MLtuner: System Support for Automatic Machine Learning Tuning

Henggang Cui, Gregory R. Ganger, and Phillip B. Gibbons
Carnegie Mellon University

CMU-PDL-16-108
October 2016

Parallel Data Laboratory
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Abstract

MLtuner automatically tunes settings for training tunables—such as the learning rate, the mini-batch size, and the data staleness bound—that have a significant impact on large-scale machine learning (ML) performance. Traditionally, these tunables are set manually, which is unsurprisingly error prone and difficult to do without extensive domain knowledge. MLtuner uses efficient snapshotting and optimization-guided online trial-and-error to find good initial settings as well as to re-tune settings during execution. Experiments with three real ML tasks show that MLtuner automatically enables performance within 40–178% of having oracle knowledge of the best settings, and outperforms oracle when no single set of settings are best for the entire execution. It also significantly outperforms most of the many feasible settings that might get used in practice.

Acknowledgements: We thank the members and companies of the PDL Consortium (including Broadcom, Citadel, Dell EMC, Facebook, Google, Hewlett-Packard Labs, Hitachi, Intel, Microsoft Research, MongoDB, NetApp, Oracle, Samsung, Seagate, Tintri, Two Sigma, Uber, Veritas, and Western Digital) for their interest, insights, feedback, and support. This research is supported in part by Intel as part of the Intel Science and Technology Center for Cloud Computing (ISTC-CC), National Science Foundation under awards CNS-1042537 and CNS-1042543 (PRObE [15]).

Keywords: Big Data infrastructure, Big Learning systems

1 Introduction

Large-scale machine learning (ML) is quickly becoming a common activity for many organizations. ML training computations generally use iterative algorithms to converge on thousands to millions of parameter values that make a pre-chosen model (e.g., a neural network or pair of factor matrices) statistically approximate the reality corresponding to the input training data over which they iterate. Trained models can be used to predict, cluster, or otherwise help explain subsequent data.

For large, complex models, parallel execution over multiple cluster nodes is warranted. The algorithms and frameworks used generally have multiple tunables that have significant impact on the execution and convergence rates. For example, the learning rate is a key tunable when using stochastic gradient descent (SGD) for training. As another example, the data staleness bound is a key tunable when using frameworks that explicitly balance asynchrony benefits with inconsistency costs [7, 25].

These tunables are usually manually configured or left to broad defaults. Unfortunately, knowing the right settings is often quite challenging. The best tunable settings can depend on the chosen model, the hyperparameters (e.g., number and types of layers in a neural network), the algorithm, the framework, and the resources on which the ML application executes. As a result, manual approaches not surprisingly involve considerable effort by domain experts or yield (often highly) suboptimal training times and model accuracy. Our interactions with both experienced and novice ML users comport with this characterization.

MLtuner is a tool for automatically tuning ML application training tunables. It hooks into and guides a training system in trying different settings, using our API. MLtuner determines initial tunable settings based on rapid trial-and-error search, wherein each option tested runs for a small (automatically determined) amount of time, to find good choices based on the convergence rate. MLtuner will repeat this process when convergence slows, to see if different settings provide faster convergence and/or higher model accuracy. We address a number of challenges in auto-tuning ML applications such as large search spaces, noisy convergence progress, variations in effective trial times, best tunable choices changing over time, when to re-tune, etc. Through careful design, MLtuner ensures that the time spent running the application with optimized settings dominates the time spent searching for such settings.

We have integrated MLtuner with two different state-of-the-art training systems and experimented with several real ML applications, including a recommendation application on a CPU-based parameter server system and both image classification and video classification on a GPU-based parameter server system. The results show MLtuner's effectiveness. MLtuner finds good tunable settings consistently, without excessive overhead from searching. For example, MLtuner incurs only 40% extra runtime on the ILSVRC12 image classification application, and reaches a higher model accuracy than not re-tuning the tunables multiple times during execution or tuning with state-of-the-art SGD learning rate tuning algorithms, such as AdaGrad [12], AdaRevision [27], AdaDelta [47], Nesterov [29], Adam [20], and RMSProp [42].

This paper makes the following primary contributions. First, it introduces the first approach for automatically tuning the multiple tunables associated with nearly any ML application within the context of a single execution of that application. Second, it describes a tool (MLtuner) that implements the approach, overcoming various challenges, and how MLtuner was integrated with two different ML training systems. Third, it presents results from experiments with real ML applications, demonstrating the efficacy of this new approach in removing the "black art" of tuning from ML application training.

