HyperDrive: Exploring Hyperparameters with POP Scheduling

Jeff Rasley
Brown University

Yuxiong He
Microsoft

Feng Yan
University of Nevada, Reno

Olatunji Ruwase
Microsoft

Rodrigo Fonseca
Brown University

Abstract

The quality of machine learning (ML) and deep learning (DL) models are very sensitive to many different adjustable parameters that are set before training even begins, commonly called hyperparameters. Efficient hyperparameter exploration is of great importance to practitioners in order to find high-quality models with affordable time and cost. This is however a challenging process due to a huge search space, expensive training runtime, sparsity of good configurations, and scarcity of time and resources. We develop a scheduling algorithm POP that quickly identifies among *promising*, *opportunistic* and *poor* configurations of hyperparameters. It infuses probabilistic model-based classification with dynamic scheduling and early termination to jointly optimize quality and cost. We also build a comprehensive hyperparameter exploration infrastructure, HyperDrive, to support existing and future scheduling algorithms for a wide range of usage scenarios across different ML/DL frameworks and learning domains. We evaluate POP and HyperDrive using complex and deep models. The results show that we speedup the training process by up to 6.7x compared with basic approaches like random/grid search and up to 2.1x compared with state-of-the-art approaches while achieving similar model quality compared with prior work.

CCS Concepts • **Computer systems organization → Cloud computing**; • **Computing methodologies → Machine learning approaches**;

Keywords Hyperparameter exploration, cluster scheduling

ACM Reference format:
Jeff Rasley, Yuxiong He, Feng Yan, Olatunji Ruwase, and Rodrigo Fonseca. 2017. HyperDrive: Exploring Hyperparameters with POP Scheduling. In *Proceedings of Middleware '17, Las Vegas, NV, USA, December 11–15, 2017,* 13 pages.
DOI: 10.1145/3135974.3135994

1 Introduction

In recent years many machine-learning frameworks have been introduced to help developers build and train machine learning models for solving different artificial intelligence tasks across supervised, unsupervised, and reinforcement learning domains [3, 6, 9, 10, 15, 31]. The task performance of trained models is very sensitive to many different adjustable parameters, called hyperparameters, which are configured prior to training a model. Examples of these hyperparameters include learning rate (in many models), number and size of hidden layers in a deep neural network, number of clusters in k-means clustering, and many more. Hyperparameter exploration searches across different configurations of a model, where each *configuration* represents a specific set of hyperparameter values. Its goal is to find good configurations that optimize the model performance (e.g., high accuracy, low loss, high reward) with affordable time and cost. This exploration involves two related problems: generating candidate configurations from the large space of hyperparameter settings, and actually scheduling and running these configurations.

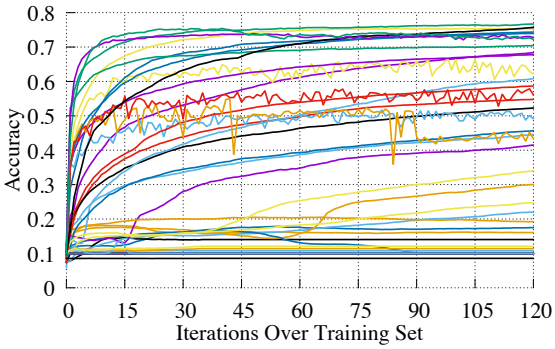


Figure 1. Performance of 50 randomly selected supervised-learning hyperparameter configurations.

Efficient hyperparameter exploration is of great importance to practitioners in order to improve model performance, reduce training time, and optimize resource usage. This is especially critical when training modern deep and complex learning models with billions of parameters and millions of training samples on cloud resources. To reach a desired training target, efficient hyperparameter exploration means shorter training time, lower resource consumption, and thus lower training costs.

However, it is challenging to design effective scheduling frameworks and algorithms that efficiently explore hyperparameter values while obtaining high model performance and optimizing time and resource costs. The first key reason is the size of the hyperparameter search space and the expensive training process for each individual configuration. Figure 1 shows the model performance (task accuracy) of 50 configurations as a function of iterations over a training dataset for a moderate-size image classification application CIFAR-10 [20] (detailed experimental setup is presented in §6). Each line in the plot represents a unique configuration. Here we present only 50 configurations in the plot for clear presentation. Each configuration needs to run about 120 iterations with each iteration taking about one minute. To fully explore only 50 configurations, we need over 4 days of computing. Commonly, models require exploring many more configurations to find high

Middleware '17, Las Vegas, NV, USA

978-1-4503-4720-4/17/12…$15.00
DOI: 10.1145/3135974.3135994

performing hyperparameters. For example, our CIFAR-10 model has 14 hyperparameters (such as learning rate, momentum, and weight decay) and most have a continuous range to explore, which results in hundreds or thousands (or more) possible configurations to explore. This problem is even worse when considering larger training models and datasets, for example, prior work has shown that a high-quality ImageNet22k image classification model can take up to ten days to train to convergence using 62 machines [8]. Therefore exhaustive search is simply not practical.

A second reason is that, for many models in practice, only few configurations lead to high performance while a majority of configurations perform very poorly. The results in Figure 1, for example, show that only three configurations are able to exceed 75% accuracy (which is considered reasonable accuracy for this type of simple CIFAR-10 model that doesn't do any data augmentation and/or intensive preprocessing steps [1]), while the majority of configurations are not able to exceed 20% accuracy. Therefore, simple and popular approaches such as grid and random hyperparameter generation bet heavily on luck, which results in very inefficient discovery of high performing configurations under reasonable cost and time constraints.

Furthermore, optimizing hyperparameter tuning involves many other factors, such as incorporating different application domain goals (e.g., supervised, unsupervised, and reinforcement learning) and applicability to different DL/ML frameworks (e.g., Tensorflow, Caffe, and CNTK). It is challenging to design an effective framework to support such a wide range of usage scenarios.

Recent work [7, 14, 24] has moved beyond grid-based hyperparameter generation with adaptive techniques using Bayesian optimization, which assigns higher probability to areas of the hyperparameter space that contain likely-good or unknown configurations. These works do not, however, address how to run these configurations. For example, how long should each configuration run? Prior work [7, 14, 18, 24, 28] executes each configuration to the same maximum iteration (which can be a large number of iterations), typically ignoring the fact that some configurations could have shown their intrinsic value much earlier. As shown in Figure 1, with basic domain knowledge, one can quickly tell that many configurations do not learn at all, with accuracy similar to random (10% accuracy in this case), which can be identified within few training iterations and terminated early to save resources. In addition, should all running configurations take the same amount of resources? Clearly, that is not the best way to assign resources since the execution progress of the configurations could have offered many insights on how well the configurations are likely to perform, thus deserving different amounts of resources. This scheduling problem of how to effectively map different configurations across the time and space dimension of resources is largely unattended by prior work.

Our work focuses on designing an efficient scheduling algorithm and an effective system framework for hyperparameter exploration. Our scheduling algorithm, POP, dynamically classifies configurations into three categories: Promising, Opportunistic, and Poor, and uses these in two major ways. First, along the time dimension, we quickly identify and terminate poor configurations that are not learning, incorporating application-specific input from model owners. For example, classification tasks have known non-learning random performance values which can be used to prune jobs. Second, along the space dimension, we use an explore-exploit strategy. We classify promising configurations that are more likely to lead to high task accuracy and prioritize them with more resources while giving configurations that are still only *potentially* promising some chances to run – these are what we call opportunistic configurations. Unlike prior work [25] that uses instantaneous accuracy only to identify a fixed set of promising configurations, we incorporate the trajectory of full learning curves and leverage a probabilistic model to predict expected accuracy improvement over time as well as prediction confidence [11]. The classification and resource allocation between promising and opportunistic configurations is dynamic, adjusting the ratio of exploration and exploitation when we observe more predicted and measured results. At the early stage of training, when there is little history information and prediction confidence is typically low, allocating more resources for exploration helps training efficiency. Conversely, at later stages, confidence is higher, and allocating more resources for exploitation can yield higher rewards.

We also design a framework, HyperDrive, which serves as a generic framework for hyperparameter exploration. HyperDrive largely decouples the scheduling policy for candidate configurations from the type of model and/or framework. It provides an API that supports not only our POP scheduling algorithm, but also existing and new ones. It also supports different learning frameworks, such as Caffe [15], CNTK [31], and TensorFlow [3], and learning domains, such as supervised and reinforcement learning. Lastly, it supports model-owner-defined metrics and inputs to improve scheduling efficiency.

