitly consider model level *diversity* and experiment the interaction through $\mathcal{H}$, especially for the modern DNN benchmarks.

V. EARLY TERMINATION STRATEGY

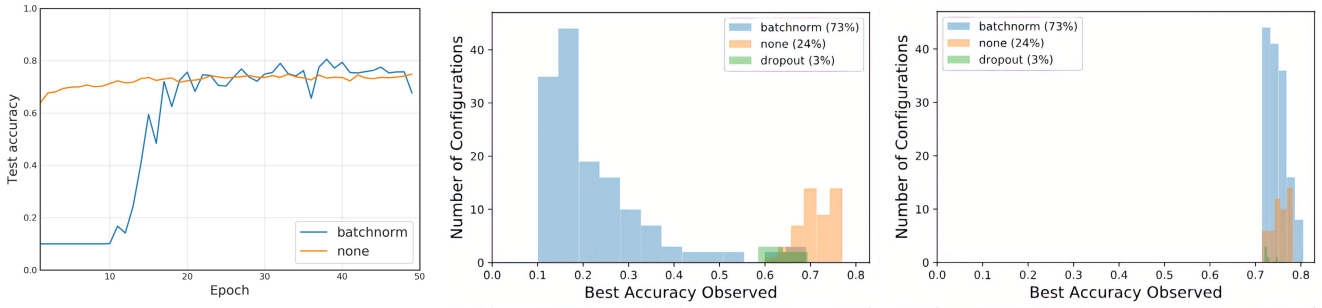
The configuration $\mathbf{x}_n$ chosen by HPO often results in a performance $f(\mathbf{x}_n)$ that is much worse than the best configuration found so far. If such a configuration can be identified in the early phase of DNN training (i.e. after training only a small number of epochs), it would be ideal to terminate the training and move on to the next candidate. This is a natural approach for DNN. For training DNN, typically *early stopping* is utilized to prevent performance degradation from an excessive training [36]. To enforce early stopping, validation performance is checked many times before completing the full training of max epoch E. Therefore, the training and validation performance samples over the course of training are made available (collectively called learning curve), and the samples can be used to predict what will be the final performance if the training is continued. This straightforward but effective idea has been studied in [10], [11], [23], [34], [35], and we call it *Early Termination Rule* (ETR) to distinguish it from early stopping.

Most of the previous works make certain assumptions on the characteristics of DNN's learning curve, introduce a probabilistic model, and perform extrapolation to predict the future performance. A common assumption is that the slope of a learning curve is initially large and gradually decreases with training. The assumption serves as the base for a variety of methods ranging over heuristic rules [10], probabilistic modeling [11], and regression [35]. In [23], envelope functions are adopted to provide theoretical bounds.

While such methods are valid and can be significantly helpful, especially when the max epoch E is set to a value much larger than what is really needed, they can also harm the performance of HPO by making undesirable early terminations.

A. RISK OF EARLY TERMINATION

Deep neural network is an active area of research. New techniques are continuously developed and popularized. For instance, dropout and batch normalization (*batchnorm*) became essential tools for designing new architectures. Whenever a new technique is introduced, the characteristics of learning curves are affected. As an example, analysis of training CIFAR10-CNN is shown in Fig. 4. First, the two best



(a) Top performing curves (b) Early performance (epoch [0 13]) (c) Later performance (epoch [38 50])

FIGURE 4. Training of CIFAR10-CNN. (a) Learning curve examples of the top performing configurations. Blue curve is with batch normalization on, and orange is with batch normalization off. (b) Histogram of performance during 0–13 epochs of training. Clearly the configurations with batch normalization tend to perform worse. (c) Histogram of performance during 38-50 epochs of training. The best performance is achieved when batch normalization is on. The results in (b) and (c) were generated using only 200 configurations in the surrogate dataset, where they end up as the top-200 configurations after full training.

performing configurations out of the 20,000 pre-evaluations were identified with batch normalization on and off, and their learning curves are shown in (a). In fact, the blue curve is the best configuration because it performs better than the orange curve. While the orange curve shows a strong correlation between the early phase of training and the later phase of training, the blue one shows an abrupt improvement around epochs 10–15. If the early termination rule is such that an early termination is decided for the blue curve, the best performing configuration will have no chance of being found by the HPO algorithm. This is an example of how a common assumption on learning curve can become completely wrong with an introduction of a new technique. Second, we have identified the top-200 configurations and generated their performance histogram for the early phase in (b) and later phase in (c). Clearly, all the high performance configurations with batchnorm on have poor performance in the early phase. The effect is less pronounced for dropout, though.

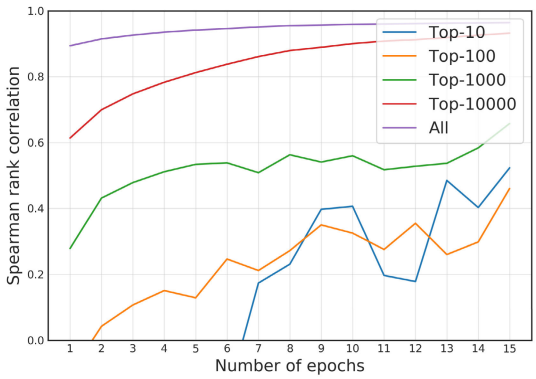


FIGURE 5. The correlation between validation performance during training and the best one found after completion.

As another example, training of MNIST-LeNet1 is considered in Fig. 5. For the top-K performing configurations, we have calculated the average Spearman rank correlation. For epoch i, the correlation was calculated between the performance of epoch i and the final performance after the full training (E epochs). For the top-10000, the correlation is high even for the early phase. As K is increased, the correlation starts to become smaller. For the top-10, the correlation for epochs between 1 and 6 is definitely not high where its values is calculated to be 0 or less. More examples on the characteristics of DNN learning curves can be found in [37].

Overall, it is still unclear what assumptions on DNN learning curves are and will be reasonable. The existing works focused on only a limited set of DNN tasks or datasets, and therefore it was possible to find a common assumption that worked well. With the assumption, early termination rules tended to be aggressive to maximally save the computing resources. When we investigated our six benchmarks, however, it was easy to identify counterexamples of the assumptions. This is similar to the problem of inductive bias design in machine learning. As long as tasks, datasets, and the component techniques of DNN are evolving, it will be impossible to come up with a common assumption on learning curve. The limitation is an important one to remember when designing early termination rules. Ideally, the early termination rule should be smart enough to learn a proper rule as more candidate configurations are evaluated. In this section, we provide the basic design elements of ETR and provide a conservative design. A further discussion is provided in the Section XI.

B. DESIGN ELEMENTS FOR EARLY TERMINATION

We surveyed the termination rules in the existing works, and they are summarized in Table 10. The common framework is to define an observation period, a time point for making an early termination decision, and a threshold for termination decision. We extracted four essential design elements and they are listed below.

- **Start point** s: It refers to the time point when the learning curve observation begins. The observations are used as the input to the termination decision.

TABLE 10. Early termination designs in previous works.

Name	Start point s	End point e	Checkpoint j	Threshold h	Termination prediction Θ
Predictive termination criterion [11]	epoch 0	same as j	tuning parameter	95% certainty the model would not improve the best-known performance when fully trained	Probabilistic model
Successive halving [23], [12]	same as e	same as j	determined by min epoch, max epoch and tuning parameter η.	1-1/η, η is a proportion of configurations discarded in each round.	Heuristic
Median stopping rule [10]	epoch 0	same as j	tuning parameter	The median value of the running averages of all completed trials' objectives reported up to point j	Heuristic
ETR-1, 2 [37]	same as e	same as j	tuning parameter	Random guess accuracy with training noise, 75% accuracy value of the best-known configuration at j	Heuristic
Termination criteria [35]	epoch 0	same as j	tuning parameter	99% certainty the model would not improve the best-known performance when fully trained. This certainty is a design parameter.	Regression model

- **End point** e: It refers to the time point when the learning curve observation ends. If $s = e$, the performance at the time point only is used as the observation.
- **Checkpoint** j: It refers to the time point when the observations are evaluated and a termination decision is made. Typically, j is set to be the same as e.
- **Threshold** h: It refers to the knob that controls the aggressiveness of termination at the checkpoint j. $h \in [0, 1]$ where $h = 0$ means no termination at all and $h = 1$ means unconditional termination.

A single termination decision consists of four elements $(s, e, j\, h)$. The number of checkpoints is also a design element, but typically only one checkpoint is used. When multiple checkpoints are used, one can use less aggressive settings in the early phase and use more aggressive settings in the later phase. The followings are the design principles that we recommend.

- Use a small number of checkpoints to keep ETR simple.
- The first checkpoint j should be sufficiently delayed to prevent unnecessary and harmful early terminations.
- To reduce the variance of estimation, s should be earlier than e ($s \neq e$).
- A model-free approach can be more robust to the changes in tasks, datasets, and techniques [10].
- When approaching the max epoch E, an aggressive threshold can be desirable.

Based on the principles, we provide a simple termination rule that will be later integrated into DEEP-BO.

C. COMPOUND RULE

We have designed a termination rule based on the above principles. We call it Compound Rule (CR) because it evaluates learning curves at two different checkpoints with adjustable thresholds. CR was designed to be conservative because ultimately we are interested in finding the top-level configurations and we should not eliminate any candidate unless confidence is very high.

We define two functions (l and t) that determine checkpoints and thresholds, respectively. These functions are represented with $\mathcal{A}$, $\mathcal{H}$, E, β, where $\mathcal{A}$ is a learning curve whose validation accuracy is typically observed for every epoch, $\mathcal{H}$ is the search history, E is max epoch, and β is the termination aggressiveness parameter.