2 Background and Related Work

2.1 Distributed machine learning

The goal of a ML task is to train a ML model on training (input) data, so that the trained model can be used to make predictions on unseen data. A model has trainable *model parameters*, and the ML task tries to determine the values of these model parameters that best fit the training data. The fitness error of the set of model parameters to the training data is defined mathematically with an *objective function*, and the current value of the objective function is often called the *training loss*. Thus, the machine learning problem is an optimization problem, whose goal is to minimize the objective function.

The ML task often optimizes the objective function with an iterative convergent algorithm, such as stochastic gradient descent (SGD). The model parameters are first initialized to some random values, and in every step, the SGD algorithm samples a batch of the training data (a *training batch*), and computes the gradient of the training loss, w.r.t. the model parameters. The model parameter update will be the opposite direction of the gradient (if we want to minimize the objective function), multiplied by a *learning rate* parameter.

To speed up ML tasks, people often distribute the ML computations across a cluster of machines. One popular way of doing that is the *data parallel* approach, where we partition the training data across many ML workers on different machines. Each ML worker keeps a local copy of the model parameters and computes model gradients based on their local model parameter copy and training data partition. The ML workers will propagate the computed model gradients and refresh their model parameter copies every *clock*, which is often logically defined as some unit of work (e.g., one pass over the training data). Those shared model parameters are often managed with a *parameter server* architecture [25, 9, 45, 8, 6, 10, 2, 32], in which a simple key-value interface is used to read and update the model parameters that are the only shared state among ML workers.

When training the model in this distributed manner, ML workers will have temporarily inconsistent model parameter copies, causing *data staleness errors* [17, 7]. To guarantee model convergence, it is important to enforce some *consistency model* that bounds the data staleness in some way. The Bulk Synchronous Parallel (BSP) model, for example, requires all workers must see the parameter updates from all other workers from the previous clocks before proceeding to the next clock. The Stale Synchronous Parallel (SSP) model [17, 7] is a looser but still bounded model, with a *slack* parameter. It allows the fastest worker to be ahead of the slowest worker by up to "slack" clocks, generalizing BSP as a special case of SSP with a slack of zero.

2.2 Machine learning tunables

Training a ML model requires selection and tuning of many *training hyperparameters*. For example, the SGD algorithm has a *learning rate* (a.k.a. step size) parameter that controls the magnitude of the model parameter updates. The *training batch size* parameter controls the amount of training data that each ML worker processes in every clock. The *data staleness bound* parameter controls the degree of data staleness that is allowed for the ML workers to have without forcing a synchronization. To emphasize that those training hyperparameters can be tuned, we call them *training tunables* in this paper.

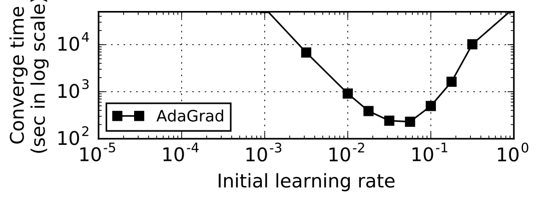
It is important to distinguish the training tunables from another class of ML hyperparameters, called *model hyperparameters*. The training tunables do not show up in the objective functions of the ML task, and they only control how we train the model. The model hyperparameters, on the other hand, show up in the objective functions and define the models. Example model hyperparameters include the structure of the model (e.g., logistic regression or SVM), the depth of a neural network, the width and the scale of each neural network layers, and the regularization method and magnitude. In this work, we focus on the training tunables, and we

assume the model hyperparameters are fixed.
Many people (as well as our own experiments) have found that the training tunables have a big impact on the completion time of a ML task (e.g., orders of magnitude slower with a bad tunable choice) [34, 30, 12, 27, 47, 20, 17, 7]. Moreover, a bad tunable choice can also cause the model to diverge or converge to suboptimal solutions [34, 30].
Tuning the tunables is hard. First, the optimal tunable choice depends on many factors, such as the ML application, the model, the dataset, the size of the cluster, and the network condition. Second, the optimal tunable choice can change during the training. For example, when training a deep neural network, people find it is beneficial (sometimes even necessary) to start with a large learning rate at the beginning, to approach the solution quickly, and then decrease the learning rate, to get better model convergence [40, 19, 39, 23, 16, 1].