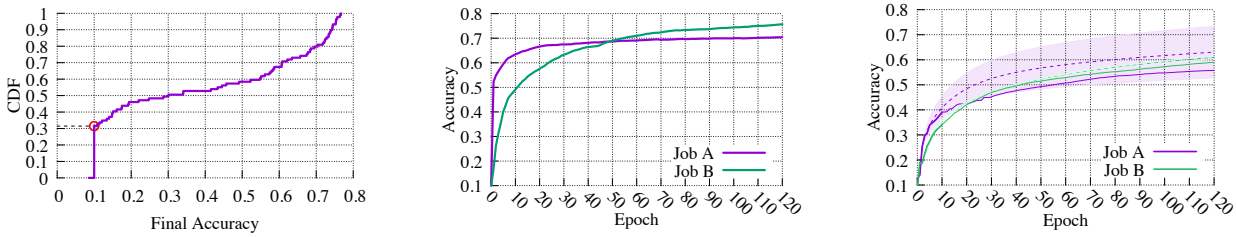
This paper makes the following contributions:

- We develop an efficient scheduling algorithm that infuses probabilistic model-based configuration classification with dynamic scheduling and early termination (§2 and §3).
- We develop an effective system framework, we call HyperDrive, that not only facilitates our proposed scheduling algorithm, but supports existing and future scheduling algorithms, and works with different domains and machine learning frameworks (§4 and §5).
- We present extensive evaluation of our scheduling algorithm and framework using workloads from both supervised and reinforcement learning domains. We show our proposed approach outperforms the basic approaches like random/grid search by up to 6.7x and outperforms state-of-the-art scheduling techniques by up to 2.1x and that our framework is practical (§6 and §7).

2 Design Principles of Scheduling Algorithm

Hyperparameter exploration is challenging as there are many configurations but often much less time and fewer resources. For deep and complex models, it is common that only a small number of configuration choices lead to high quality results. Thus, significant amounts of time and resources could be wasted in searching for good configurations if considerable attention is not paid to search efficiency. To enable efficient model exploration and scheduling, we follow three key design principles: i. early identification and termination of *poor configurations* that are either not learning or learning very slowly; ii. early classification of *promising configurations* that are more likely to lead to high task performance among

[1] http://rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html



(a) Final validation accuracy distribution of 90 randomly selected CIFAR-10 configurations.

(b) Final learning curves for two different configurations A and B.

(c) A has higher predicted accuracy but lower confidence, while B has better final accuracy.

Figure 2. CIFAR-10 validation accuracy distribution, along with learning curves for two configurations A and B.

the remaining group of *opportunistic configurations*; iii. prioritized execution of promising configurations by devoting more resources to them without starving opportunistic configurations; striking a desired balance between *exploitation* of promising configurations and *exploration* of opportunistic configurations. Our search algorithm POP achieves these design principles by solving the following three challenges.

2.1 Identify poor configurations early

Early detection of configurations with poor learning performance can reduce wasted resources during model search. Figure 1 shows that in the search for high-quality *CIFAR-10* models a significant portion of possible model configurations either do not learn or learn very slowly during the entire training process. We present the final validation accuracy distribution of different configurations in Figure 2c. The red circle shows the percentile of configurations that achieve below the random validation accuracy of 10%[2] We can see there are 32% of configurations with poor validation accuracy, i.e., at or below random validation accuracy. Such a significant amount of configurations with poor validation accuracy demonstrates the importance of identifying and terminating poor configurations as early as possible to reduce wasted time and resources.

An efficient way to identify *poor configurations* early is to incorporate domain knowledge from the model owner. For example, in many supervised-learning tasks it is common for poor hyperparameter values to result in models that only achieve "random" validation accuracy (which is defined by the task), an example of this can be seen in Figure 1, where the task is forced to choose a label from 10 categories, therefore many configurations only achieve random validation accuracy around 10%. The search algorithm can incorporate this knowledge to improve search by terminating configurations early if they fail to escape this "random" validation accuracy threshold after a few iterations. Similarly, a user can incorporate early termination for many reinforcement-learning models due to a common "not learning" range which can be determined based on the environment being trained on. In addition it is common for reinforcement-learning tasks to also have unique "solved" conditions that can be incorporated into a search algorithm, for example a task may only be considered "solved" when it sustains a certain reward for some number of iterations.

2.2 Classify promising configurations early and judiciously

To classify *promising configuration* early and judiciously, one effective way is to develop an accurate methodology for predicting future task performance. There are three important questions to answer for developing an accurate prediction methodology.

a) Would the most recent performance alone be sufficient? As an example, Figure 2b shows full validation accuracy curves for two configurations A and B. At the early stage, i.e., before the 50th epoch, A's validation accuracy is higher than B's. However, the final validation accuracy of B is higher than A, thus B *overtakes* A. If we simply rely on the most recent performance, we will not discover that B is the most promising configuration until after the 50th epoch and thus waste a lot of resources. We observe this overtake phenomenon sometimes can even be more pronounced than seen in Figure 2b. Therefore, in order to classify *promising configuration* early, the most recent performance alone, as used in prior work [25], is not enough. In order to make effective predictions, we use a probabilistic model to predict expected future task performance by incorporating partial task performance history, i.e., the task's learning curve.

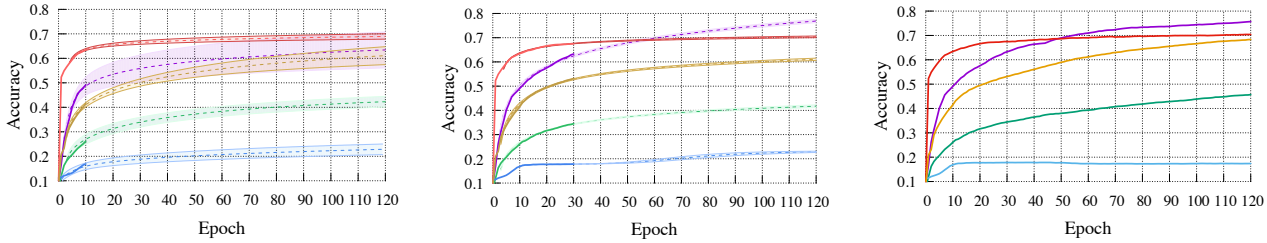
b) Would predicting expected future task performance alone be sufficient? We answer the question by showing an example using Figure 2c. The dotted lines show A and B's expected validation accuracy and the solid lines show the measured validation accuracy. The results indicate A's expected validation accuracy is higher than B at epoch 10 but with much larger variance and lower confidence than B (the shadow represents the confidence intervals). However, in the final validation accuracy, B is actually higher than A, which indicates expected future validation accuracy alone can be misleading and we need to assess the quality of the prediction. To quantify the prediction quality, we calculate the confidence of the prediction.

c) Would a static threshold be sufficient to decide a *promising configuration*? One way to classify promising configurations is by using a static threshold for the probability of achieving target task performance, e.g., if the probability is higher than the threshold, the configuration is promising. However, the problem with this approach is if the threshold is too high, it becomes difficult to identify promising configurations early in the training process. On the other hand, if the threshold is too low, we may classify too many configurations as promising and results in an ineffective way to allocate resources. Therefore, when determining the threshold, we need to take into consideration both the characteristics of the model and the available resources.

2.3 Resource allocation between promising and opportunistic configurations

The key insight here is that static resource allocation between promising and opportunistic configurations is insufficient as configurations can change status between promising, opportunistic, and poor over time. Figure 3 gives an example of the prediction

[2]Random accuracy is defined as 10% here due to CIFAR-10 having 10 categories, thus a random guess yields a 10% chance of being correct.



(a) Prediction at the 10th epoch. **(b)** Prediction at the 30th epoch. **(c)** Final validation accuracy curves.

Figure 3. Predicted and measured validation accuracy curves of multiple configurations varying over time.

confidence at three different stages during training. For example, at the beginning stage Figure 3a, there is little trajectory information and thus low confidence to differentiate configurations: all active configurations are classified as opportunistic and all resources are designated opportunistic. As training progresses, more promising configurations emerge thus we allocate more resources for promising configurations. At later stages, it is possible that one or several configurations have very high confidence to achieve their target, therefore we can allocate resources to them in a much more aggressive way or even use an "all-in" strategy. Therefore, the configuration classification and resource allocation should be coordinated based on the progress of training.

3 Scheduling Algorithm POP

The search for high-quality models, from a candidate set of model configurations, typically involves multiple iterations of training each model configuration with a training dataset and evaluating model performance against a validation dataset. The objective of POP is to improve the efficiency of discovering high performing model configurations: minimize the time to find a configuration satisfying a target performance. It can also be used for finding configurations with the best performance within a time budget, which is a corresponding dual problem. To achieve efficient search and scheduling, we need to promptly and accurately identify configurations that are more likely to result in high-quality models i.e., *promising configurations* (§3.1). In addition, we need an efficient resource allocation strategy that prioritizes promising configurations. Here we develop an infused methodology incorporating both configuration classification and scheduling (§3.2).

3.1 Configuration Classification

As discussed in §2, in order to classify configurations, POP needs to consider both the expected final task performance and the quality of prediction based on performance history information. Recall that the objective is to minimize the time to find a configuration that achieves a target performance. Therefore, at any point of time in training, the configurations with the smallest expected remaining time to achieve the target performance and with high prediction quality are considered promising configurations. To classify configurations accurately and promptly, an accurate estimation model of the expected remaining time and a methodology to evaluate the prediction quality is key.