Firstly, we introduce two formulaic tools for computing the above functions.

$$F_X(x) = P(X \leq x) \tag{2}$$

$$\bar{f}_{i:j}(\mathbf{x}) = \frac{1}{j}\sum_{k=i}^{j} f_k(\mathbf{x}) \quad \text{if } i \leq j \tag{3}$$

$F_X(x)$ refers to the cumulative distribution function for the distribution of a given random variable X. Secondly, we show l which returns the two checkpoints' epoch locations (epoch j_1 and j_2).

$$l(E, \beta) = \{\lfloor 0.5E \rfloor,\ \lfloor (1-\beta)E \rfloor\} = \{j_1,\ j_2\} \tag{4}$$

$$S = \{s \in \mathcal{H} \mid \text{for } \mathcal{A} \in s,\ n(\mathcal{A}) > j_1\} \tag{5}$$

We note that j_1 is set to be larger than the nearest integer to $E/2$ to make CR *low-risk*. We hypothesize that this choice can decrease the occurrences of false positives which distract BO. Thus, we save *less* resources compared to using a smaller j_1 value for some single configuration but we save resources *frequently* using multiple checkpoints. This approach would be more resource-efficient in aggregate since BO might be able to find more desirable configurations based on a more reliable history. Thirdly, we show t which returns one of two thresholds. If $j \leq j_1$, t returns threshold h_1 for epoch j_1, and otherwise returns threshold h_2 for epoch j_2.

$$t(j, \mathcal{A}, \mathcal{H}, \beta) = \begin{cases} F_X^{-1}(\beta) & \text{if } j \leq j_1 \\ F_X^{-1}(1-\beta) & \text{otherwise} \end{cases} \quad \text{for,}$$

$$X = \begin{cases} \{x \mid x = \bar{f}_{1:j_1}(\mathbf{x}) \text{ for } \mathbf{x} \in \mathcal{H}\} & \text{if } j \leq j_1 \\ \{x \mid x = \bar{f}_{j_1:j_2}(\mathbf{x}) \text{ for } \mathbf{x} \in \mathcal{S}\} & \text{otherwise} \end{cases} \tag{6}$$

The S holds the history for configurations that survived the checkpoint j_1 computed by the function l.

Fig. 6 illustrates that the learning curves of hyperparameter configurations can be categorized into one of three types: untrainable, desirable, or the rest. CR is motivated to be more error-proof by ignoring the rest. Dual checkpoints support the understanding that the clearly untrainable ones are much

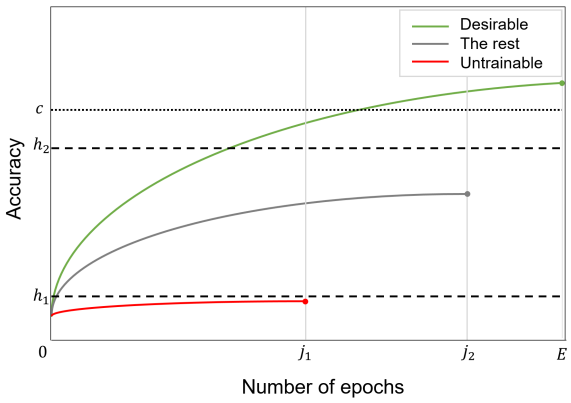


FIGURE 6. Illustration of learning curves evaluated by the Compound Rule. Note that c is the current best accuracy.

easier to spot with lesser resources compared to the truly desirable. These checkpoints are comprised by thresholds h_1, h_2, and epochs j_1, j_2 which are determined by the percentile $\beta \in (0.0, 0.5]$ and max epoch E.

- Checkpoint-1 aims to discard untrainable configurations. It terminates the training of the current k_{th} configuration $\mathbf{x}_k$ at epoch j_1, $0 < j_1 < E$, if its until-then best performance is worse than threshold h_1. Checkpoint-1 exclusively terminates them with a low threshold h_1.
- Checkpoint-2 aims to spare desirable configurations likely to outperform the current best. $\mathbf{x}_k$ that survives checkpoint-1 is stopped at epoch j_2, $j_1 < j_2 < E$, if its until-then best performance is below threshold h_2.

The β is yet another hyperparameter that may be tuned based on practical user constraints to max epoch E. Because the checkpoints and thresholds are expressed respectively by β and $(1 - \beta)$ in an inversely proportional relationship, untrainable/desirable configurations can be determined more convincingly when β is smaller. The performance of CR can be highly dependent on the choice of max epoch E, and it will be discussed further in Section XI.

VI. PARALLELIZATION STRATEGY

Parallelization is an important strategy for handling computation intensive deep neural network problems. For instance, hundreds of machines were used to reach state-of-the-art performance in [38] and [39]. It is also important for the hyperparameter optimization of deep neural networks because HPO is one of the most time-consuming DNN problems.

For many of the traditional BO problems, the evaluation time of $f(\mathbf{x}_n)$ is fixed and not dependent on the choice of $\mathbf{x}_n$. In such cases, it is natural to run M evaluations in parallel over M processors, update $\mathcal{H}$ with the newly obtained M results, choose the next batch of M configurations to evaluate, and repeat from the beginning. In this case, the core problem becomes the selection of the next batch configuration with M candidates to evaluate. Many related works exist where some also provide theoretical analysis [40]–[45].

The time for DNN training, however, can significantly vary depending on the chosen configuration $\mathbf{x}_n$. Then, synchronous parallelization is not desirable because many of the processors will need to stay idle until the last processor completes the training. For this reason, practical BO algorithms like [3], [4] assume asynchronous parallelization, and we also only focus on asynchronous parallelization in this work. For an asynchronous case, solving batch configuration becomes very difficult because even the concept of a batch cannot be well defined. Instead, we treat the situation as a single sequential selection problem with pending evaluations.

A. HANDLING PENDING EVALUATIONS

When $\mathcal{H}$ is shared asynchronously among parallel processors (workers), the most fundamental issue is handling the pending evaluations. The most straightforward policy is to ignore the others and include only the completed results into $\mathcal{H}$. The downside of this strategy is that it can result in a duplicate selection where the chosen $\mathbf{x}^*$ is already being evaluated by another worker. To avoid duplicates, [3] uses the next candidate available among the ones that are not being evaluated. In this study, we consider the following three simple heuristics when $\mathbf{x}^*$ is chosen as one of the pending evaluations:

- *random*: randomly select a new candidate
- *next candidate*: choose the next best candidate that is not pending
- *in-progress*: temporarily add the premature results of the pending evaluations to $\mathcal{H}$ and calculate $\mathbf{x}^*$

DNN is almost always trained with Stochastic Gradient Descent (SGD), and the validation performance is frequently evaluated before completing the training. Therefore, premature performance evaluations become available during DNN training, and *in-progress* takes advantage of them. Apparently, the use of a premature evaluation can be harmful if the final evaluation result is quite different from the premature evaluation. The negative effect, however, is self-corrected after completing the training because the final result can be used to replace the premature result in $\mathcal{H}$.

TABLE 11. *Expected time* $\mathbb{E}[\tau]$ for three duplicate handling strategies ($\mu \pm 0.25\sigma$ hours). Over $M = 6$ processors, GP-EI was run for the difficult targets.

Benchmark	*random*	*next candidate*	*in-progresss*
MNIST-LeNet1	6.9 ± 1.2	6.6 ± 1.2	$\mathbf{6.1 \pm 1.2}$
MNIST-LeNet2	4.2 ± 0.9	4.9 ± 0.9	$\mathbf{3.2 \pm 0.8}$
PTB-LSTM	2.1 ± 0.2	2.4 ± 0.3	$\mathbf{1.7 \pm 0.2}$
CIFAR10-CNN	2.4 ± 0.4	2.7 ± 0.4	$\mathbf{1.2 \pm 0.2}$
CIFAR10-ResNet	14.7 ± 1.9	13.9 ± 1.8	$\mathbf{10.6 \pm 1.5}$
CIFAR100-CNN	1.2 ± 0.1	1.3 ± 0.1	$\mathbf{1.0 \pm 0.1}$

We evaluated the expected time $\mathbb{E}[\tau]$ using the three duplicate handling strategies, and the results are shown in Table 11. For all six benchmarks, we can see that in-progress achieves the best $\mathbb{E}[\tau]$.

B. PER-PROCESSOR EFFICIENCY

It would be ideal if the calculation speed is increased by M times by using M processors, i.e., the normalized speed remains at one regardless of M. For the asynchronous parallelization, the selection of $\mathbf{x}^*$ needs to be made while $M-1$ of the evaluations are pending. Compared to the single processor case, this means $\mathcal{H}$ will be less informative with $M-1$ data samples missing. Therefore, we can expect the normalized speed to decrease as M is increased. This loss in per-processor efficiency can be especially large at the beginning of BO because all of the first M candidates need to be chosen while $\mathcal{H}$ remains empty.

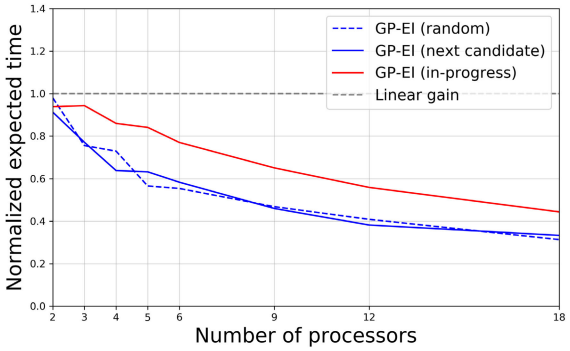


FIGURE 7. Normalized performance ($\frac{\mathbb{E}_1[\tau]}{M\mathbb{E}_M[\tau]}$) as the number of processors is increased. CIFAR10-ResNet benchmark with the difficult target was evaluated using GP-EI.

An experiment result is shown in Fig. 7 where normalized performance $\frac{\mathbb{E}_1[\tau]}{M\mathbb{E}_M[\tau]}$ is plotted with respect to the number of processors ($\mathbb{E}_1[\tau]$ is $\mathbb{E}[\tau]$ for one processor and $\mathbb{E}_M[\tau]$ is $\mathbb{E}[\tau]$ for M processors). For the experiment, in-progress turns out to outperform the other two options and the normalized performance is around 0.8 for $M = 6$ and around 0.4 for $M = 18$. Though not shown in the figure, we have experimented with the other benchmarks and similar shapes of per-processor efficiency were observed. The result in Fig. 7 is for the difficult target. For the easy target, HPO is completed much faster than for the difficult target. Therefore, the inefficiency in the beginning of BO can have a bigger impact on the overall performance.

C. DIVERSIFICATION UNDER PARALLEL PROCESSING

A natural way of implementing model level diversification under asynchronous parallelization is to have N models take turns as in S-Div whenever a processor becomes available. We call this simple extension as *P-Div*. For the special case of $N = M$, each model is assigned to a processor at any time.

When rotating over N models, a neat side effect is that the n'th model will choose the candidate $\mathbf{x}^* = \arg\max_{\mathbf{x}\in\mathcal{X}} \hat{f}_n(\mathbf{x})$ where $\mathbf{x}^*$ is unlikely to be same for the other models even for the same $\mathcal{H}$. Therefore, the chance of duplication becomes smaller. When $N < M$, some of the models might be concurrently used and therefore a duplicate handing strategy needs to be adopted.