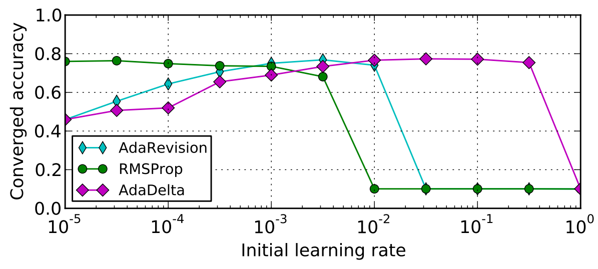
2.3 Related work on tuning tunables

This section describes the related work on tuning tunables.
Manual tuning by domain experts. Most previous large-scale ML literature does not talk about the process of selecting the training tunables at all. Users usually just *manually* set the tunables, either by using some built-in defaults or trying various tunable settings offline and picking a settings via trial-and-error. This process is inefficient, and the tunable settings chosen are often suboptimal.
For some problems, such as deep neural network training, people find dynamically changing the learning rate during execution is necessary to achieve good model accuracy. There are typically two ways of doing that. The first way (taken by [23, 1, 48]) is to manually change the learning rate every a few iterations or when the model accuracy plateaus (i.e., stops increasing). The second way (taken by [40, 19, 39]) is to let the learning rate automatically decrease with some function, such as $\eta = \eta_0 \times \gamma^t$. The speed of the learning rate decrease is controlled by the γ parameter, and the best γ choice varies for different models and datasets. This γ parameter is essentially a training tunable that is much harder to tune than the learning rate, because we are not able to see its effect immediately. To decide the γ parameter, people actually need to manually train the model multiple times, with different γ choices, and identify the best ones as their experimental setups.
Hyperparameter optimization. There are some prior works on optimizing ML hyperparameters (sometimes also called model search), including [35, 37, 43, 3, 21, 4, 26, 31, 13, 41]. However, as their name suggests, these works target a quite different goal from ours. They focus on tuning the model hyperparameters, rather than the training tunables.
Because each set of model hyperparameter settings corresponds to a different ML model, evaluating each set of hyperparameter settings generally requires training the model *to completion*. Because of the high overhead of evaluating a hyperparameter choice, automated hyperparameter tuning schemes usually cannot afford trying all possible hyperparameter choices with a grid search. Instead, they use more sophisticated algorithms, such as Bayesian optimization [28], to decide based on a much smaller number of hyperparameter setting trials [35].
These hyperparameter optimization works either do not mention the training tunables at all, or they optimize them in the same way as optimizing the model hyperparameters, which is training the model to completion with every hyperparameter/tunable combination [35, 37]. This is very inefficient. Moreover, with this approach, people can only use a single tunable choice for the whole training, and they do not dynamically change the tunables during the training.
SGD learning rate tuning algorithms. Because the SGD algorithm is well-known for being sensitive to the learning rate, people have designed many learning rate tuning algorithms for SGD, such as AdaGrad [12], AdaRevision [27], AdaDelta [47], RMSProp [42], Nesterov [29], and Adam [20]. These algorithms will use relatively large learning rates for model parameters with small gradients, and relatively small learning

rates for model parameters with large gradients. However, they still require the user to select the initial learning rate. Even though they are less sensitive to the initial learning rate choice than the original SGD algorithm, our experiments show that choosing a good initial learning rate is still crucial. For example, a bad initial learning rate choice can cause the training time to be orders of magnitude longer (e.g., Figure 1(a)) and/or cause the model converge to suboptimal solutions (e.g., Figure 1(b)). MLtuner's auto-tuning approach actually complements these SGD learning rate tuning algorithms, because the user can use MLtuner to pick the initial learning rate for those algorithms.



(a) Matrix factorization convergence times.



(b) Deep neural network (AlexNet on Cifar10) converged model accuracies.

Figure 1: Different initial learning rates have big impact on convergence times and converged model accuracies. Experimental setup details are described in Section 5.

Moreover, our experiments (as well as many other people) find that using these SGD learning rate tuning algorithms alone is sometimes not enough to train the model to *full convergence*, especially for complicated models such as the deep neural networks. That is why careful manual learning rate tuning (during the training) is still a common approach taken by ML practitioners [40, 19, 39, 23, 16, 1, 48]. MLtuner automatically re-tunes the tunables during execution.

3 System support for tunable tuning

Training tunables should be tuned automatically, with a much smaller overhead, and dynamically re-tuned during the training. This section describes our *auto-tuning* approach and the high level design of our realization, *MLtuner*.

3.1 MLtuner overview

MLtuner is a system for automatic tuning. It can be connected to existing training systems, such as a parameter server, and automatically pick and tune the training tunables for them. The detailed interface is described in Section 4.4.

MLtuner lets the user specify the tunables to be tuned via a config file. Similar to the hyperparameter optimization approach [35], each tunable will be specified with the type, either continuous or discrete, and the valid value range. The user will provide MLtuner with the minimum and maximum valid values (for a continuous tunable), or the list of valid discrete values (for a discrete tunable).