3.1.1 Expected Remaining Time Estimation

We develop a probabilistic approach for estimating the expected remaining time for a given configuration.

In many learning domains it is common to periodically evaluate a model's performance (e.g., validation accuracy in supervised-learning). The frequency at which a model's performance is evaluated is often at the end of a training epoch, so to compute the expected remaining time, we can first compute the expected number of remaining epochs and then multiply with the average epoch duration[3]. The idea is to compute the probability of achieving the target performance at future epochs and then use a probability mass function to estimate the expected number of remaining epochs.

Problem formulation: We define the problem as predicting the expected remaining time for a given configuration to achieve the target accuracy. The following parameters are required from users as input parameters[4]:

- T_{max}: the maximum experiment time a user can tolerate;
- y_{target}: a target model performance;

These two parameters are based on the user's domain knowledge, therefore values provided by experts should ideally lower estimation overheads and improve search efficiency compared to values provided by beginners. As we first do prediction based on epochs, we compute the maximum number of remaining epochs M_i for a given configuration i as $M_i = (T_{max} - T_{pass})/Epoch_i$, where T_{pass} is the measured time duration that has passed from the beginning of the experiment, and $Epoch_i$ is the measured average epoch duration. We define $p1, p2, ..., p_m, ..., P_M$ as the probability that the target performance can be reached in $1st, 2nd, ..., Mth$ epoch and p as the prediction confidence, which is defined as the probability that a configuration can achieve the target performance within T_{max}, i.e., $p = p1 + p2 + ... + p_m + ... + P_M$. The prediction model output is the expected remaining time ERT_i for configuration i to achieve target accuracy y_{target}.

Model performance prediction: To predict future model performance, we rely on the configuration's learning curve. More specifically, we compute the probability $P(m)_i$ of configuration i reaching a model performance y after epoch m in the future based on the observed validation performance history of the configuration, as follows:

$$P(m)_i = P(y(m)_i \geq y | y(1:m-1)_i), \quad (1)$$

where $y(m)_i$ is the predicted performance after epoch m for configuration i and $y(1:m-1)_i)$ represents the observed performance of configuration i from epoch 1 to $m-1$. We leverage a probabilistic learning curve model proposed in prior work [11] to compute $P(y(m)_i \geq y | y(1:m-1)_i)$.

The learning curve prediction model we use relies on a weighted combination of 11 different parametric models (e.g., vapor pressure, Weibull, Janoschek) and uses Markov Chain Monte Carlo (MCMC)

[3] An epoch represents training over an entire training data set once. Epoch durations are assumed to be roughly constant, see §9 for more details.
[4] See §9 for a discussion on these user parameters.

inference to predict possible values of these weights based on the observed partial performance curve. This probabilistic model is then used to compute the probability $P(m)_i$ of each configuration periodically online, which allows the scheduling policy to see a global view of performance across all active configurations. Due to the non-deterministic nature of MCMC inference, we define a prediction accuracy PA to be the standard deviation across all MCMC samples. Further discussion of our implementation and optimization of the learning curve prediction model is discussed later in §5.2.

Modeling expected remaining time and prediction confidence: To model the expected remaining training epochs x_i for configuration i to achieve the target performance y_{target}, we compute the probability that the target performance can be reached at the $1st, 2nd, ..., mth, ..., Mth$ epoch respectively[5]. According to the definition of the probability mass function defined based on accumulative distribution, we have:
$p1 = P(y(1)_i \geq y_{target})$,
$p2 = P(y(2)_i \geq y_{target}) - P(y(1)_i \geq y_{target})$,
...
$p_m = P(y(m)_i \geq y_{target}) - P(y(m-1)_i \geq y_{target})$,
...
$p_M = P(y(M)_i \geq y_{target}) - P(y(M-1)_i \geq y_{target})$.
Thus the expected number of remaining epochs x_i for configuration i can be estimated as:

$$x_i = 1 * p1 + 2 * p2 + ... + m * p_m + ... + M * p_M \quad (2)$$

Therefore the expected remaining training time ERT_i for configuration i is:

$$\begin{aligned} ERT_i &= x_i * Epoch_i \\ &= (1 * p1 + 2 * p2 + ... + m * p_m + ... \\ &\quad + M * p_M) * Epoch_i \end{aligned} \quad (3)$$

Ideally, the probability $p1, p2, ..., p_m, ..., P_M$ should sum to 100% (i.e., $\sum_{m=1}^{M}(p_m) = 1$). But in reality, we do not need to sum further if the expected remaining training time is larger than the maximum experiment time duration that user can tolerate, i.e., $ERT_i > T_{max} - T_{pass}$. In other words, we stop summing further for p_m and set $ERT_i = T_{max} - T_{pass}$ since the search algorithm will not run further than $T_{max} - T_{pass}$. Therefore, the probability $p1, p2, ..., p_m, ..., P_M$ may not sum up to 100% (i.e., $\sum_{m=1}^{M}(p_m) \leq 1$). Here we define the probability sum as the prediction confidence p as the higher the probability sum, the more certain the expected remaining training time[6].

Classify configurations: We define p_{thred} as a threshold for prediction confidence of classifying promising configurations: if $p \geq p_{thred}$, then the configuration is classified as a *promising configuration*, otherwise the configuration is classified as an *opportunistic configuration* or *poor configuration*. To distinguish between *opportunistic configuration* and *poor configuration*, we rely on the domain knowledge as explained in §2. The remaining question is how to determine the classification threshold p_{thred}? As explained in §2, a static threshold is insufficient and must consider available resources to determine the threshold value. Next, we develop an infused methodology for determining the threshold and making judicious scheduling decisions.

[5]Note M is actually M_i because it is configuration-specific. We omit i here for ease of presentation.
[6]Note p is actually p_i because it is configuration-specific. We omit i here for ease of presentation.

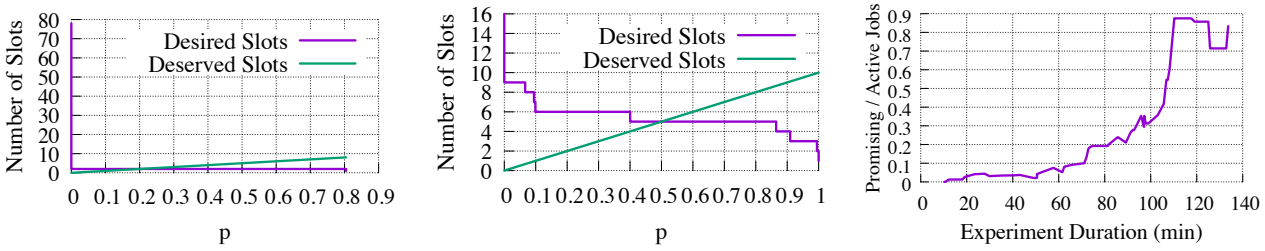
3.2 Infused Classification & Scheduling Methodology

Ideally, if we could perfectly predict future model performance and expected remaining time of model configurations, we could allocate all resources to the most promising configuration(s) (i.e., with shortest expected remaining time). However, in practice, since predication cannot be 100% accurate, we need to allocate resources based on the prediction quality as well as the available resources. The proposed search algorithm employs both an exploration and exploitation approach for resource allocation. We allocate dedicated resources to exploit promising configurations as they are more likely to produce high quality results; we also reserve resources for exploring the opportunistic configurations as when more information is available, i.e., after more epochs of execution, they may become promising. Therefore, the available resources are divided into two pools accordingly: a *promising resource pool* and an *opportunistic resource pool* (we do not allocate resources for *poor configurations*). We dynamically adjust the resource division based on the computed prediction confidence and measured prediction accuracy. In other words, during training, we adjust the ratio of resources dedicated for exploitation versus exploration as we observe more predicted and measured results.

Assume S is the total number of slots (e.g., machines, GPUs), which is typically much smaller than the total number of configurations. Let $N_{satisfying}(p)$ denote the number of configurations with confidence p that can achieve the target performance within the maximum experiment time that the user can tolerate, or equivalent to $\{i | ERT_i(p) \leq T_{MAX}\}$, where $ERT_i(p)$ is the expected remaining training time for configuration i to achieve the target performance with confidence p (an extended definition of the expected remaining training time ERT_i). Naturally, a large $N_{satisfying}(p)$ value under the same confidence p corresponds to a large number of promising configurations. Also, high values of confidence p typically results in small values of $N_{satisfying}(p)$.

To decide the effective number of slots $S_{effective}$ for promising configurations, we look at the problem from two angles, their *desired* number of slots $S_{desired}$ and the *deserved* number of slots $S_{deserved}$. For any given confidence p, we consider those configurations satisfying the confidence as promising, i.e., the number of promising configurations is $N_{satisfying}(p)$. Assume each promising configuration gets a dedicated number of slots k. For example, if a slot represents a machine, a sequential execution of a configuration has $k = 1$ What does a sequential execution of a config mean?. The desired number of slots for promising configurations $S_{desired}(p)$ is equal to $N_{satisfying}(p) \times k$. On the other hand, the total number of slots is limited by resource availability, and the number of slots promising jobs *deserve* is related to the confidence p — the higher the confidence, the more resources they shall get. We calculate the desired number of slots as $S_{deserved}(p) = S \times p$. The actual resources that promising jobs shall receive must be both desired and deserved, and thus $S_{effective}(p) = \min(S_{desired}(p), S_{deserved}(p))$.