D. ADDITIONAL EXPLANATIONS ON DIVERSIFICATION

Now that we have addressed both diversification and parallelization, we provide additional explanation on diversification here. For an HPO algorithm, we can define its failure rate as $p_f = 1 - \mathbf{P}_c(\tau \le t)$ where τ is a random variable representing the algorithm's time to achieve the target accuracy c. If the algorithm is run over M parallel workers without sharing their histories, the chance of all M attempts failing is simply p_f^M because τ forms an i.i.d. process. Therefore, we can see that M'th order diversity gain (increase in the exponent of the failure rate) is achieved at the cost of M times increase in resources.

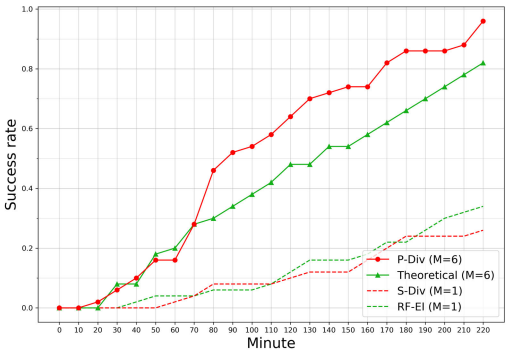
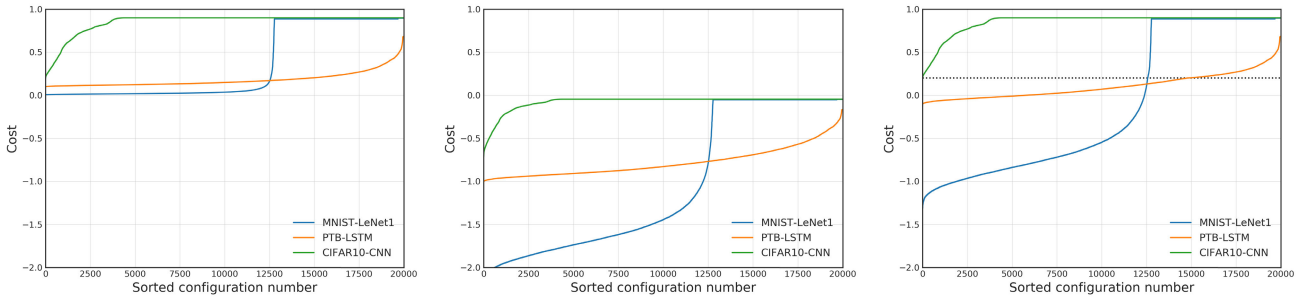


FIGURE 8. An illustration of diversity gain. Success rate is plotted for MNIST-LeNet2 benchmark with the difficult target. For S-Div and P-Div, $N = 6$ was used (GP-EI, GP-PI, GPUCB, RF-EI, RF-PI, and RF-UCB).

In Fig. 8, success rate curves are shown for MNIST-LeNet2 benchmark with the difficult target accuracy. RF-EI was the best performing individual model as was shown in Fig. 3b. The success rate of RF-EI($M = 1$) is equal to $1-p_f$. Theoretical($M = 6$) is a synthetic curve generated by plotting $1-p_f^6$. The success rate of Theoretical($M = 6$) ramps up much faster than RF-EI($M = 1$) which demonstrates the power of order six diversity. S-Div($M = 1$) also utilizes diversity over the six models, but it suffers from resource sharing (taking turns) and performs only as well as the best performing single model, RF-EI($M = 1$). P-Div($M = 6$) utilizes six processors and also blends the six individual models through $\{\mathcal{H}\}$. As a result, it performs as well as Theoretical($M = 6$). In fact, P-Div($M = 6$) performs even better than Theoretical($M = 6$) after passing 70 minutes. This indicates that P-Div($M = 6$) is not only capable of realizing order six diversity over RF-EI($M = 1$) but also capable of bringing an extra enhancement. A large portion of this substantial extra enhancement is due to the mix of $N = 6$ individual algorithms with different modeling characteristics. This comes in two-fold. The first is the different characteristics of the N algorithms, where at least one will likely work well. This itself is another type of diversification effect that is different from the p_f^M effect, and basically says that at least one of $p_{f,1}, \ldots, p_{f,N}$ will likely be small over the N individual models for any given task. The second is the *cooperative* nature of the diversified algorithm thanks to the sharing of $\{\mathcal{H}\}$. When an individual model A is not capable of



(a) Without transformation (b) Log transformation (c) Hybrid transformation (α = 0.2)

FIGURE 9. The error rate distributions are shown for three benchmarks. (a) In the low error range (left side of the figure), the green curve shows visually recognizable change for different configurations, but orange and blue look flat. The blue curve stays very close to zero for a wide range. (b) Log transformation can amplify the performance difference when the error rate is very close to zero. (c) Hybrid approach is a compromise where log transformation is applied only when the error rate is between 0 and α. A new hyperparameter α needs to be introduced, though.

TABLE 12. *Expected time* $\mathbb{E}[\tau]$ for three benchmarks ($\mu \pm 0.25\sigma$ hours). GP-EI was used for the evaluations with the difficult target.

Benchmark	No transformation ($\alpha = 0$)	Hybrid transformation					Log transformation ($\alpha = 1$)
		α = 0.1	α = 0.3	α = 0.5	α = 0.7	α = 0.9	
MNIST-LeNet1	71.0 ± 21.8	$\mathbf{7.5 \pm 1.1}$	10.6 ± 2.1	9.7 ± 1.6	9.25 ± 1.8	8.3 ± 1.4	12.3 ± 2.2
PTB-LSTM	8.4 ± 0.9	8.3 ± 0.9	$\mathbf{5.9 \pm 0.7}$	6.1 ± 0.6	6.4 ± 0.6	6.8 ± 0.6	35.8 ± 8.8
CIFAR10-CNN	6.3 ± 1.0	6.4 ± 1.0	6.4 ± 0.9	$\mathbf{6.2 \pm 0.8}$	$\mathbf{6.2 \pm 0.8}$	6.6 ± 0.8	7.1 ± 1.1

selecting good target candidates for exploration and is stuck in bad regions, some of the other models might make better selections that serve as exploratory samples to the model A. Then the poorly performing model A can escape from the bad cycle and start to perform well.

VII. COST FUNCTION TRANSFORMATION STRATEGY

Another important topic to consider is the cost function transformation. Hyperparameter tuning of deep neural networks is often applied to classification tasks where the performance value is an error rate. When the target error rate is close to 0, it becomes difficult to model the small performance change of $f(\mathbf{x})$ if a linear scale is used to represent the target error rate. In such cases, it can be advantageous to use a log transformation that is known to be effective for modeling nonlinear and multimodal functions [46]. When the error rate is large and far from 0, however, using log-transformation may be harmful, too. Therefore, we introduce *hybrid transformation* where log transformation is applied only when the error rate is between 0 and α, and the original linear scale is used when the error rate is larger than α. When $f(\mathbf{x})$ of BO is error rate and its value is err, hybrid transformation can be expressed as below.

$$g(err, \alpha) = \begin{cases} err & \text{if } err > \alpha \\ \log(err) + (\alpha - \log(\alpha)), & \text{otherwise} \end{cases} \quad (7)$$

Note that $(\alpha - \log(\alpha))$ needs to be added to make $g(err, \alpha)$ continuous at $err = \alpha$. When $\alpha = 0$, the hybrid transformation does not have any effect. When $\alpha = 1$, the hybrid transformation becomes the same as the log transformation. When the cost is not error rate, such as in regression tasks, the data can be pre-processed to have its target range adjusted between 0 and 1 as we have done so for the PTB-LSTM benchmark. The overhead for such a manual pre-processing might or might not be worth the effort depending on the data and task.

Fig. 9 shows how cost function transformation reshapes the values of the response surface for three of the benchmarks. Table 12 shows how the choice of α affects the HPO's $\mathbb{E}[\tau]$ performance for the same three benchmarks. For MNIST-LeNet1 benchmark, the best accuracy is 0.9939, difficult target accuracy is 0.9933, and easy target accuracy is 0.9903 (see Table 8). The corresponding error rates are 0.0061, 0.0067, and 0.0087, respectively. Because the error rates are very close to 0, we can expect log transformation to be beneficial and the results in Table 12 confirm the expectation. In fact, $\mathbb{E}[\tau]$ for no transformation is about 6 times worse than $\mathbb{E}[\tau]$ for log transformation. One can do even better by adopting hybrid transformation and tuning it, and $\alpha = 0.1$ turns out to be a good choice. Also, choosing any of 0.1, 0.3, 0.5, 0.7, and 0.9 turns out to be better than both no transformation and log transformation. For PTB-LSTM benchmark, the corresponding error rates are 0.0992, 0.1007, and 0.1065, respectively, and the values are somewhat far from 0. For the benchmark, log transformation turns out to be quite harmful where it increases $\mathbb{E}[\tau]$ by more than four times. Hybrid transformation is helpful with $\alpha = 0.3$ as the best choice. As in MNIST-LeNet1, all the other choices of α under hybrid transformation happened to be helpful for the PTB-LSTM benchmark. For the CIFAR10-CNN benchmark, the corresponding error rates are 0.1948, 0.2164, and 0.3815, respectively, and the values are even farther from 0 than the

PTB-LSTM benchmark. CIFAR10-CNN corresponds to the green curve in Fig. 9 and it is the easiest one to visually recognize the performance difference for the low error rate configurations. For this benchmark, it turns out that all the configurations produce comparably good results.

Clearly, cost function transformation can be essential for HPO as empirically shown with the MNIST-LeNet1 benchmark. However, the cost function transformation should be carefully chosen because its effect can be different for different tasks. Based on the empirical results above, it might be prudent to choose hybrid transformation and avoid extreme choices such as no transformation or log transformation.

VIII. DEEP-BO

So far, we have introduced four basic strategies to enhance performance and robustness of DNN HPO. Here, we combine S-Div/P-Div, CR, in-progress, and hybrid transformation and design a new BO algorithm named DEEP-BO (Diversified, Early-termination Enabled, Parallel Bayesian Optimization). The algorithm was specifically designed to address the tuning of *deep neural networks*, and the pseudocode is shown in Algorithm 2.

IX. EXPERIMENT SETTINGS

We evaluated DEEP-BO and several well-known HPO algorithms using the six DNN benchmarks. The HPO algorithms are summarized below.