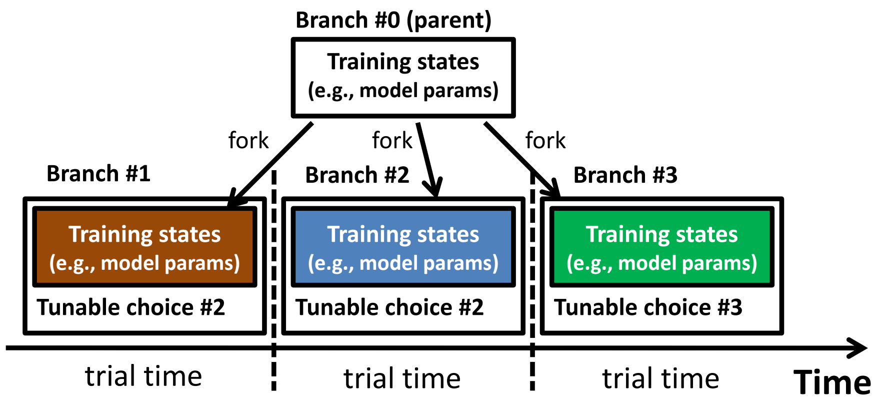


Figure 2: Trying tunable choices in training branches.

3.2 Trying & evaluating tunable choices

MLtuner tries and evaluates the tunable choices with forked *training branches*, each running with a different *tunable choice*. This is illustrated in Figure 2. Except for the tunables, the forked training branches have exactly the same initial states (e.g., model parameters, worker-local states, and training data), and run on the same hardware environment. MLtuner will run these training branches, each for a short period of *trial time*, and collect their *training progress* to measure the *convergence speed*. MLtuner will pick the branch with the fastest convergence speed, kill the other branches, and keep training only the best branch.

In our example applications, such as deep neural networks and matrix factorization, the training progress is simply the current value of the objective function (i.e., training loss). Because SGD-based training algorithms use the training loss to compute the gradients, the training loss can be obtained at no extra cost.

As is illustrated in Figure 2, the training branches are scheduled by MLtuner in a *time-sharing* manner, running in the same cluster of machines. We decided not to run multiple training branches in parallel, for two reasons. First, we want each training branch to use all the machines in the cluster; otherwise, the extra machines will be wasted when the trials are not running (which is most of the time). Second, we want the training branches to run independently, with no interference with each other, so that we can precisely measure their convergence speeds. Moreover, we find it better to keep all the training branches in the same training system instance, so that the branches can be forked with little overhead (simply memory copy within the same process), and some resources, such as the memory for cache and immutable training data, can be shared.

3.3 Tunable searching

This section describes the procedure of tunable searching with the trial branches, illustrated in Figure 3. MLtuner first sets the current model state as the parent branch, and all trial branches will be forked from it. Then, inside the searching loop, a *tunable searcher* module (described in Section 4.3) proposes the tunable choices to try. For each proposed tunable choice, MLtuner will instruct the training system to fork a new branch from the parent branch, with the tunable choice, and schedule it to run for some amount of *trial time*. Section 4.2 describes how MLtuner decides the trial time. Then MLtuner will collect the training progress of the trial branch from the training system, and use a *progress summarizer* module (described in Section 4.1) to summarize its *convergence speed*. The convergence speed will be reported to the tunable searcher to guide its future tunable choice proposals. The searching finishes, when the tunable searcher thinks the best tunable choice (or a good enough one) has been found.

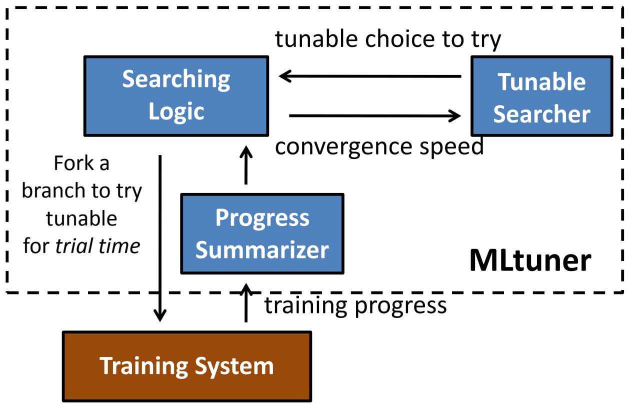


Figure 3: Tunable searching procedure.

Adjusting tunables. MLtuner not only searches the tunable choices at the beginning of the training, but it will also *adjust* the tunable choices during the training, because the best tunable choice might change. MLtuner adjusts tunables with the same searching procedure as is described above, but instead of searching from scratch, MLtuner uses a special type of tunable searcher that is able to take advantage of the fact that we know the previous best tunable choice before adjustment, which is described in more details in Section 4.3. MLtuner will try to adjust the tunables, after the current training branch has been running for more than $10\times$ of the last searching time, which bounds the searching overhead to be 10%. MLtuner will also try to adjust the tunables, when the model stops making forward convergence progress with the current tunable choice. This happens quite often for the deep learning application, where large learning rates cause the model accuracy to plateau.