Among all p values, we choose the one that maximizes $S_{effective}(p)$, i.e., the number of slots for promising configurations is equal to $S_{promising} = argmax_p(S_{effective}(p))$. These slots are assigned to run promising configurations with dedicated resources. The remaining



(a) Desired slots are low early on due to low confidence.

(b) Desired slots are high later on due to higher confidence.

(c) Ratio of promising slots increases over time.

Figure 4. Allocation of resources over an experiment's lifetime.

slots are allocated to the *opportunistic resource pool*, where the resources are equally shared between opportunistic configurations, e.g., in a round robin manner.

Figure 4a and Figure 4b show the number of desired slots and deserved slots under different prediction confidence values p. Figure 4a shows a snapshot taken in the early stage of an experiment (after about 20 min) when most of the p values are very small due to limited history information available. Figure 4b is a snapshot taken at a later stage of the experiment (after almost 2 hrs). From both figures, we can see that: (1) $S_{desired}(p)$ is a monotonically non-increasing function of p, since when p increases, $N_{satisfying}(p)$ will not increase and can only decrease. (2) $S_{deserved}(p)$ is a monotonically increasing function of p, since higher p deserves more resources. The cross point of the desired slot and deserved slot curves maximizes $S_{effective}$, which corresponds to the number of slots given to promising configurations $S_{promising}$.

To further understand how our resource allocations change over time, consider Figure 4c which shows how the ratio of resources allocated for exploitation (the *promising resource pool*) versus resources for exploration (the *opportunistic resource pool*) change over the experiment's lifetime. It is clear that at the early stage a higher share of resources are used for exploration, however later the share of exploitation resources increase significantly as we improve our overall prediction quality.

4 HyperDrive Design

In designing the POP scheduling algorithm, two things became clear: first, the concerns of the policy are largely independent of the exact learning domain or framework, provided the scheduler can extract the right information from the tasks; second, to efficiently schedule the learning tasks, we needed a slightly richer interface than that of traditional task schedulers such as YARN or Spark. For example, we needed the ability to suspend and resume tasks to effect resource allocation, and to convey to the policy model-owner-defined metrics.

Our HyperDrive framework addresses these observations, and is a step towards providing a separation between hyperparameter search algorithms and their runtime environment.

4.1 Design Considerations

We designed HyperDrive with the following goals in mind:

Support and enable reuse of existing and future search and scheduling algorithms. As there will be new or customized hyperparameter optimization methodologies, the scheduling framework shall be flexible enough to allow users to swap in and out different search and scheduling algorithms.

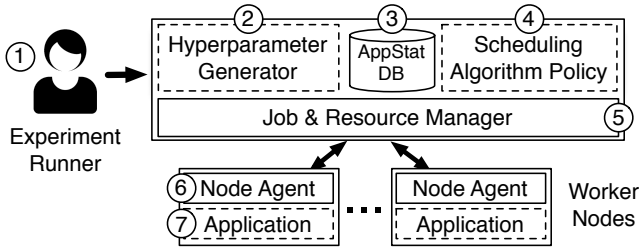


Figure 5. HyperDrive architecture

Monitor and report job status to support dynamic resource adjustment and early termination. To support judicious scheduling decisions, such as dynamic resource adjustment and early termination, the framework should be able to monitor and report current and history job status.

Support different learning domains by allowing inputs from model owners. The scheduling framework should support different learning domains (e.g., supervised, unsupervised, reinforcement learning) by allowing model owners to specify domain specific requirements.

Support different learning frameworks. There are many different learning frameworks in use today, such as CNTK, TensorFlow, Theano, Caffe, MXNet, etc. The scheduling framework should be learning framework agnostic, i.e., not bound to a specific one.

4.2 HyperDrive Framework

Figure 5 shows HyperDrive architecture, which is described below. Numbers in circles correspond to the components in the figure.

Job and Resource Management The core job and resource management components of HyperDrive (⑤) provide the basic ability of executing jobs on remote machines.

The Resource Management (RM) component is responsible for keeping track of currently allocated and idle resources (e.g., machines, GPUs). We leave its description short for brevity and its simplicity. However, if executing HyperDrive in a cloud environment this piece is customized for the specific environment (e.g., reserve an Azure/AWS instance). The RM provides a simple API to other components:

- $reserveIdleMachine() \rightarrow machineId$
- $releaseMachine(machineId)$

The Job Manager (JM) provides the ability to start, resume, suspend, and terminate jobs on specific machines obtained from the RM. It keeps track of each job's state based on the actions performed on it. The JM provides the following API to other components:

- $getIdleJob() \rightarrow jobID$
- $startJob(jobID, machineID)$

- $resumeJob(jobID, machineID)$
- $suspendJob(jobID, machineID)$
- $terminateJob(jobID, machineID)$
- $labelJob(jobID, <float> priority)$

Suspend and resume support is used to enable flexible scheduling of jobs, which means that the framework must be able to train a model for an unspecified amount of time, suspend training, and then resume training later on any machine associated with the experiment. Suspend and resume requires that training state is saved and synchronized with the AppStat database (③), which allows any machine to recieve the state and resume training. The JM also provides the ability to label a job with a priority value, which the JM uses to order idle jobs. Priority ordering is especially important when adding a suspended job to the list of idle jobs. If no priority is given then idle jobs are ordered according to FIFO order.

Node Agent The Node Agent (⑥) is daemon running on a worker machine responsible for job execution and acting as an intermediary between the HyperDrive scheduler and the training application (⑦). All Job Manager calls from HyperDrive that deal with job execution are received and executed by a Node Agent. In addition all application statistics reported by the training application are sent to its local Node Agent and then forwarded to the HyperDrive scheduler.

Scheduling Algorithm Policy A user-provided Scheduling Algorithm Policy (SAP) (④) is written in an imperative style using the following three HyperDrive *up-call* events:

- $AllocateJobs()$
- $ApplicationStat(jobEvent)$
- $OnIterationFinish(jobEvent)$

AllocationJobs is triggered on detection of an idle resource to allow the SAP to schedule a new job on that resource. *ApplicationStat* is triggered on receiving application stats (e.g., accuracy) from the training job to enable the SAP to store or process the data as appropriate. Lastly, *OnIterationFinish* is triggered when a training iteration finishes to allow the SAP to decide whether to continue, suspend, or terminate the job, or collect additional statistics (e.g., iteration timings). We find that these simple scheduling primitives allow us to write a diverse collection of SAPs that cover some prior work and our own scheduling algorithm as described in §2.

A SAP is notified when a job finishes an iteration. Then it makes a decision whether to continue training the job or terminate/suspend the job. By default, HyperDrive uses a schedule-as-it-goes approach to maximize resource usage since configurations with short epoch durations do not need to wait for those with long durations. HyperDrive also supports barrier-like epoch scheduling, which some SAPs may prefer as it can help explore job configurations in a breadth-first-style (i.e., executing many jobs for a short period of time in each round). Barrier-like epoch scheduling can be achieved by allowing the SAP to suspend jobs at every epoch boundary.

Default SAP The default SAP simply greedily allocates idle jobs to idle machines, which is implemented by starting as many idle jobs (via $startJob$) as there are idle machines. This policy ignores all application statistics and iteration finish up-calls, but provides a simple base for more advanced SAPs.

Hyperparameter Generator The Hyperparameter Generator (②, HG) is responsible for generating specific parameter values within ranges specified by the experiment runner. The generator implementation is pluggable as long as it provides the following API:

- $createJob() \rightarrow (jobID, hyperparameters)$
- $reportFinalPerformance(jobID, performance)$

We consider the use of several different HG techniques, which are built separate from HyperDrive itself. Along with more complicated approaches we have built simple random and grid search techniques as HGs, where a user-provides the parameter names and search ranges. The HG then selects new random or grid values upon each call to $createJob$. In these approaches the $reportFinalPerformance$ call is not used.

Adapative techniques (e.g., Bayesian optimization) are popular alternatives to random and grid search and used in in frameworks like HyperOpt [18], Auto-WEKA [28], Spearmint [24], and GPyOpt [5]. These frameworks generate new hyperparameter values based on the observed performance of previous values. These type of approaches can be plugged into HyperDrive with the use of a shim that exposes the HG API.

AppStat Database The application statistics database (AppStatDB ③) is used to store and retrieve model-generated application statistics such as performance stats (e.g., accuracy, reward), epoch duration, etc. In addition the AppStatDB stores model state used to enable suspend and resume training across machines. The AppStatDB is used to sharea state between the SAP, Hyperparameter Generator, and training job itself.