- The random algorithm in [1]. It is often used to replace grid-search.
- The six individual BO algorithms - GP-EI, GP-PI, GP-UCB, RF-EI, RF-PI, and RF-UCB. The two popular modeling algorithms (GP, RF) [4], [20] are combined with three acquisition functions (Expected Improvement, Probability of Improvement, and Upper Confidence Bound).
- GP-Hedge introduced in [9], where GP-EI, GP-PI, GP-UCB are used as the three arms of a multi-armed bandit setting. We used $\eta = 0.1$ for the coefficient of gain update.
- RF-EI with Learning Curve Extrapolation (LCE) [11].
- GP-EI with Median Stopping Rule (MSR) [10]
- BOHB that allocates resources using an infinite bandit-based approach as in Hyperband (HB) [23]. Unlike Hyperband, BOHB utilizes density estimation algorithm [3], whereas Hyperband randomly chooses the next candidates [12].
- Our algorithm, DEEP-BO.

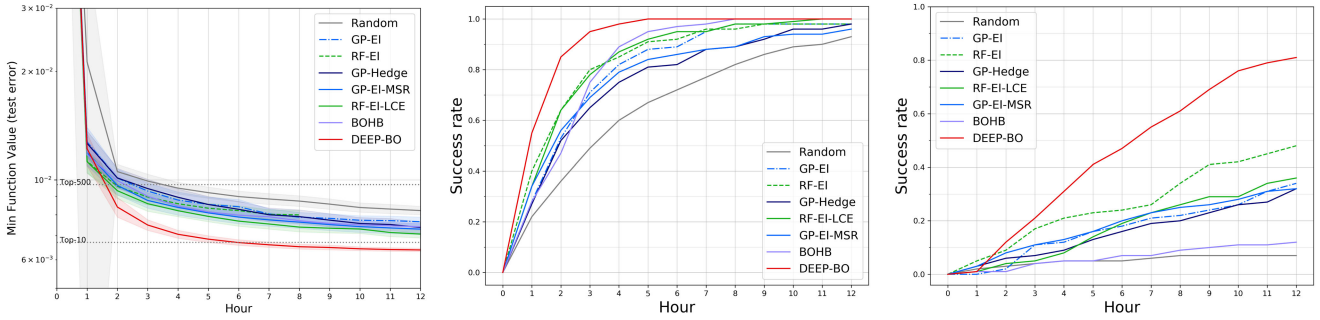
HPO algorithms have their own hyperparameters. The following is the list of hyperparameters and their values that were used for the experiments. DEEP-BO's number of processors $M = 1$ or 6, diversification level $N = 6$ (GP-EI, GP-PI, GP-UCB, RF-EI, RF-PI, and RF-UCB), CR termination percentile $\beta = 0.1$, in-progress option for duplicate handling of $M = 6$, and hybrid transformation with $\alpha = 0.3$. We did not try to adjust the details of DEEP-BO because we wanted a robust and general solution for tuning a broad range of deep neural networks. Therefore, all the design decisions were intended for robustness. A further improvement should be possible by choosing a better and more diverse set of N models, by tuning threshold values for the given data and task (in a light manner before running DEEP-BO), and so on.

For the termination criterion of LCE and MSR, we chose their j values to be the same as the j_1 of CR, i.e., $1/2E$. In this way, we made sure that LCE and MSR do not behave too aggressively. Maximum epoch E for each benchmark can be found in Table 1. We chose the predictive model setting of LCE to be conservative and to terminate when the posterior probability of a configuration is greater than the threshold h described in Table 10. In this case, the time to build a probabilistic model was also added when it terminated.

Algorithm 2 DEEP-BO Diversified, Early-Termination-Enabled, and Parallel Bayesian Optimization

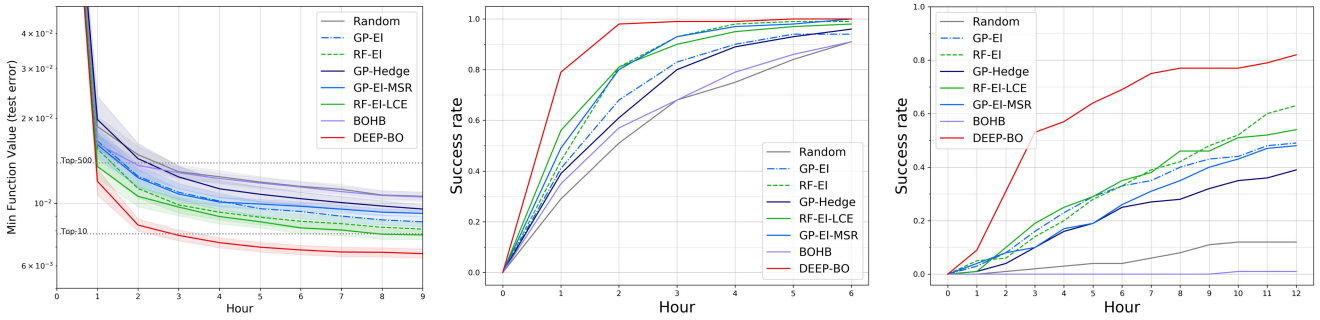
Inputs: True black-box DNN f, base models $\hat{f}_1, \ldots, \hat{f}_N$, processors $p_1, \ldots, p_M$, hyperparameter space $\mathcal{X}$, target accuracy c, early termination functions t and l, early termination parameter β, hybrid transformation g, hybrid transformation parameter α, max epoch E.

$\mathcal{H} \leftarrow \emptyset$
for $i = 1, 2, \ldots$ **do**
  $n \leftarrow \mod(i, N) + 1$
  $m \leftarrow \mod(i, M) + 1$
  # ==========
  # *m processes of the following part are run in parallel*
  Select $\mathbf{x}^* \in \arg\max_{\mathbf{x} \in \mathcal{X}} \hat{f}_n(\mathbf{x}; \mathcal{H})$
  $\mathcal{A} \leftarrow \emptyset$, $y^* \leftarrow 0$
  for $j = 1, \ldots, E$ **do**
    Train f for one epoch on p_m
    Evaluate $y_j \leftarrow f_j(\mathbf{x}^*)$ where $f_j(\mathbf{x}^*)$ is accuracy evaluated after training f for j epochs
    $\mathcal{A} \leftarrow \mathcal{A} \cup (j, y_j)$
    $y^* \leftarrow \max(y^*, y_j)$
    if $j = 1$ **then**
      Update $\mathcal{H} \leftarrow \mathcal{H} \cup (\mathbf{x}^*, g(1.0 - y^*, \alpha), \mathcal{A})$
    else
      Replace $\mathcal{H}$'s element that contains $\mathbf{x}^*$ with $(\mathbf{x}^*, g(1.0 - y^*, \alpha), \mathcal{A})$
    end if
    if $j \in l(E, \beta)$ **then**
      if $y^* < t(j, \mathcal{A}, \mathcal{H}, \beta)$ **then**
        break # *early termination*
      end if
    end if
  end for
  # ==========
  if $y^* > c$ **then** # *any termination rule can be used*
    break # *terminate if desired accuracy*
  end if
end for



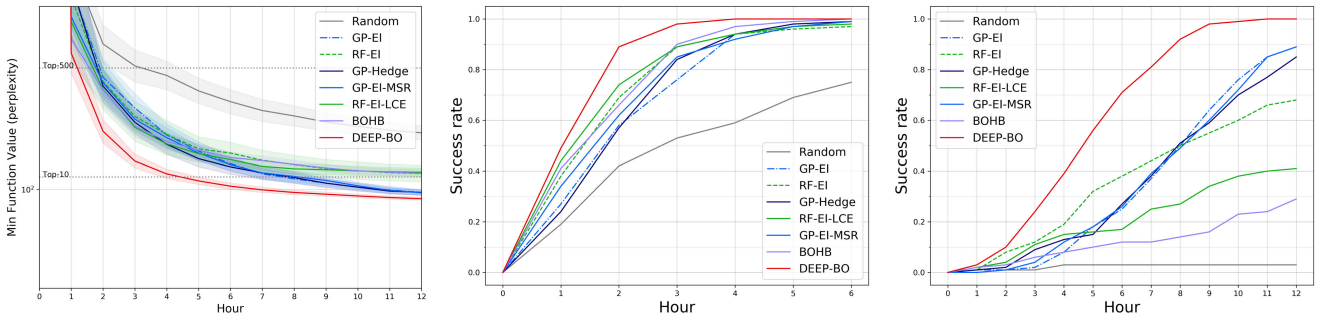
(a) Minimum function value (b) Success rate for easy target (c) Success rate for difficult target

FIGURE 10. Performance plots for MNIST-LeNet1 using a single processor.



(a) Minimum function value (b) Success rate for easy target (c) Success rate for difficult target

FIGURE 11. Performance plots for MNIST-LeNet2 using a single processor.



(a) Minimum function value (b) Success rate for easy target (c) Success rate for difficult target

FIGURE 12. Performance plots for PTB-LSTM using a single processor.

In the case of BOHB, their hyperparameters were tuned for our benchmarks, and the following values were used. The scaling parameter $\eta = 3$, the number of samples used to optimize the acquisition function is 64, the fraction of purely random configuration $\rho = 0.33$, the minimum budget is 1, and the maximum budget is E. These settings, except the minimum and maximum budgets, turned out to be the same as the values used in [12] for the MNIST dataset.

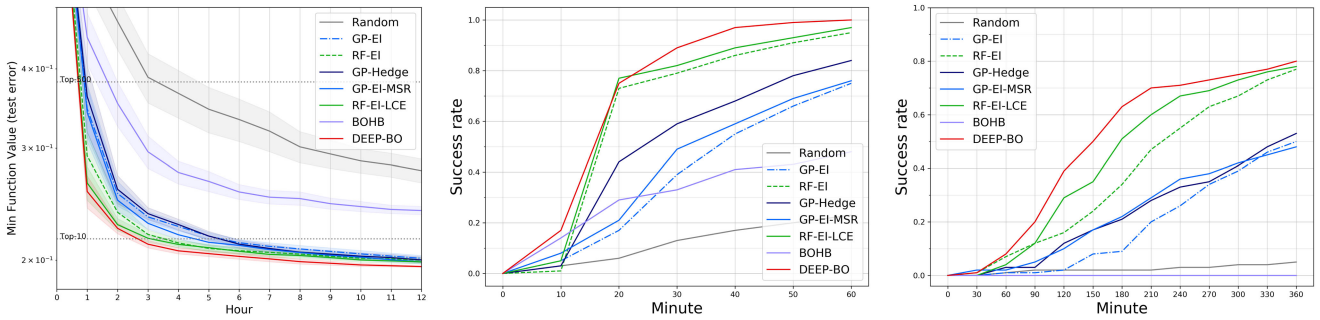
We repeated a full episode of HPO 100 times for each algorithm. As mentioned in Section III, we chose two target goals for each benchmark (easy and difficult targets). As we will show in the result section, some HPO algorithms work better for easy targets and some others for difficult targets. To have a fair evaluation, we have included both easy and difficult targets.