4 MLtuner implementation details

This section describes the design and implementation details of MLtuner.

4.1 Measuring convergence speed

The progress summarizer module takes in the training progress trace (which is timeseries data) of each tunable trial branch, and summarizes its convergence speed. The progress trace has the form of $\{< t_i, x_i >\}_{i=1}^{N}$, where t_i is the timestamp, and x_i is the progress. In our example applications, we often use the current value of the objective function (i.e., training loss) as the progress, and a smaller x value means better progress. The convergence speed can be naturally expressed as the slope of the progress trace. However, for real world applications, there are several tricky aspects that need to be taken into account, and we will describe them in this section.

Downsampling the progress trace. The progress slope can be calculated as $s = \frac{x_N - x_1}{t_N - t_1}$. However, this simple calculation only works when the progress trace is smooth. In many real world applications, such as deep neural networks, the progress trace is often quite noisy, such as the one shown in Figure 4.

To deal with the noisy progress trace, our progress summarizer will *downsample* the progress trace of each branch. The downsampling uniformly divides the original timeseries data into K windows, and the value of each window will be downsampled as the average of all data points in that window. Suppose the downsampled progress trace is $\{< \tilde{t}_i, \tilde{x}_i >\}_{i=1}^{K}$, MLtuner will use slope of the downampled progress trace

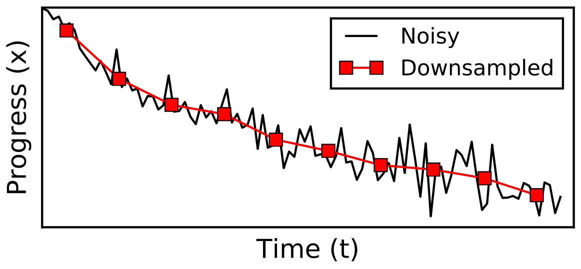


Figure 4: Downsampling noisy progress into 10 samples.

as the convergence speed, which is $s = \frac{\tilde{x}_N - \tilde{x}_1}{\tilde{t}_N - \tilde{t}_1}$. By default, MLtuner downsamples the training progress of each branch into 10 samples (i.e., $K = 10$ uniformly divided windows). Figure 4 shows a real example of downsampling the progress trace for deep neural network training. Downsampling smoothens the training progress.

Convergence and stability checks. Because the progress trace is noisy, even after the downsampling, calculating the slope by simply looking at the first and last downsampled points is likely to treat branches that have unstable jumpy losses as a good converging branch. To deal with this problem, the summarizer module will also check the stability of the training progress.

The progress summarizer labels the branch with a downampled progress trace of $\{< \tilde{t}_i, \tilde{x}_i >\}_{i=1}^{K}$ as *converging*, when both condition holds:

(1) $\tilde{x}_K < \tilde{x}_1$;

(2) $\max_{1 \le i \le K-1} (\tilde{x}_{i+1} - \tilde{x}_i) < |\tilde{x}_K - \tilde{x}_1| \times \epsilon$.

The first condition means the progress slope must be negative, assuming the progress trace of a converging branch should be going down. The second condition means that each data point in the downsampled progress trace cannot jump up from its previous point by too much (more than ϵ times the whole range). We usually use 10% as the default value for the threshold ϵ.

If the progress trace violates either of the two conditions, the progress summarizer will label this branch as *not converging*. Note that a "not converging" branches is not necessarily diverging. It might be because its progress is very noisy, and we need to run it for longer to get stable progress.

4.2 Deciding tunable trial time

MLtuner is able to automatically decide the tunable trial time, based on the convergence information from the progress summarizer module. Ideally, the trial time should be the minimal amount of time that allows a good tunable choice to have stable converging progress.

MLtuner automatically decides the trial time, with a procedure shown in Algorithm 1. MLtuner first initializes the trial time with some small value. Currently, we initialize the trial time to be at least as long as the tunable searcher decision time, so that the decision time can efficiently overlap with the running of the trial branches. MLtuner tries the tunable choices with the trial time, and if none of the tunable choices tried so far is converging, MLtuner will double the trial time, and the next time it tries another tunable choice, it will try the new tunable choice as well as all the previous tunable choices for longer, with the doubled trial time. When MLtuner successfully finds a converging tunable choice, the trial time is decided and will be used to try future tunable choices.