Experiment Runner (Client) The Experiment Runner (①) is responsible for specifying the following items when running an experiment with HyperDrive:

- Search Algorithm Policy to use (with any SAP specific parameters)
- Hyperparameter generation technique along with parameter names and search ranges
- Model training files to run on remote machines
- Total number of machines

5 HyperDrive Implementation

We implemented a HyperDrive prototype based on the components described in §4 in Python. All communication between the scheduler, node agents, and applications is done via GRPC [1]. We implemented two HyperDrive application libraries in Python and C++ that we use to support Theano, Keras, TensorFlow, and Caffe.

5.1 Suspend & Resume Support

The ability to suspend and resume training jobs is an important feature of HyperDrive. Typical learning frameworks provide functionality to snapshot and restore training job state, which simplifies implementing suspend and resume in HyperDrive. However, if a model uses state external to the underlying framework (e.g., Python models using TensorFlow/Theano) it can be difficult to snapshot all framework and model state together for simple suspend/resume. In this case, of mixed state utilize CRIU [2], a tool for snapshotting and restoring arbitrary application state, to implement suspend and resume.

5.2 Learning Curve Prediction

We implemented the learning curve prediction model by adapting a public implementation [7] of the model from [11]. The overhead of running the unmodified learning curve prediction model for a single learning curve can time consuming (several minutes). We

[7] https://github.com/automl/pylearningcurvepredictor

identify three optimizations for performing parallel learning curve prediction in HyperDrive.

Reduce total MCMC samples. At its core the learning curve prediction module uses a computationally expensive Markov Chain Monte Carlo (MCMC) inference technique to predict future training performance. In order to reduce the total time to create the learning curve prediction model we reduced the total number of MCMC samples from 250,000 (nwalkers=100, nsamples=2500) to 70,000 (nwalkers=100, nsamples=700). This reduced our learning curve prediction time by over 2x without significant degradation in our policy's performance.

Distributed Curve Prediction. A simple implementation of HyperDrive would run all learning curve prediction at the central scheduler. However, this approach does not scale as the number of Node Agents increases since training jobs may require simultaneous curve predictions. Instead we push the learning curve prediction to the Node Agents. The Node Agents keep track of the curve history for each job they are responsible for and report to the central scheduler the results of a prediction. If a training job is suspended and resumed on different machines the learning curve history is sent to the new Node Agent when the job is resumed.

Overlap training and prediction. A simple prediction implementation would block training while the prediction is computed. Instead, as soon as the Node Agent detects that prediction should be started it does so in parallel to training. We have found, and our evaluation shows, that the end-to-end performance gains outweigh any slowdown that the training may experience due to resource contention. Although a similar approach was taken by [11], resource contention was not an issue since training was done exclusively on GPUs and prediction on CPUs. Our evaluation covers both CPU and GPU-based training using this strategy.

5.3 Scheduling Policies

We now describe how we use HyperDrive to implement three scheduling policies used in our evaluation: the POP policy from §3 and two state-of-the-art policies from prior work: a bandit allocation policy from [25], and an early termination policy from [11].

POP We implement the POP algorithm as described in §3. When *OnIterationFinish* is called the policy checks to see if the current iteration (n) is on an evaluation boundary (b), if so we perform several steps. We first compute the expected number of iterations the job has remaining (k) Then compute our p value for the job as described in §3. In order to calculate our desired and deserved slots we compute the tail distribution across all currently active (non-terminated) job's p values. Then compute our dynamic $p_{threshold}$. Then compare our threshold to all active jobs and determine if they are in our *promising resource pool* or not, we label each promising job (using *labelJob*) with a priority value of p. Lastly, if the job is opportunistic we suspend it and start a new job.

We now discuss domain and task-level knowledge we incorporate into POP to prune poor configurations. Before computing any learning prediction we first check to see if the job's performance has passed a user-defined *kill-threshold* based on the specific learning task. For example, in the CIFAR-10 task (discussed in §6.1) it is known that random non-learning validation accuracy is 10%, therefore we set the kill-threshold to a value slightly over random accuracy at 15%. In our LunarLander reinforcement-learning task (discussed in §6.1) we know that non-learning performance is -100 therefore we set our kill-threshold to -100. In addition, in order to prune off jobs that are unlikely to achieve our target, we compare a job's p value against a lower-bound threshold, if it is less than 0.05 we terminate it. Lastly, we set b to 10 for supervised-learning and to 2,000 for reinforcement-learning.

Bandit Our Bandit policy is based on the action elimination algorithm [12] used by TuPAQ [25] in their bandit allocation strategy. We extend the Default SAP described in §4. Model performance stats are sent to the policy every epoch, the SAP keeps track of the global best model performance (*globalBest*) along with the best model performance per job (*jobBest*). When *OnIterationFinish* is called the SAP checks to see if the current iteration is on an evaluation boundary (b), if so it checks if $jobBest * (1 + \epsilon) > globalBest$. If true, the job continues training, if false the policy terminates the job. Based on prior work [25], ϵ is set to 0.50 and b is set to 10 for supervised-learning. Prior work focused on supervised-learning, therefore we have no guidance on setting an evaluation boundary for reinforcement-learning. Thus, we use the same value as our POP policy (i.e., 2,000 iterations).

EarlyTerm The EarlyTerm policy is a parallel version of prior work [11] that introduced the learning curve prediction model used in our POP policy, we use the same optimizations here as described in §5.2. Like Bandit, we extend the default SAP. The EarlyTerm policy implements the "predictive termination criterion" described in [11]. Model performance stats are sent to the policy where it keeps track of the full history of performance across each job, along with $\hat{y}$ which is the global best model performance seen. When *OnIterationFinish* is called the policy checks if the current iteration (n) is on an evaluation boundary (b), if so it computes $p_{val} = P(y_m \geq \hat{y} | y_{1:n})$ using its probabilistic model. If $p_{val} < \delta$ then the job is immediately terminated. The value of m is set to the max epoch set for the training jobs. We use the same b value of 30 and δ to 0.05 as [11]. Similar to the Bandit policy, prior work provides no guidance on b values for reinforcement-learning, thus we use the same value as our POP policy (i.e., 2,000 iterations).

6 Evaluation

We evaluate the effectiveness of our proposed POP scheduling algorithm and HyperDrive framework in two different domains: supervised-learning (§6.2) and reinforcement-learning (§6.3).

6.1 Experimental Setup

Scheduling Policies. In each learning domain, we compare POP against three baseline approaches: (1) Default, (2) Bandit, and (3) EarlyTerm. The Default policy (see §4) schedules jobs greedily on idle machines and runs them until completion (i.e., a max number of epochs). Bandit and EarlyTerm are based on prior work and their implementation is described in §5.3.

Workloads. For supervised learning, we use a popular image classification task, CIFAR-10 [20], that classifies 32x32 images into 10 categories (e.g., cat, truck). We use a convolutional neural network (CNN) based on the *layers-18pct* configuration from Krizhevsky's cuda-convnet [19]. Even though this model does not have state-of-the-art accuracy, it is a popular version coming with Caffe. State-of-the-art models often employ data augmentation and/or intensive preprocessing steps, which are orthogonal to hyperparameter exploration that our work focuses on. We follow the standard approach

of training on 50k images and evaluating model performance on a validation dataset of 10k images.

For reinforcement learning, we use a model for a popular task from the OpenAI Gym [17] called "LunarLander". LunarLander comes from a game where an agent has control over a lander to do four discrete actions: do nothing, fire left engine, fire main engine, or fire right engine. The environment rewards the agent based on how efficiently it uses its resources to successfully land between two goal posts (without crashing). The problem is considered "solved" if the agent consistently achieves an average reward of 200 over 100 consecutive trials. If the lander crashes it receives a reward of -100 and the trial ends. We use a model written in Keras [9] and Theano [6] provided by the authors of [4]. Different than supervised-learning that uses validation accuracy as its performance metric, reinforcement-learning uses reward.

Hyperparameter Sets. To ensure fair comparison, we use the same set of hyperparameters for evaluation, i.e., using the same random search Hyperparameter Generator with the same initial random seed. The hyperparameter set consists of 100 configurations for both supervised and reinforcement learning experiments. Specifically, we explore up to 14 different hyperparameters for CIFAR-10 with the same hyperparameters and value ranges as in Table 3 of [11]. We explore 11 different hyperparameters for 'LunarLander" and we use ranges and values provided by the authors of the model [4].

Testbed. We conduct live GPU experiments for supervised-learning on a private 4-machine GPU cluster that we refer to as *private-cluster*. We co-locate the HyperDrive scheduler with one of the training machines in the cluster. Each machine is equipped with an Intel Xeon E5-2680 v2 2.80GHz CPU, 128 GB of memory, 10 Gbps network connectivity, and one Tesla K40m GPU. We use Ubuntu 14.04.5 LTS with Python 2.7.6, CUDA v8.0.44, and the CuDNN library v5.1.10. We use a version of Caffe 1.0.0-rc3 that we modified to report application metrics (e.g., accuracy) to a local Node Agent running on the same machine.