X. EXPERIMENT RESULTS

In this section, the experiment results are presented.

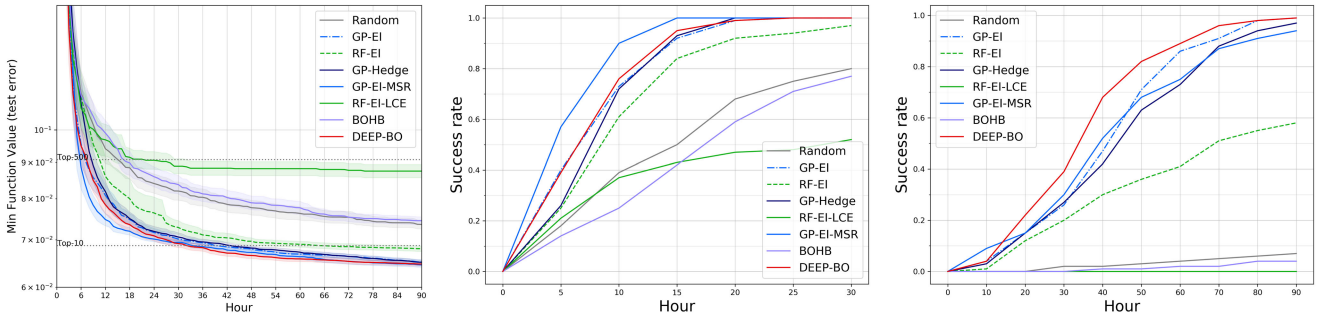
A. PERFORMANCE FOR SINGLE PROCESSOR

The experiment results for $M = 1$ are shown in Fig. 10–15. In the left column, performance is shown using the standard metric of min function value. In the middle and right columns, performance is shown using *success*



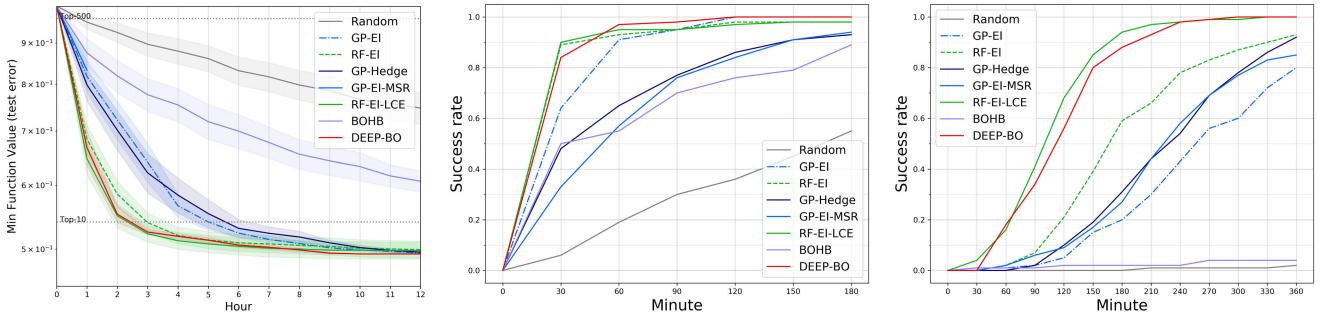
(a) Minimum function value (b) Success rate for easy target (c) Success rate for difficult target

FIGURE 13. Performance plots for CIFAR10-CNN using a single processor.



(a) Minimum function value (b) Success rate for easy target (c) Success rate for difficult target

FIGURE 14. Performance plots for CIFAR10-ResNet using a single processor.



(a) Minimum function value (b) Success rate for easy target (c) Success rate for difficult target

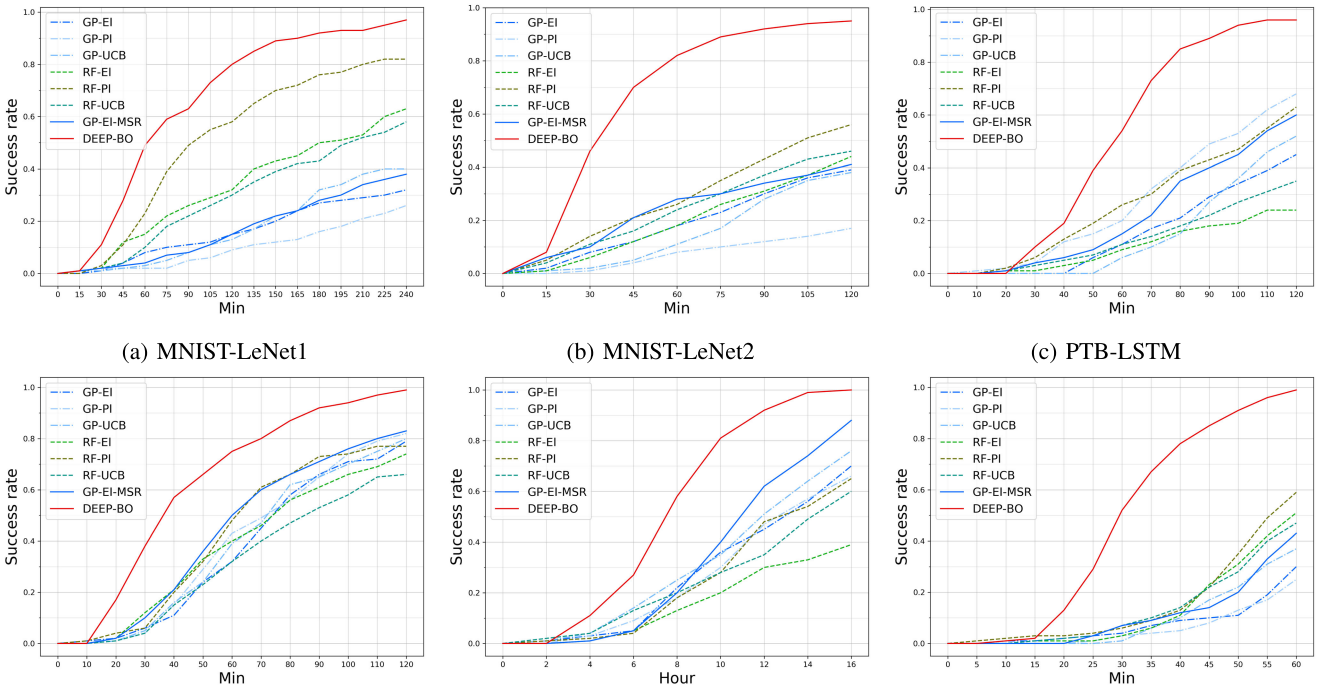
FIGURE 15. Performance plots for CIFAR100-CNN using a single processor.

rate where the middle column is for the easy targets and the right column is for the difficult targets. The random algorithm consistently performed very poorly, especially for the difficult targets. When the hyperparameter space $\mathcal{X}$ is large, random search pretty much fails. DEEP-BO consistently showed top-level performance for all of the six benchmarks and the two target difficulties. For the difficult targets, DEEP-BO often outperformed all the other algorithms with a wide performance gap. Overall, DEEP-BO showed a robust performance for the cases that we have evaluated. All the other algorithms failed to show top-level performance in a consistent way.

In addition to the minimum function value and the success rate, evaluation was repeated using *expected time* $\mathbb{E}[\tau]$ as the performance metric. The results are summarized in Table 13. The $\mathbb{E}[\tau]$ tends to have a high variance because of the nature of HPO problems. Therefore, we show the performance of $\mathbb{E}[\tau]$ with its 0.25 standard deviation that was calculated using the 100 randomly seeded runs. In general, DEEP-BO outperformed the other algorithms, and it achieved the best

TABLE 13. ***Expected time*** $\mathbb{E}[\tau]$ **when using a single processor (**$\mu \pm 0.25\sigma$ **hours). Both of the easy and difficult targets were considered.**

Target	Benchmark	Random	GP-EI	GP-Hedge	GP-EI-MSR	RF-EI	RF-EI-LCE	BOHB	DEEP-BO
Easy (top-500th accuracy)	MNIST-LeNet1	4.4 ± 1.0	2.7 ± 0.9	3.2 ± 0.9	3.0 ± 0.9	2.2 ± 0.7	2.1 ± 0.5	2.2 ± 0.4	$\mathbf{1.2 \pm 0.2}$
	MNIST-LeNet2	2.6 ± 0.6	1.9 ± 0.6	2.0 ± 0.5	1.3 ± 0.3	1.4 ± 0.4	1.4 ± 0.4	2.5 ± 0.6	$\mathbf{0.7 \pm 0.2}$
	PTB-LSTM	3.9 ± 0.9	2.0 ± 0.3	1.9 ± 0.3	1.9 ± 0.4	3.2 ± 3.8	1.6 ± 0.4	1.6 ± 0.3	$\mathbf{1.1 \pm 0.2}$
	CIFAR10-CNN	3.3 ± 0.8	0.8 ± 0.1	0.6 ± 0.1	0.7 ± 0.1	0.4 ± 0.1	0.4 ± 0.1	1.4 ± 0.3	$\mathbf{0.2 \pm 0.1}$
	CIFAR10-ResNet	18.0 ± 3.8	6.0 ± 0.9	8.4 ± 1.2	$\mathbf{5.5 \pm 0.7}$	9.1 ± 2.0	58.1 ± 13.5	21.0 ± 4.1	7.3 ± 1.1
	CIFAR100-CNN	3.7 ± 0.7	0.5 ± 0.1	1.2 ± 0.4	1.3 ± 0.5	0.4 ± 0.1	0.4 ± 0.2	1.4 ± 0.5	$\mathbf{0.3 \pm 0.1}$
	Mean	6.0 ± 1.5	2.3 ± 0.5	2.9 ± 0.7	2.3 ± 0.5	2.8 ± 0.8	10.7 ± 5.8	5.0 ± 2.0	$\mathbf{1.8 \pm 0.7}$
Difficult (top-10th accuracy)	MNIST-LeNet1	169.1 ± 35.7	53.1 ± 18.6	17.0 ± 2.1	16.3 ± 2.2	43.2 ± 14.9	16.2 ± 2.1	20.7 ± 1.6	$\mathbf{8.2 \pm 2.1}$
	MNIST-LeNet2	99.9 ± 21.5	29.6 ± 11.7	29.3 ± 7.8	24.2 ± 7.6	16.5 ± 6.0	41.4 ± 16.7	35.8 ± 3.0	$\mathbf{7.6 \pm 2.6}$
	PTB-LSTM	188.3 ± 42.3	8.2 ± 0.9	8.3 ± 0.9	7.9 ± 0.7	25.8 ± 15.4	69.2 ± 26.1	28.3 ± 5.7	$\mathbf{4.8 \pm 0.6}$
	CIFAR10-CNN	138.2 ± 34.4	7.4 ± 1.1	6.6 ± 1.0	6.8 ± 1.1	4.7 ± 0.8	4.1 ± 0.7	29.4 ± 0.5	$\mathbf{3.4 \pm 0.6}$
	CIFAR10-ResNet	756.1 ± 181.2	44.1 ± 7.2	50.6 ± 7.4	43.8 ± 6.4	55.7 ± 7.6	120.5 ± 0.1	129.8 ± 4.5	$\mathbf{38.4 \pm 4.3}$
	CIFAR100-CNN	195.8 ± 38.8	4.1 ± 0.4	4.1 ± 0.5	4.2 ± 0.8	3.1 ± 1.4	$\mathbf{1.8 \pm 0.2}$	24.8 ± 2.5	1.9 ± 0.2
	Mean	257.9 ± 61.7	24.4 ± 5.3	19.3 ± 4.5	17.2 ± 4.3	24.8 ± 5.3	42.2 ± 11.5	44.8 ± 10.5	$\mathbf{10.7 \pm 3.5}$