Algorithm 1 Deciding trial time

```
$trialTime \leftarrow 0$
Parent branch $\leftarrow$ current model states
while none of the tunables is converging do
    Get $tunableChoice$ from $TunableSearcher$
    $trialTime \leftarrow \max(trialTime, searcherDecisionTime)$
    if $tunableChoice$ is not empty then
        Fork a branch from the parent branch with $tunableChoice$
        Append the new branch to $trialBranches$
    end if
    for each branch in $trialBranches$ do
        Schedule branch to run for $trialTime - timeAlreadyRun$
    end for
    Summarize the progress of all $trialBranches$
    if any tunable choice is converging then
        $bestChoice \leftarrow$ tunable choice that has the best convergence
        Free the branches of the non-best tunable choices
        Search tunables with $trialTime$
    else
        $trialTime \leftarrow trialTime \times 2$
    end if
end while
```

4.3 Tunable searchers

The tunable searcher is a replaceable module that proposes the trial tunable choices based on the reported convergence speeds of previous trials. We have implemented and explored several types of tunable searchers on MLtuner.

GridSearcher. The *GridSearcher* simply proposes every possible tunable choices in the tunable range, without considering the reported convergence speeds. This searcher only works for discrete tunables, so continuous tunables (such as the learning rate) need to be discretized.

BayesianOptSearcher. The *BayesianOptSearcher* searches the tunable space with the Bayesian optimization algorithm [28], a tool for optimizing black-box functions. Unlike the previous model hyperparameter optimization approach [35], where the model is trained to completion for every choice and the model accuracy is used as the feedback, MLtuner just trains with each tunable choice for a short amount of trial time, and uses the convergence speed as the feedback.

Our BayesianOptSearcher uses an open-source Bayesian optimization implementation, called Spearmint [35]. The Bayesian optimization tool itself does not have a stopping condition of when to stop trying new choices, but our BayesianOptSearcher needs a stopping condition to terminate the searching loop. So we have to add our own stopping condition, which is when the convergence speed of the top five best tunable choices differ by less than 10%.

Since the BayesianOptSearcher proposes tunable choices based on the reported convergence speeds, it can cause problems if we directly use the progress slopes of the diverged branches as the convergence speeds. For example, suppose we have two diverged branches, we shouldn't consider the one with the smaller diverged losses as a better choice and closer to the optimal choice. Instead, we should treat all diverged branches as the same quality, so our BayesianOptSearcher will use zero as the convergence speeds of the diverged branches.

MarginalSearcher for adjusting tunables. When MLtuner adjusts the tunables during the training, it is able to take advantage of the fact that we know the original best tunable choice, and that the new best tunable choice should be close to it. MLtuner achieves that by using a variant of the GridSearcher, called *MarginalSearcher*.

Method name	**Input**	**Description**
	Messages sent from MLtuner	
`ForkBranch`	(clock, branchId, parentBranchId, tunable, [type])	fork a branch from a parent branch
`FreeBranch`	(clock, branchId)	free a branch
`ScheduleBranch`	(clock, branchId)	schedule the branch to run at clock
	Messages sent to MLtuner	
`ReportProgress`	(clock, progress)	report per-clock training progress

Table 1: MLtuner message signatures.

Instead of trying all possible tunable choices, the MarginalSearcher starts from the original best tunable choice, and changes only one of its dimensions. For example, suppose we have two tunables A and B to search, both have N possible values, so the GridSearcher will have to try N^2 possible tunable choices in total, which are $\{\{< A_i, B_j >\}_{i=1}^{N}\}_{j=1}^{N}$. If we know the original best tunable choice is $< A_s, B_s >$, the MarginalSearcher only needs to try $2N-1$ tunable choices, which are $\{< A_i, B_s >\}_{i=1}^{N} \cup \{< A_s, B_i >\}_{i=1}^{N}$.

4.4 Training system interface

MLtuner consumes very little CPU computing power and network bandwidth, so we can just run it on one of the training machines in the cluster. MLtuner communicates with the training system with messages sent via network sockets, and Table 1 lists the message signatures.

MLtuner identifies each branch with a unique *branch ID*, and uses *clock* to indicate logical time. When MLtuner forks a branch, it expects the training system to copy all the states (e.g., model parameters) from the parent branch to the child branch, and an empty parent branch ID is provided, when it wants to create a new branch from no parent. When MLtuner frees a branch, the training system can then reclaim all the resources (e.g., storage for model parameters) associated with that branch. All the branch operations will be sent in clock order, and exactly one `ScheduleBranch` message will be sent for every clock. The training system is expected to report its *training progress* with the `ReportProgress` message every clock.