We conduct reinforcement-learning experiments on AWS. We use 15 c4.xlarge instances for training and a single m4.xlarge instance for running the HyperDrive scheduler. Each training machine uses Ubuntu 16.04.02 LTS, Python 2.7.12, Theano 0.9.0, and Keras 1.1.0. For suspending and resuming a configuration, we incorporate CRIU 2.6 into HyperDrive.

Non-Determinism. A challenge with evaluating scheduling algorithms for hyperparameter exploration is non-determinism that comes from the asynchronous nature of model training algorithms [22]. We observe that this non-determinism could vary model performance at a given epoch by up to 2%. To reduce the effect of this non-determinism, we run each experiment multiple times: 10 times for supervised learning and 5 times for reinforcement learning.

6.2 Supervised-Learning

6.2.1 Job Execution Duration

Figure 6 shows the distribution of job execution durations for POP, Bandit, and EarlyTerm. POP spends considerably less time across all jobs than the other policies, this is especially the case when looking at longer running jobs. Particularly we see that Bandit and EarlyTerm spend around 30 min or more on almost 15% of jobs, where POP spends 30 min or more on only 5% of jobs. We see in the following section that by spending less time overall executing less-promising jobs we are able to achieve improved performance.

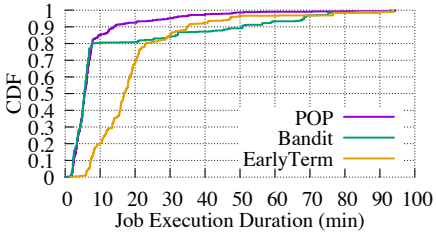


Figure 6. Job execution duration distribution comparing different scheduling policies with supervised-learning workload.

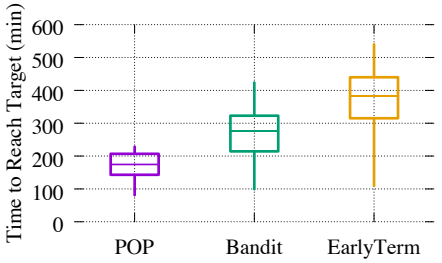


Figure 7. Time to reach target validation accuracy (CIFAR-10).

6.2.2 Scheduling Performance Comparison

We evaluate the performance of different policies by comparing the training time to reach a given target accuracy using the same cluster. We select a target accuracy of 77% based on the domain knowledge of our CIFAR-10 model [19], which is close to the best accuracy reported for this model. Our choice of this model is discussed in §6.1. For each policy, we repeat the experiment 10 times. The results are presented in Figure 7 as box plots showing the different quartiles for achieving the target accuracy under each policy. On average POP reached the target accuracy in only 2.8 hours, whereas Bandit took 4.5 hours and EarlyTerm took 6.1 hours. POP outperforms Bandit by 1.6x and outperforms EarlyTerm by 2.1x. In addition, the difference between the minimum and maximum training times using POP is much smaller (around 2x) than Bandit and EarlyTerm. Even the worst performing run of POP is faster than the best case of the Bandit and EarlyTerm. This indicates that POP is not only faster in reaching target accuracy, but also offers more stable performance, thanks to its judicious classification and scheduling. We experimented with different training accuracy targets and different variations of CIFAR-10 and the observations are consistent. In the interest of space, we omit the results here.

6.2.3 Scheduling Overhead

The advantages of HyperDrive with POP are at the cost of extra scheduling overhead compared to other approaches. The cost of suspending & resuming training jobs can incur higher overhead than other scheduling algorithms. Suspending training jobs involves capturing model state that enables later resumption of training. The captured model state of different jobs are sent to HyperDrive for storage and dissemination. Therefore, the overhead includes suspend/resume time and storage costs for model state. In this study we measure *suspend latency*, which measures the time between when the scheduler sends a suspend request to the Node Agent until the scheduler finishes stored the model state. For brevity we omit additional figures, and instead summarize our findings here.

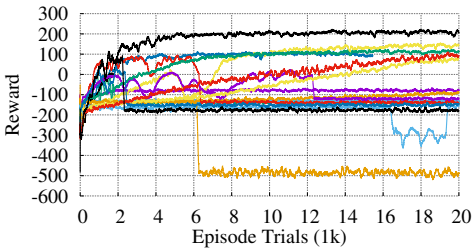


Figure 8. Performance of 15 randomly selected LunarLander model configurations over 20,000 episode trials.

On average this latency is only 157.69 ms with a standard deviation of 72 ms. We observe the 95th percentile latency to be 219 ms and a maximum of 1.12 sec. In terms of the model state size we observe an average total size of 357.67 KB with a standard deviation of 122.46 KB. We observe the 95th percentile to be 685.26 KB and a maximum of 686.06 KB. We find in practice and show in our end-to-end evaluation results that these overheads show negligible impact on scheduling and training performance.

6.3 Reinforcement-Learning

Figure 8 shows the performance of 15 randomly selected LunarLander configurations. Unlike supervised-learning, we observe that many jobs learn for some period of time and then experience what we call a "learning-crash", in which the reward falls and remains at or below a non-learning value. In the plot, the non-learning value is -100, which is related to the negative reward given by the environment when the lander crashes. We observe that over 50% of jobs are non-learning and should not be fully executed.

Reward values in the LunarLander task generally range between -500 and 300, in order for any scheduling policy to compare relative performance between configurations, we normalize all reward values using min-max scaling. We transform every reward value r as follows:

$$r_{norm} = \frac{r - r_{min}}{r_{max} - r_{min}}, \tag{4}$$

In our experiments, we use $r_{min} = -500$ and $r_{max} = 300$. The upper-bound range (r_{max}) is determined by the environment and task while the lower bound range (r_{min}) is determined empirically (we use this method) by observing a small number of poor performing runs or can by calculated the time allowed per episode and the maximum number of actions allowed.

6.3.1 Scheduling Performance Comparison

A priori target performance is common in many reinforcement-learning tasks. In LunarLander, the environment explicitly sets a "solved" condition that can be used as our target, i.e., an average reward of 200 over 100 consecutive trials.

Figure 9 presents the time to reach target results for each policy. We repeat the same experiment five times for each policy. We observe POP achieves a median time to target 2.07x faster than Bandit and 1.26x faster than EarlyTerm. Again, training time variations are much lower for POP compared to Bandit (9.7x smaller) and EarlyTerm (3.5x smaller) policies. These results show that compared to state-of-the-art approaches, HyperDrive with POP is faster in reaching target accuracy, and also more stable performance-wise for reinforcement learning.

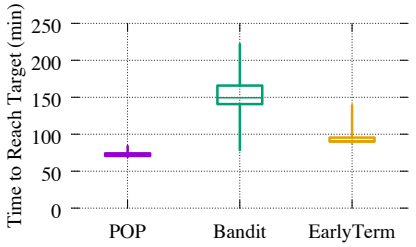


Figure 9. Time to reach target reward (LunarLander).

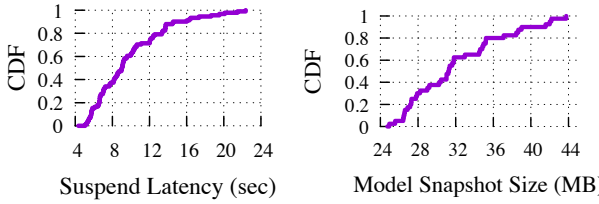


Figure 10. Suspend latency (left) and snapshot size (right) distributions for LunarLander workload.

6.3.2 Scheduling Overhead

We use CRIU to assist suspending/resuming training jobs. When a training job is suspended, all of its processes state is snapshotted and transferred back to HyperDrive. Instead of adding save/resume support to our model we use a more general approach using CRIU to snapshot the entire process state. We recognize that this method may incur higher overhead than a custom solution for our model. This study measures the overhead of suspending our LunarLander training job from the perspective of HyperDrive. Figure 10 presents the distributions of both suspend latency and model snapshot size. We see that model size does not exceed 43.75 MB and latency does not exceed a maximum of 22.36 sec, which is considerably small compared with job training time.

In summary, POP is faster in reaching target performance and more stable for different learning domains compared to state-of-the-art approaches such as Bandit and EarlyTerm.

7 Sensitivity Analysis

In this section, we perform sensitivity analysis relating to the resource capacity and configuration order for both supervised and reinforcement learning using different polices. Due resource constraints, we opt for developing a simulator to perform sensitivity analysis. To ensure accurate simulation, we feed the simulator with traces collected from live system experiments.

7.1 Simulator

Our goal is to compare the scheduling efficiency (i.e., time to reach target accuracy) between different policies under different resource capacities and configuration orders. We develop a trace-driven simulator consisting of the following three main components, see Figure 11:

- **Trace Generator** collects the traces from live system experiments and creates a replayable workload that contains iteration timing and performance metrics. In addition, the Trace Generator can create traces by changing the configuration orders. This feature is useful to conduct sensitivity analysis of configuration orders.

- **Simulator Engine** is a trace-driven discrete event simulator that accurately emulates the execution process of HyperDrive, i.e., the order of configurations and the resource management logic.
- **Pluggable Scheduling Policy** dictates the scheduling decisions on configuration ordering and the resources allocated to different configurations over time.