(a) MNIST-LeNet1 (b) MNIST-LeNet2 (c) PTB-LSTM

(d) CIFAR10-CNN (e) CIFAR10-ResNet (f) CIFAR100-CNN

FIGURE 16. **Success rate plots for the difficult targets using six processors.**

TABLE 14. ***Expected time*** $\mathbb{E}[\tau]$ **when using six processors (**$\mu \pm 0.25\sigma$ **hours). Only the difficult targets were considered.**

Benckmark	GP-EI	GP-EI-MSR	GP-PI	GP-UCB	RF-EI	RF-PI	RF-UCB	DEEP-BO
MNIST-LeNet1	10.4 ± 2.4	4.0 ± 0.4	11.9 ± 2.2	8.9 ± 2.0	4.7 ± 1.3	2.4 ± 0.5	4.6 ± 1.1	$\mathbf{1.4 \pm 0.3}$
MNIST-LeNet2	4.6 ± 1.4	2.4 ± 0.4	8.4 ± 1.8	3.8 ± 1.0	3.0 ± 0.7	3.3 ± 1.1	2.4 ± 0.5	$\mathbf{0.8 \pm 0.3}$
PTB-LSTM	2.1 ± 0.2	1.8 ± 0.2	1.7 ± 0.2	2.0 ± 0.2	2.5 ± 0.2	1.8 ± 0.2	2.3 ± 0.2	$\mathbf{1.0 \pm 0.1}$
CIFAR10-CNN	1.5 ± 0.2	1.3 ± 0.2	1.5 ± 0.3	1.5 ± 0.3	1.5 ± 0.3	1.4 ± 0.3	1.8 ± 0.3	$\mathbf{0.8 \pm 0.1}$
CIFAR10-ResNet	13.0 ± 1.4	11.4 ± 1.0	14.1 ± 1.7	12.3 ± 1.4	18.1 ± 1.8	14.4 ± 1.7	14.6 ± 1.7	$\mathbf{7.7 \pm 0.6}$
CIFAR100-CNN	1.2 ± 0.1	1.1 ± 0.1	1.4 ± 0.1	1.2 ± 0.1	1.1 ± 0.1	1.1 ± 0.3	1.0 ± 0.1	$\mathbf{0.5 \pm 0.1}$
Mean	5.5 ± 1.3	3.7 ± 1.0	6.5 ± 1.4	5.0 ± 1.2	5.2 ± 1.6	4.1 ± 1.3	4.5 ± 1.3	$\mathbf{2.0 \pm 0.7}$

performance for ten out of twelve test cases. Even for the two cases, DEEP-BO's performance was very close to the top performance.

B. PERFORMANCE FOR SIX PROCESSORS

The results for $M = 6$ processors are shown in Fig. 16 and Table 14. Only difficult targets were considered because

TABLE 15. Ablation test results of DEEP-BO with 6 processors. The best performance in each column is marked in bold. In the same column, the results that overlap with the best are highlighted in green.

Diversification	Early Termination	In-progress	Hybrid Transformation	*Expected time* $\mathbb{E}[\tau]$ ($\mu \pm 0.25\sigma$ hours)					
				MNIST-LeNet1	MNIST-LeNet2	PTB-LSTM	CIFAR10-CNN	CIFAR10-ResNet	CIFAR100-CNN
on	on	on	on	1.4 ± 0.3	0.8 ± 0.3	$\mathbf{1.0 \pm 0.1}$	0.8 ± 0.1	$\mathbf{7.7 \pm 0.6}$	$\mathbf{0.5 \pm 0.1}$
on	off	on	on	$\mathbf{1.3 \pm 0.2}$	1.0 ± 0.4	1.2 ± 0.1	0.9 ± 0.2	8.6 ± 0.7	0.8 ± 0.1
on	on	off	on	1.6 ± 0.3	$\mathbf{0.7 \pm 0.1}$	1.1 ± 0.1	0.9 ± 0.2	10.5 ± 0.7	0.7 ± 0.1
on	on	on	off	2.2 ± 0.4	1.0 ± 0.2	1.5 ± 0.2	$\mathbf{0.7 \pm 0.1}$	8.9 ± 0.8	$\mathbf{0.5 \pm 0.1}$
on	off	off	on	1.8 ± 0.3	1.4 ± 0.4	1.4 ± 0.2	1.4 ± 0.2	11.6 ± 1.0	1.1 ± 0.1
on	off	on	off	2.4 ± 0.5	1.5 ± 0.3	1.8 ± 0.2	1.1 ± 0.2	10.5 ± 1.1	0.8 ± 0.1
on	on	off	off	2.7 ± 0.5	1.2 ± 0.3	1.7 ± 0.2	1.0 ± 0.2	10.6 ± 1.1	0.8 ± 0.1
on	off	off	off	2.4 ± 0.4	1.8 ± 0.3	2.1 ± 0.2	1.2 ± 0.2	12.7 ± 1.2	1.1 ± 0.1
off	off	off	off	10.4 ± 2.1	4.6 ± 1.4	2.1 ± 0.2	1.5 ± 0.2	13.0 ± 1.4	1.2 ± 0.1

the easy targets were too easy with six processors. With six processors, DEEP-BO clearly outperformed all the other algorithms for all the experiment cases. The performance gap to the second best algorithm was usually very large (48%–200%). From the viewpoint of robustness, the variance shown in Table 14 can be investigated. For each benchmark, DEEP-BO's variance was much smaller than those from the other algorithms. Even when compared to the second best HPO algorithm in each benchmark category, DEEP-BO's variance gap was 40%–100%.

XI. DISCUSSION

In this section, several important topics are discussed.

A. ABLATION TEST OF DEEP-BO

To understand the effectiveness and importance of the four basic enhancement strategies, we have performed an ablation test with DEEP-BO. The results are shown in Table 15. First of all, we can notice that the performance is severely degraded when diversification is turned off. We used only the GP-EI model when diversification is off, and the degradation was intolerably large for all six benchmarks. This indicates that diversification is the most important enhancement strategy among the four.

When diversification is on, the best performing combination had at least two of the three remaining strategies turned on. Other than that, no particular pattern was noticeable. While none of the three strategies might be strongly required for a high performing solution, the ablation test results indicate that it might be reasonable to use all three strategies.

B. CHOOSING MAXIMUM EPOCH E

Learning rate is known as the most important hyperparameter of DNN for a successful training. When tuning deep neural networks with an HPO algorithm, the maximum epoch E is a very important hyperparameter of the HPO algorithm itself. When E is too small, HPO will not be able to reach a satisfactory performance. When E is too large, HPO will be able to reach a satisfactory performance but it will waste too much time on training each candidate $\mathbf{x}_n$ for many extra epochs. In this regard, ETR is an essential strategy for handling the case of large E. If an HPO algorithm is not equipped with an ETR strategy, one can easily choose a large enough E to make the algorithm's performance worse than any simple HPO with ETR.

For our six benchmarks, we did not want the classic HPO algorithms with no ETR to be unfairly evaluated. Therefore, we had to perform a manual investigation to choose a reasonable E for each benchmark (roughly speaking, 1.5–2 times larger than needed for the best performance). Such an investigation can be efficient if the researcher is familiar with the task and dataset, but it can also be a difficult and time consuming pre-HPO work for a completely new problem.

Ideally, an ETR algorithm needs to be smart enough to learn the rules and be able to handle this maximum epoch problem. In this regard, the CR method in Section V is problematic because the checkpoints were chosen for the particular E values in the benchmark. Obviously, waiting until the first check point of $E/2$ can be unacceptable if E was chosen to be much larger than what we have in benchmarks. If E was much larger than what we have chosen, the ablation test might have indicated that ETR was the most important strategy.

The proper value of E is dependent on the performance target. It is well-known that a larger E is needed when one wishes to reach top-level performance for a DNN. When one wishes to reach an easily achievable performance, E does not need to be as large. Related to this, the relative performance of HPO algorithms can be highly dependent on the performance target. See Fig. 17 for an example.