Although in our auto-tuning design, the branches are scheduled based on time, instead of clocks, MLtuner still sends the per-clock branch schedules to the training system. We made this design choice, in order to ease the modification of the training systems. To make sure each branch runs for (approximately) the amount of scheduled trial time, MLtuner will first schedule that branch to run for a small number of clocks (e.g., three), and measure the per-clock time, and MLtuner will then schedule it to run for more clocks, based on the measured per-clock time.

Distributed training support. The large-scale machine learning tasks are often trained on distributed training systems (e.g., with a parameter server architecture). The distributed training system will have multiple training workers, and MLtuner will broadcast the branch operations to all the training workers (with the operations in the same order). MLtuner also allows each of the training workers to report their training progress separately, and MLtuner will aggregate the training progress from all the workers with a user-defined aggregation function. For the example applications we used in this paper, this aggregation function simply adds together the training progress of all the workers. This aggregation is useful, because, when we do data-parallel training with SGD, each training worker trains on a distinct partition of the training data, and the objective value of the whole model is simply the summation of the objective values of all the partitions.

Evaluating the model on validation set. In some applications, such as image classification, the quality of the model and the convergence criteria are defined as the classification accuracy on a set of validation data, instead of the training loss. This can be easily done with MLtuner. To test the model on the validation set, MLtuner will fork a branch with a special `TESTING` flag as the *branch type*, telling the training system to use this branch to do the testing. The reported progress of the testing branch will be the validation accuracy.

4.5 Training system modifications

This section describes the possible modifications to be made, for a training system to work with MLtuner. The modified training system will be able to keep multiple versions of the training states (e.g., model parameters and local states), and make consistent decisions on which version of the training states to use.
We have modified two training systems to work with MLtuner. The first one is a CPU-based parameter server system that is very similar architecturally to [45, 8, 25]. The second one is GeePS [9], a recently released parameter server for deep learning GPU applications [1].
Both parameter server implementations store the parameter data as key-value pairs in memory, sharded across all the machines. We modified their parameter data storage to keep multiple versions of the parameter data, by adding the branch ID as another index. When a new branch is forked, we will allocate its data storage and copy the data from its parent branch. When the branch is freed, we will free its memory. MLtuner tries to keep as few active branches as possible, and except when exploring the trial time (with Algorithm 1), MLtuner usually needs only three active branches to be kept in memory, the parent branch, the current best branch, and the current trial branch. Because the parameter data is sharded across all the machines, keeping extra copies of the parameter data is usually not an issue. For our example applications, we find the memory is sufficient for keeping at least 50 copies of the parameter data in memory.
Those parameter server implementations also have multiple levels of caches, for worker machines to cache the parameter data locally, and since MLtuner runs only one branch at a time, the branches can share the cache memory. In fact, sharing the cache memory is critical for the GeePS system work on MLtuner, because GeePS caches parameter data in GPU memory, and since GPU memory is a scarce resource, we usually cannot afford having one GPU cache for every branch.

5 Evaluation

This section evaluates our auto-tuning design with the MLtuner system on several real ML applications and models. The results confirm that MLtuner can robustly pick and adjust the tunables for ML tasks, with reasonable overhead. In some applications, such as deep neural network training, adjusting tunables during the training results in higher model accuracies than the state-of-art learning rate tuning algorithms.

5.1 Experimental setup

5.1.1 Application setup

Our experiments used three applications, *image classification* with deep convolutional neural network, *video classification* with deep recurrent neural network, and *movie recommendation* with matrix factorization. Table 2 summaries their distinct characteristics (e.g., supervised vs. unsupervised, one mini-batch per clock vs. a whole training data pass per clock, trained with GPUs vs. trained with CPUs).
Image classification w/ convolutional neural network. The image classification application is a supervised learning task that classifies images (raw pixel maps) into pre-defined labels. The model is trained on a set of

[1]We used the open-sourced GeePS code from `https://github.com/cuihenggang/geeps` as of June 3, 2016.

Application	Model	Supervised/Unsupervised	Clock size	Hardware
Image classification	Convolutional neural network	Supervised learning	One mini-batch	GPU
Video classification	Recurrent neural network	Supervised learning	One mini-batch	GPU
Movie recommendation	Matrix factorization	Unsupervised learning	Whole data pass	CPU

Table 2: Applications used in the experiments. They have distinct characteristics.

training images with known labels. The state-of-art approach for this task is to use the deep convolutional neural networks (CNNs) [23, 39, 19, 40, 16]. The deep neural network model consists of multiple layers of interconnected neurons. The first layer of the neurons (input of the network) are the raw pixels of the input image, and the last layer of the neurons (output of the network) are the probabilities that this image should be assigned to each label. There is a *weight* associated with each neuron connection, and those weights are the parameters of this model and will be trained from the training data.