To validate the accuracy of our simulator, we compare the simulation results with the live system results using different policies in Figure 12a, which shows the time to reach target accuracy for LunarLander using 15 machines. We see the simulation results are quite accurate, i.e., compared to the live system results, the max error of simulation is only 13%, which is well below the error bar of live system results.

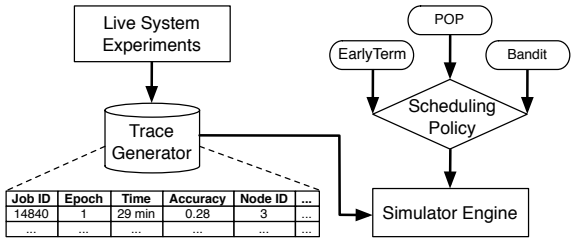


Figure 11. Simulator Design.

7.2 Supervised-Learning

All simulator traces for supervised-learning are collected from live AWS runs using 4 g2.2xlarge instances (one K520 GPU each)[8] for training and one m4.xlarge instance for the HyperDrive scheduler.

7.2.1 Sensitivity Analysis of Resource Capacity

We study the sensitivity of resource capacity (i.e., total number of machines) by comparing the time it takes to reach our target validation accuracy (77%) for different policies using CIFAR-10. Figure 12b presents the results of simulation experiments. As we would expect, the time to achieve our target improves with more machines across all of our scheduling policies. POP always outperforms other policies under different resource capacities. In addition, with larger resource capacities, POP shows even more performance improvement to the second best policy. These results verify the effectiveness of our POP policy when using different amounts of resources.

7.2.2 Sensitivity Analysis of Configuration Order

The configuration order that a scheduling policy sees can heavily impact experiment performance. For example, since configurations are generated randomly it's possible an exhaustive search technique could "luckily" pick the optimal configuration as the first configuration (or in the first batch of configurations) to explore, its performance can be as good as or even better than any sophisticated policies. In order to understand the sensitivity of configuration order for different policies, we run simulation experiments with 25 random configuration orders on 5 machines. The results presented in Figure 12c demonstrate the distribution of time to reach target validation accuracy under different polices. It is clear that POP yields much better performance in all percentiles. In addition, it is also more consistent in performance than other policies, e.g., POP has a maximum difference in completion time of 4.05 hours compared to Bandit with 8.33 hours, EarlyTerm with 8.50 hours, and Default with a staggering difference of 25.74 hours. These results suggest POP is less sensitive to configuration order and therefore is more reliable.

[8]The K520 GPU is slower than the K40m GPU used in our private-cluster.

7.3 Reinforcement-Learning

We collected traces for reinforcement-learning experiments from running live experiments on AWS using 15 c4.xlarge instances for training and a single m4.xlarge instance for the HyperDrive scheduler. We conduct the same sensitivity analysis of resource capacity and configuration order for reinforcement-learning as for supervised-learning. We observe similar results as in supervised-learning. In the interest of space, we omit the results and detailed discussion here.

8 Related Work

We discuss related hyperparameter optimization work for ML/DL models.

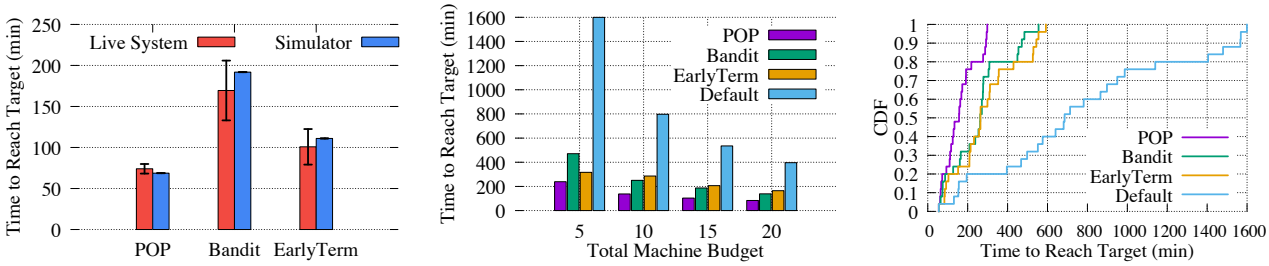
Learning Curve Prediction. Several pieces of related work present learning curve prediction models [11, 16, 26]. This prior work could be used as a drop-in replacement for the learning curve prediction model in our POP algorithm, we currently employ [11]. This line of work is complementary to POP, where we focus on how to effectively use them for classification and resource management.

Hyperparameter Generation. Several pieces of prior work [5, 14, 18, 24, 28] have moved beyond grid/random-based hyperparameter generation towards exploiting Bayesian optimization. Based on the accuracy of the configurations that have executed, they assign higher probability to exploit the areas of the hyperparameter space that contain likely-good or unknown configurations. This line of work is complementary to POP, where we can plug in different hyperparameter generation techniques. Our focus is to decide how much resources to map to the current set of configurations.

Sequential Search Algorithms. HyperBand [21] uses a multi-armed bandit approach for hyperparameter exploration. They use random search hyperparameter generation along with an aggressive bandit-based pruning technique. Swersky et al. [26] propose to suspend and resume uncompleted training jobs during model search, incorporating partial training information with learning curve prediction. Their prediction model does not seem to work well for deep neural networks as discussed in [11]. In comparison, both prior work focus on using a sequential execution of models while POP is designed for speeding up model exploration using multiple machines and exploiting resource usage along the spatial dimension.

Parallel Search System. Recently, distributed computing frameworks such as YARN [27] and Spark [30] have recognized the importance of parallel hyperparameter exploration [13]. The proposed approaches enable parallel job execution for grid and random search. They do not support active monitoring of job metrics, early termination and dynamic resource allocation among configurations.

TuPAQ [25] is closely related, we present an overview of its bandit algorithm in §5.3. POP employs job execution history for improved prediction and uses the predicted accuracy and confidence to enhance scheduling decision, while TuPAQ looks at the instantaneous accuracy of jobs when deciding who to terminate. In terms of system framework, TuPAQ focuses on a single learning domain (supervised) and computation framework (MLBase), while



(a) Simulator validation (LunarLander). **(b)** Resource capacities (CIFAR-10). **(c)** Random job orderings (CIFAR-10).

Figure 12. Sensitivity analysis via simulation.

HyperDrive is designed as a comprehensive framework supporting different algorithms across frameworks and domains. For performance, our evaluation results (in §6 and §7) show POP consistently out-performs our Bandit policy, which is based on TuPAQ.

9 Discussion

Epoch durations. Our POP policy assumes epoch durations remain relatively constant during training (see §3.1.1). Epoch durations may differ between unique sets of hyperparameter values but for a specific configuration this duration remains relatively constant. This behavior is common in learning domains evaluated in this work (supervised and reinforcement), however we leave evaluating other domains that may experience non-constant epoch durations (e.g., genetic algorithms) to future work.

Learning curve prediction. Our POP policy relies on a learning curve prediction model. In this paper, we choose the model proposed in prior work [11] (which has been studied with extensive evaluation) for this component, we have found it to work well for the workloads we are using. In addition, we design the learning curve prediction module as a pluggable component of HyperDrive, so users can easily switch to other prediction methods as preferred. Learning curve prediction is an active area of research [11, 16, 26] and we foresee no issue using different approaches as research advances in this area.

User inputs. Our POP policy aims to achieve a training goal within specific time/resource constraints, therefore it requires as input a maximum experiment time (T_{max}) and target performance (y_{target}). Specifying T_{max} requires some knowledge related to typical configuration training time, since a too small T_{max} may result in insufficient time to finish model training. Setting y_{target} is natural for domains with known goals, such as our LunarLander task (see §6.3). However, if a y_{target} is unknown we have successfully used a dynamic target approach to automatically adjust y_{target} by gradually increasing the target once it is reached. In the interest of space, we leave the details and evaluation of this approach to future work. In our work with practitioners we have found that before starting hyperparameter exploration for their model they have a good idea about both of POP's required inputs.

POP, Bandit, and EarlyTerm policies all require a user-defined evaluation boundary (b) to be specified. Setting this value is a common problem in the space of early-termination policies. Like in prior work, b is model/domain specific and should reflect the time it takes to compute model performance and how long a user is willing to let a configuration execute before possible termination. We have found success with a heuristic of setting b to be between 5-10% of the max epoch for a job. We leave automatically setting this parameter to future work.

Ongoing Work. HyperDrive enables model owners to schedule resources based on monitored application-level metrics. Typically there is a primary metric being optimized (e.g., accuracy) which is what POP utilizes. However, additional metrics of concern can be impacted by hyperparameter choices as well, such as inference/serving latencies, model sparsity/compressibility, etc.