C. CHOOSING HYPERPARAMETER SPACE $\mathcal{X}$

Besides the maximum epoch E, there are many other hyperparameters. The range needs to determined for each hyperparameter, and other decisions might be relevant as shown in Fig. 18. Ideally, one should be able to list all the relevant hyperparameters and choose the maximum range to make sure that the search space $\mathcal{X}$ contains the configuration that we are looking for. The rest should be efficiently handled by

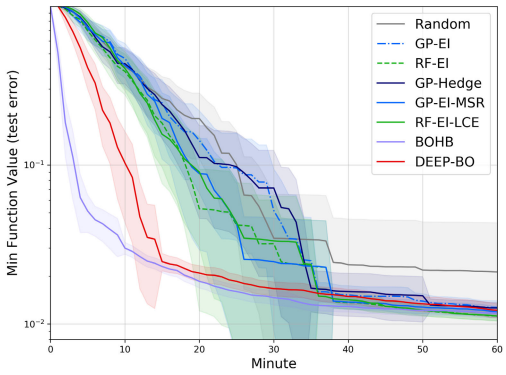


FIGURE 17. Performance of the first one hour for MNIST-LeNet2 benchmark. The shaded areas are 0.25σ variance regions.

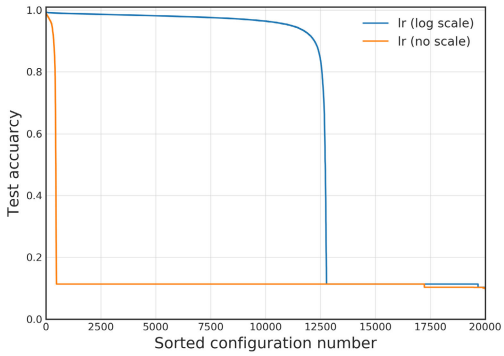


FIGURE 18. Test accuracy distribution of MNIST-LeNet1 benchmark is shown in blue. Learning rate was modeled in log-scale and Sobol sampling was applied to choose the 20,000 configurations. When the range of learning rate was kept the same but a linear scale was used for Sobol sampling, the selected 20,000 configurations ended up with a quite different distribution as shown in orange.

HPO itself. But the state-of-the-art HPO algorithms are not perfect, and their performance can be greatly affected by the choice of $\mathcal{X}$.

An HPO algorithm has its own hyperparameters. E and $\mathcal{X}$ are common ones and there can be other hyperparameters that are specific to each HPO algorithm. Of course, we can think of another layer of HPO for tuning these hyperparameters (two tiers of HPOs), but that would be too time consuming given that it already takes a long time to complete a single episode of the lower tier HPO.

A possible solution is to mix manual investigation and HPO. In this case, HPO will never be fully automatic. Rather, the researcher has the choice on how much to rely on manual investigation and how much on automatic HPO. As of today, many of the researchers heavily rely on manual investigation while utilizing simple grid search or random search as HPO. In such cases, $\mathcal{X}$ is chosen to have at most a few hyperparameters because the simple HPO algorithm is not capable of handling more sophisticated problems. Typically, the manual investigation and HPO need to be iterated many times. Perhaps a less popular approach is to perform a light manual investigation before running HPO, where the goal is to find roughly adequate values of E, $\mathcal{X}$, and others. Then HPO is run for only one time. Obviously, there can be a compromise to repeat the process a few times. While less popular today, we believe the second approach will become much more popular in time as we start to have better HPO algorithms and as we understand HPO of DNN better.

Another possible solution is to fully automate HPO. As mentioned in the introduction, domain knowledge and experience in other DNN optimization should be integrated into HPO algorithms. Techniques like transfer learning and meta learning should be fully integrated into HPO.

D. OVERHEAD OF GP CALCULATION

The evaluation of BO model for choosing $\mathbf{x}_n$ requires computing time. However, the BO computing time is usually ignorable for tuning deep neural networks, because the training time of $f(\mathbf{x})$ is usually much larger than the time needed to evaluate BO model. For GP, however, its calculation time grows as $O(n^3)$ and can become a critical bottleneck if an episode of BO does not terminate after many iterations. To avoid an excessive effect from the $O(n^3)$ increase in calculation time, [47], [48] considered replacing GP with DNN where it requires the DNN to be trained. In our work, we avoided this GP specific problem by limiting the maximum number of observations used as BO's input to be 200. To do so, we have randomly resampled 200 observations from the history whenever more than 200 observations were available. This was a reasonable choice for our study, because our benchmarks required at most several hundreds of observations before terminating. The time for BO model evaluation was included in all of our results.

E. IMPLEMENTATION DETAILS

We have unified several BO algorithms into a single framework. Specifically, GP is based on Jasper Snoek's implementation[1] where the automatic relevance determination (ARD) with Matérn 5/2 kernel and a Monte Carlo estimation of the integrated acquisitions are used. RF is based on scikit-learn ML library.[2] The number of trees and the minimum number of items in a split were set to 50 and 2, respectively.

Although the explanations in this study are based on the accuracy of classification problems, the DEEP-BO implementation can handle any other cost minimization setting. For surrogate space modeling, the categorical hyperparameters were one-hot coded and each hyperparameter was normalized to have values between 0 and 1. For parallelization, we have implemented a flexible solution where any number of GPUs and CPUs over multiple servers can be utilized. Each available GPU is used as an independent training node. The system was built on Web Oriented Architecture (WOA) using Service Oriented Architecture (SOA) and REST as follows [49]:

$$WOA = SOA + WWW + REST$$

REST, which stands for Representational State Transfer, is an architectural style that defines a set of constraints to be used

[1]https://github.com/JasperSnoek/spearmint
[2]https://scikit-learn.org/

for creating Web services that are interoperable between systems on the Internet. Further information can be found from https://github.com/snu-adsl/DEEP-BO/wiki.

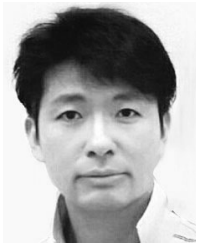
XII. CONCLUSION

In this work, we have investigated four basic enhancement strategies of BO - diversification, early termination, parallelization, and cost function transformation. We have exclusively focused on a deep neural network application where a large number of hyperparameters need to be optimized. Our goal was to identify how BO can be made robust over a variety of deep neural network tuning problems, instead of designing an algorithm that works very well for some problems while failing for some others. In this regard, we have created six benchmark datasets with pre-evaluated performance over a range of hyperparameter configurations such that the empirical evaluation can be made faster. The individual investigations of the four strategies can be summarized as follows. First, it is important to adopt diversification over multiple algorithms because we do not know which one will work well or fail for a given deep neural network tuning problem. As long as diversification is implemented, however, even a simple strategy worked as well as an adaptive strategy. Second, it is essential to be conservative when employing an early termination strategy. The validation performance for a single training can suddenly improve even at a later stage of the deep neural network training, and thus one should not early terminate as often as in the traditional machine learning tasks. Third, when parallelizing over multiple processors or servers, utilizing the validation performance of incomplete training can be very helpful. While simple, we have found this strategy to be important for speed improvement. Lastly, the cost function can be transformed in a heuristic way such that the BO's modeling can be made easier. The transformation, however, requires an overhead of manual tuning before the automated BO can be launched. With these enhancement strategies in mind, we have designed a simple yet very robust BO algorithm called DEEP-BO. DEEP-BO showed top or top-level performance over all the benchmarks. For a single processor, it showed top performance for ten out of twelve benchmarks. Even for the two benchmarks, its performance was very close to the top performance. For six processors, it showed top performance for all of the six benchmarks. In general, DEEP-BO showed a robust performance, and it showed a high performance especially for the difficult targets under the use of multiple processors. Considering that DEEP-BO is merely a collection of four basic enhancement strategies, there might be a large additional room for improving HPO's speed. Some of the issues, such as choosing the hyperparameters to optimize, setting the range of each hyperparameter to consider, and selecting the maximum number of epochs, still remain as the limitations of this work.

REFERENCES

[1] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization," *J. Mach. Learn. Res.*, vol. 13, pp. 281–305, Feb. 2012.
[2] B. Shahriari, K. Swersky, Z. Wang, R. P. Adams, and N. de Freitas, "Taking the human out of the loop: A review of Bayesian optimization," *Proc. IEEE*, vol. 104, no. 1, pp. 148–175, Jan. 2016.
[3] J. S. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl, "Algorithms for hyper-parameter optimization," in *Proc. Adv. neural Inf. Process. Syst.*, 2011, pp. 2546–2554.
[4] J. Snoek, H. Larochelle, and R. P. Adams, "Practical Bayesian optimization of machine learning algorithms," in *Proc. Adv. Neural Inf. Process. Syst.*, 2012, pp. 2951–2959.
[5] D. Maclaurin, D. Duvenaud, and R. P. Adams, "Gradient-based hyperparameter optimization through reversible learning," in *Proc. 32nd Int. Conf. Mach. Learn.*, 2015, pp. 2113–2122.
[6] D. Yogatama and G. Mann, "Efficient transfer learning method for automatic hyperparameter tuning," *Artif. Intell. Statist.*, vol. 33, Apr. 2014, pp. 1077–1085.
[7] V. Perrone, R. Jenatton, M. W. Seeger, and C. Archambeau, "Scalable hyperparameter transfer learning," in *Proc. Adv. Neural Inf. Process. Syst.*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds. Montreal, QC, Canada: Curran Associates, 2018, pp. 6846–6856. [Online]. Available: http://papers.nips.cc/paper/7917-scalable-hyperparameter-transfer-learning.pdf
[8] M. Feurer, B. Letham, and E. Bakshy, "Scalable meta-learning for Bayesian optimization," 2018, *arXiv:1802.02219*. [Online]. Available: http://arxiv.org/abs/1802.02219
[9] M. D. Hoffman, E. Brochu, and N. de Freitas, "Portfolio allocation for Bayesian optimization," in *Proc. 27th Conf. Uncertainty Artif. Intell.*, 2011, pp. 327–336.
[10] D. Golovin, B. Solnik, S. Moitra, G. Kochanski, J. Karro, and D. Sculley, "Google vizier: A service for black-box optimization," in *Proc. 23rd ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining KDD*, Aug. 2017, pp. 1487–1495.
[11] T. Domhan, J. T. Springenberg, and F. Hutter, "Speeding up automatic hyperparameter optimization of deep neural networks by extrapolation of learning curves," in *Proc. 24th Int. Joint Conf. Artif. Intell. (IJCAI)*, Jun. 2015, pp. 1–9.
[12] S. Falkner, A. Klein, and F. Hutter, "BOHB: Robust and efficient hyperparameter optimization at scale," 2018, *arXiv:1807.01774*. [Online]. Available: http://arxiv.org/abs/1807.01774
[13] D. H. Wolpert and W. G. Macready, "No free lunch theorems for optimization," *IEEE Trans. Evol. Comput.*, vol. 1, no. 1, pp. 67–82, Apr. 1997.
[14] K. Eggensperger, M. Feurer, F. Hutter, J. Bergstra, J. Snoek, H. Hoos, and K. Leyton-Brown, "Towards an empirical foundation for assessing Bayesian optimization of hyperparameters," in *Proc. NIPS Workshop Bayesian Optim. Theory Pract.*, Dec. 2013, pp. 1–5.
[15] N. Hansen, A. Auger, D. Brockhoff, D. Tušar, and T. Tušar, "COCO: Performance assessment," 2016, *arXiv:1605.03560*. [Online]. Available: http://arxiv.org/abs/1605.03560
[16] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," 2015, *arXiv:1502.03167*. [Online]. Available: http://arxiv.org/abs/1502.03167
[17] J. Mockus, V. Tiesis, and A. Zilinskas, "The application of Bayesian methods for seeking the extremum," *Toward global Optim.*, vol. 2, pp. 117–129, Dec. 1978.
[18] E. Brochu, V. M. Cora, and N. de Freitas, "A tutorial on Bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning," 2010, *arXiv:1012.2599*. [Online]. Available: http://arxiv.org/abs/1012.2599
[19] C. E. Rasmussen, "Gaussian processes in machine learning," in *Advanced Lectures on Machine Learning*. Berlin, Germany: Springer, 2004, pp. 63–71.
[20] F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Sequential model-based optimization for general algorithm configuration," in *Proc. Int. Conf. Learn. Intell. Optim.* Berlin, Germany: Springer, 2011, pp. 507–523.
[21] H. J. Kushner, "A new method of locating the maximum point of an arbitrary multipeak curve in the presence of noise," *J. Basic Eng.*, vol. 86, no. 1, pp. 97–106, Mar. 1964.
[22] N. Srinivas, A. Krause, S. M. Kakade, and M. Seeger, "Gaussian process optimization in the bandit setting: No regret and experimental design," 2009, *arXiv:0912.3995*. [Online]. Available: http://arxiv.org/abs/0912.3995
[23] L. Li, K. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar, "Hyperband: A novel bandit-based approach to hyperparameter optimization," *J. Mach. Learn. Res.*, vol. 18, no. 1, pp. 6765–6816, 2017.