We have seen promising early results exploring hyperparameters specific to models that use Long Short-Term Memory (LSTM) units. We are working together with authors of recent work on improving CNN model sparsity [29]. The work aims to reduce the size of LSTMs structurally, for both storage saving and computation time saving, without perplexity loss (the primary metric for our task). This is done through the use of group Lasso regularization [32], which adds enforcement on each group of a model's weights. The method uses a new hyperparameter (i.e., λ) which makes a trade-off between sparsity and model perplexity. We have evaluated several state-of-the-art models from recent work [23, 33] with a new HyperDrive policy, exploring λ values (plus other hyperparameters) while monitoring both perplexity and a sparsity-related metric. We have seen significantly reduced training times by enabling user-defined global termination criteria through HyperDrive's SAP API.

Lastly, we are working with Microsoft engineers to productize HyperDrive internally, which will continue improving HyperDrive's usability, scalability, and effectiveness. Hyperparameter exploration will continue to be an active research area but currently there are limited options to develop/evaluate parallel approaches that incorporate techniques along *both* hyperparameter generation and scheduling (e.g., early termination, suspend/resume).

10 Conclusion

This paper presents an approach for improving the efficiency of developing machine learning models by optimizing hyperparameter exploration. Our approach includes two techniques: (i) the POP scheduling algorithm and (ii) the HyperDrive framework. POP employs dynamic classification of model configurations and prioritized resource scheduling to discover high-quality models faster than state-of-the-art approaches. HyperDrive is a flexible framework that enables convenient evaluation of different hyperparameter exploration algorithms to improve the productivity of practitioners. We present experimental results that demonstrate the performance benefits of using POP and HyperDrive to develop high-quality models in supervised and reinforcement learning domains.

Acknowledgements: We thank our shepherd, Marco Canini, and anonymous reviewers for their valuable comments; those at Microsoft helping to productize HyperDrive, Radu Kopetz, Prasanth Pulavarthi, Sherlock Huang, Yue-Sheng Liu; Kavosh Asadi for providing the RL model; early HyperDrive users Minjia Zhang and Wei Wen; Trishul Chilimbi for supporting the initial effort and brainstorming.

References

[1] 2017. A high performance, open-source universal RPC framework. https://grpc.io. (2017).

[2] 2017. Checkpoint/Restore In Userspace (CRIU). https://criu.org/. (2017). Accessed: 2017-09-13.

[3] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, GA, 265–283. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi

[4] Kavosh Asadi and Jason D. Williams. 2016. Sample-efficient Deep Reinforcement Learning for Dialog Control. *CoRR* abs/1612.06000 (2016). http://arxiv.org/abs/1612.06000

[5] The GPyOpt authors. 2016. GPyOpt: A Bayesian Optimization framework in python. http://github.com/SheffieldML/GPyOpt. (2016).

[6] James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. 2010. Theano: A CPU and GPU Math Compiler in Python . In *Proceedings of the 9th Python in Science Conference*, Stéfan van der Walt and Jarrod Millman (Eds.). 3 – 10.

[7] J. Bergstra, D. Yamins, and D. D. Cox. 2013. Making a Science of Model Search: Hyperparameter Optimization in Hundreds of Dimensions for Vision Architectures. In *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28 (ICML'13)*. JMLR.org, I–115–I–123. http://dl.acm.org/citation.cfm?id=3042817.3042832

[8] Trishul Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. 2014. Project Adam: Building an Efficient and Scalable Deep Learning Training System. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Broomfield, CO, 571–582. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/chilimbi

[9] François Chollet. 2015. Keras. https://github.com/fchollet/keras. (2015).

[10] Ronan Collobert, Samy Bengio, and Johnny Mariéthoz. 2002. *Torch: A Modular Machine Learning Software Library*. Idiap-RR Idiap-RR-46-2002. IDIAP.

[11] Tobias Domhan, Jost Tobias Springenberg, and Frank Hutter. 2015. Speeding Up Automatic Hyperparameter Optimization of Deep Neural Networks by Extrapolation of Learning Curves. In *Proceedings of the 24th International Conference on Artificial Intelligence (IJCAI'15)*. AAAI Press, 3460–3468. http://dl.acm.org/citation.cfm?id=2832581.2832731

[12] Eyal Even-Dar, Shie Mannor, and Yishay Mansour. 2006. Action Elimination and Stopping Conditions for the Multi-Armed Bandit and Reinforcement Learning Problems. *Journal of machine learning research* 7, Jun (2006), 1079–1105.

[13] Tim Hunter. 2016. Deep Learning with Apache Spark and TensorFlow. https://databricks.com/blog/2016/01/25/deep-learning-with-apache-spark-and-tensorflow.html. (January 2016).

[14] F. Hutter, H. H. Hoos, and K. Leyton-Brown. 2011. Sequential Model-Based Optimization for General Algorithm Configuration. In *Proc. of LION-5*. 507âĂŞ523.

[15] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional Architecture for Fast Feature Embedding. *arXiv preprint arXiv:1408.5093* (2014).

[16] Aaron Klein, Stefan Falkner, Jost Tobias Springenberg, and Frank Hutter. 2017. Learning curve prediction with Bayesian neural networks. *Proc. of ICLR* 17 (2017).

[17] Oleg Klimov. 2017. LunarLander-v2. https://gym.openai.com/envs/LunarLander-v2. (2017).

[18] Brent Komer, James Bergstra, and Chris Eliasmith. 2014. Hyperopt-sklearn: automatic hyperparameter configuration for scikit-learn. In *ICML workshop on AutoML*.

[19] Alex Krizhevsky. 2017. cuda-convnet. https://code.google.com/p/cuda-convnet/. (2017).

[20] Alex Krizhevsky and Geoffrey Hinton. 2009. Learning multiple layers of features from tiny images. (2009). Technical report, University of Toronto.

[21] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. 2017. Hyperband: Bandit-based Configuration Evaluation for Hyperparameter Optimization. *Proc. of ICLR* 17 (2017).

[22] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. 2011. Hogwild: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent. In *Advances in Neural Information Processing Systems 24*, J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. Pereira, and K. Q. Weinberger (Eds.). Curran Associates, Inc., 693–701.

[23] Min Joon Seo, Aniruddha Kembhavi, Ali Farhadi, and Hannaneh Hajishirzi. 2016. Bidirectional Attention Flow for Machine Comprehension. *arXiv CoRR* abs/1611.01603 (2016). http://arxiv.org/abs/1611.01603

[24] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. 2012. Practical bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems*. 2951–2959.

[25] Evan R. Sparks, Ameet Talwalkar, Daniel Haas, Michael J. Franklin, Michael I. Jordan, and Tim Kraska. Automating Model Search for Large Scale Machine Learning. In *Proceedings of the Sixth ACM Symposium on Cloud Computing (SoCC 2015)*. 13. DOI: https://doi.org/10.1145/2806777.2806945

[26] Kevin Swersky, Jasper Snoek, and Ryan Prescott Adams. 2014. Freeze-Thaw Bayesian Optimization. *arXiv preprint arXiv:1406.3896* (2014).

[27] Wangda Tan and Vinod Kumar Vavilapalli. 2017. Distributed TensorFlow Assembly on Apache Hadoop YARN. https://hortonworks.com/blog/distributed-tensorflow-assembly-hadoop-yarn/. (March 2017).

[28] Chris Thornton, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. 2013. Auto-WEKA: Combined Selection and Hyperparameter Optimization of Classification Algorithms. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '13)*. ACM, New York, NY, USA, 847–855. DOI: https://doi.org/10.1145/2487575.2487629

[29] Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. 2016. Learning Structured Sparsity in Deep Neural Networks. In *Advances in Neural Information Processing Systems 29*, D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett (Eds.). Curran Associates, Inc., 2074–2082. http://papers.nips.cc/paper/6504-learning-structured-sparsity-in-deep-neural-networks.pdf

[30] Lee Yang, Jun Shi, Bobbie Chern, and Andy Feng. 2017. Open Sourcing TensorFlowOnSpark: Distributed Deep Learning on Big-Data Clusters. http://yahoohadoop.tumblr.com/post/157196317141/open-sourcing-tensorflowonspark-distributed-deep. (February 2017).

[31] Dong Yu, Adam Eversole, Mike Seltzer, Kaisheng Yao, Oleksii Kuchaiev, Yu Zhang, Frank Seide, Zhiheng Huang, Brian Guenter, Huaming Wang, Jasha Droppo, Geoffrey Zweig, Chris Rossbach, Jie Gao, Andreas Stolcke, Jon Currey, Malcolm Slaney, Guoguo Chen, Amit Agarwal, Chris Basoglu, Marko Padmilac, Alexey Kamenev, Vladimir Ivanov, Scott Cypher, Hari Parthasarathi, Bhaskar Mitra, Baolin Peng, and Xuedong Huang. 2014. *An Introduction to Computational Networks and the Computational Network Toolkit*. Technical Report.

[32] Ming Yuan and Yi Lin. 2006. Model selection and estimation in regression with grouped variables. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)* 68, 1 (2006), 49–67.

[33] Wojciech Zaremba, Ilya Sutskever, and Oriol Vinyals. 2014. Recurrent Neural Network Regularization. *arXiv CoRR* abs/1409.2329 (2014). http://arxiv.org/abs/1409.2329