[24] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proc. IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov. 1998.
[25] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2016, pp. 770–778.
[26] H. Sak, A. Senior, and F. Beaufays, "Long short-term memory recurrent neural network architectures for large scale acoustic modeling," in *Proc. 15th Annu. Conf. Int. speech Commun. Assoc.*, 2014, pp. 338–342.
[27] K. Eggensperger, F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Efficient benchmarking of hyperparameter optimizers via surrogates," in *Proc. AAAI*, 2015, pp. 1114–1120.
[28] A. Klein and F. Hutter, "Tabular benchmarks for joint architecture and hyperparameter optimization," 2019, *arXiv:1905.04970*. [Online]. Available: http://arxiv.org/abs/1905.04970
[29] I. M. Sobol', "On the distribution of points in a cube and the approximate evaluation of integrals," *Zhurnal Vychislitel'Noi Matematiki i Matematicheskoi Fiziki*, vol. 7, no. 4, pp. 784–802, 1967.
[30] T. G. Dietterich, "Ensemble methods in machine learning," in *Proc. Int. Workshop Multiple Classifier Syst.* Berlin, Germany: Springer, 2000, pp. 1–15.
[31] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: MIT Press, 2018.
[32] G. Malkomes, C. Schaff, and R. Garnett, "Bayesian optimization for automated model selection," in *Proc. Adv. Neural Inf. Process. Syst.*, 2016, pp. 2900–2908.
[33] G. Malkomes and R. Garnett, "Automating Bayesian optimization with Bayesian optimization," in *Proc. Adv. Neural Inf. Process. Syst.*, 2018, pp. 5984–5994.
[34] A. Klein, S. Falkner, J. T. Springenberg, and F. Hutter, "Learning curve prediction with Bayesian neural networks," in *Proc. Int. Conf. Learn. Represent. (ICLR)*, Toulon, France, 2016. [Online]. Available: https://openreview.net/forum?id=S11KBYclx
[35] B. Baker, O. Gupta, R. Raskar, and N. Naik, "Accelerating neural architecture search using performance prediction," 2017, *arXiv:1705.10823*. [Online]. Available: http://arxiv.org/abs/1705.10823
[36] L. Prechelt, "Automatic early stopping using cross validation: Quantifying the criteria," *Neural Netw.*, vol. 11, no. 4, pp. 761–767, Jun. 1998.
[37] D. Choi, H. Cho, and W. Rhee, "On the difficulty of DNN hyperparameter optimization using learning curve prediction," in *Proc. TENCON IEEE Region Conf.*, Oct. 2018, pp. 0651–0656.
[38] B. Zoph and Q. V. Le, "Neural architecture search with reinforcement learning," 2016, *arXiv:1611.01578*. [Online]. Available: http://arxiv.org/abs/1611.01578
[39] M. Jaderberg, V. Dalibard, S. Osindero, W. M. Czarnecki, J. Donahue, A. Razavi, O. Vinyals, T. Green, I. Dunning, K. Simonyan, C. Fernando, and K. Kavukcuoglu, "Population based training of neural networks," 2017, *arXiv:1711.09846*. [Online]. Available: http://arxiv.org/abs/1711.09846
[40] E. Contal, D. Buffoni, A. Robicquet, and N. Vayatis, "Parallel Gaussian process optimization with upper confidence bound and pure exploration," in *Proc. Joint Eur. Conf. Mach. Learn. Knowl. Discovery Databases*. Berlin, Germany: Springer, 2013, pp. 225–240.
[41] T. Desautels, A. Krause, and J. W. Burdick, "Parallelizing exploration-exploitation tradeoffs in Gaussian process bandit optimization," *J. Mach. Learn. Res.*, vol. 15, no. 1, pp. 3873–3923, 2014.
[42] J. Wu and P. Frazier, "The parallel knowledge gradient method for batch Bayesian optimization," in *Proc. Adv. Neural Inf. Process. Syst.*, 2016, pp. 3126–3134.
[43] J. González, Z. Dai, P. Hennig, and N. Lawrence, "Batch Bayesian optimization via local penalization," *Artif. Intell. Statist.*, vol. 51, May 2016, pp. 648–657.
[44] T. Kathuria, A. Deshpande, and P. Kohli, "Batched Gaussian process bandit optimization via determinantal point processes," in *Proc. Adv. Neural Inf. Process. Syst.*, 2016, pp. 4206–4214.
[45] Z. Wang, C. Gehring, P. Kohli, and S. Jegelka, "Batched large-scale Bayesian optimization in high-dimensional spaces," 2017, *arXiv:1706.01445*. [Online]. Available: http://arxiv.org/abs/1706.01445
[46] D. R. Jones, M. Schonlau, and W. J. Welch, "Efficient global optimization of expensive black-box functions," *J. Global Optim.*, vol. 13, no. 4, pp. 455–492, 1998.
[47] J. Snoek, O. Rippel, K. Swersky, R. Kiros, N. Satish, N. Sundaram, M. Patwary, M. Prabhat, and R. Adams, "Scalable Bayesian optimization using deep neural networks," in *Proc. Int. Conf. Mach. Learn.*, Jun. 2015, pp. 2171–2180.
[48] J. T. Springenberg, A. Klein, S. Falkner, and F. Hutter, "Bayesian optimization with robust Bayesian neural networks," in *Proc. Adv. Neural Inf. Process. Syst.*, 2016, pp. 4134–4142.
[49] J. Dong, R. Paul, and L.-J. Zhang, *High Assurance Services Computing*. Berlin, Germany: Springer, 2009.



HYUNGHUN CHO received the B.S. degree in computer science and engineering from Chung-Ang University, Seoul, South Korea, in 2005, and the M.S. degree in transdisciplinary studies from Seoul National University, Seoul, in 2017, where he is currently pursuing the Ph.D. degree in transdisciplinary studies.

From 2005 to 2015, he was a Software Engineer with the Software Center, Samsung Electronics. His research interest includes hyperparameter optimization of deep networks using Bayesian optimization and meta-learning.

YONGJIN KIM is currently pursuing the B.S. degree in statistics and the B.A. degree in business administration with Seoul National University, Seoul, South Korea.

From 2015 to 2017, he served in the Korean Army at the Defense Security Command on cyber-intelligence. In 2018, he was a Research Assistant with the Applied Data Science Laboratory, Seoul National University. His research interests include hyperparameter optimization and data science.

EUNJUNG LEE received the B.S. degree in digital media design and applications from Seoul Women's University, Seoul, South Korea, in 2014. She is currently pursuing the Ph.D. degree in transdisciplinary studies with Seoul National University, Seoul.

Her current research interests include data science and time-series data analysis via deep learning, especially meta-learning.

DAEYOUNG CHOI received the B.S. degree in computer science from the Republic of Korea Air Force Academy, Cheongju, South Korea, in 2004, the M.S. degree in electrical and computer engineering from The Ohio State University, Columbus, OH, USA, in 2010, and the Ph.D. degree in transdisciplinary studies from Seoul National University, Seoul, South Korea, in 2019.

He is currently a Network Operations Officer with Korea Air Force. His research interests include artificial intelligence, hyperparameter optimization of deep networks, and representation learning.

YONGJAE LEE received the B.S. degree in computer science and mathematical sciences and the Ph.D. degree in industrial and systems engineering from the Korea Advanced Institute of Science and Technology (KAIST), Daejeon, South Korea, in 2011 and 2016, respectively.

He has been working as an Assistant Professor with the Ulsan National Institute of Science and Technology (UNIST), Ulsan, South Korea, since 2018. He is particularly interested in applying optimization and data science techniques to enhance individuals' financial health. He has published several academic articles in the fields of operations research and finance. His research topics include optimization, data science, and financial engineering.

WONJONG RHEE (Fellow, IEEE) received the B.S. degree in electrical engineering from Seoul National University, Seoul, South Korea, in 1996, and the M.S. and Ph.D. degrees in electrical engineering from Stanford University, Stanford, CA, USA, in 1998 and 2002, respectively.

From 2001 to 2003, he was a Research Staff with ArrayComm. From 2004 to 2013, he was with ASSIA Inc., as a Founding Member and later as an ASSIA Fellow. He is currently an Associate Professor with the Department of Transdisciplinary Studies, Seoul National University. His general research interests are in the fields of data science and machine learning, where information theory, optimization, and signal processing are often utilized.