The deep neural networks are often trained with the SGD algorithm, but because the training data is often quite large (e.g., one million labelled images), people often divide the training data into many *mini-batches*, and compute the gradients and parameter updates with only one mini-batch of the training data at a time [10, 6, 9, 23, 39, 19, 40, 16]. As an optimization, people often smooth the gradients across mini-batches with a method called *momentum* [38], and we also did that.

We used two datasets and two models for the image classification experiments. The first dataset is Large Scale Visual Recognition Challenge 2012 (ILSVRC12) [33], which has 1.3 million training images labelled to 1000 classes, and 5000 validation images. For this dataset, we used the Inception-BN [19] network as the model. [2] The second dataset is Cifar10 [22], which has 50,000 training images labelled to 10 classes, and 10,000 validation images. We used the AlexNet [23] network for the Cifar10 dataset.

Image classification is a supervised classification task, and the quality of the trained model is evaluated by testing it on a set of validation data, in terms of classification accuracy. The model convergence is defined as when the validation accuracy plateaus (i.e., stops increasing). In our ILSVRC12 experiments, we test the model, each time we finish a whole pass over the training data (i.e., every *epoch*). For the Cifar10 experiments, we tested the model every five epoches, because its training set is quite small compared to the validation set. We declare the model is converged, when the validation accuracy plateaus over the last five tests (i.e., when the maximum validation accuracy was observed more than five tests ago). We did the image classification experiments on our modified GeePS parameter server linked with MLtuner.

Video classification w/ recurrent neural network. To capture the sequence information of the video, video classification tasks often use a structure called *recurrent neural network* (RNN). The RNN network often uses a special type of neuron layers called *Long-Short Term Memory* (LSTM) [18] as the building blocks [11, 44, 46]. A common approach for using RNNs on video classification tasks is to first encode each image frame of the video with a convolutional neural network, and then feed the encoded image feature vector sequences into the RNN network.

Our video classification experiments used the UCF-101 dataset [36], which contains about 8,000 training videos and 4,000 testing videos, categorized into 101 human action classes. Similar to the approach described by Cui et al. [9] and Donahue et al. [11]. we used the GoogLeNet [39] convolutional neural network, trained with the ILSVRC12 image data, to encode the image frames, and fed the feature vectors into a RNN network with LSTM layers. We extracted the video frames at a rate of 30 frames per second and trained the model

[2]The Inception-BN paper [19] did not release some minor details of the model, and we used an open-sourced version of the Inception-BN model from a popular open-sourced deep learning system, MXNet [5].

Tunable	Valid range
Learning rate	10^x, where $x \in [-5, 0]$ for BayesianOptSearcher, $x \in \{-5, -4.5, ..., 0\}$ for GridSearcher
Data staleness	$\{0, 1, 3, 7\}$
Mini-batch size	Inception-BN: $\{2, 4, 8, 16, 32\}$ AlexNet: $\{4, 16, 64, 256\}$ Video classification: $\{1\}$ Matrix Factorization: N/A

Table 3: Tunable setups in the experiments.

with randomly selected video clips of 32 frames each. We did the video classification experiments also on our modified GeePS parameter server linked with MLtuner.

Movie recommendation with matrix factorization. The movie recommendation task tries to predict how much a user will like a movie, based on lots of existing user-movie ratings. The movie recommendation application is often modelled as a *sparse matrix factorization* problem, where we have a partially filled matrix X, with entry $< i, j >$ being user i's rating of movie j, and we want to factorize X into two low ranked matrices L and R, such that their product approximates X (i.e., $X \approx L \times R$) [14]. The matrix factorization model is often trained with the SGD algorithm [14], and because the model parameter values are updated with uneven frequency, people often use AdaGrad [12] or AdaRevision [27] to decide the per-parameter learning rate adjustment, based on an initial learning rate [45].

Our matrix factorization (MF) experiments used the Netflix dataset, which has 100 million known ratings from 480 thousand users for 18 thousand movies. We factorized the rating matrix with a rank of 400, which is the same setting used by Wei et al. [45], and we set the convergence criteria as fixed loss threshold of 2e-7, which is also the same as the one used by Wei et al. [45]. [3] The matrix factorization experiments used on our modified CPU-based parameter server linked with MLtuner.

5.1.2 MLtuner setup

Table 2 summarizes the tunables to be searched in our experiments. The image classification and video classification tasks have three tunables, learning rate, data staleness bound, and mini-batch size. The matrix factorization task has two tunables, learning rate and data staleness bound. The tunable ranges for the learning rate and data staleness bound are the same for all applications and models, because we assume the user has no prior knowledge about how to choose the tunables. The mini-batch size range is different for each model, which is based on the maximum mini-batch size that can fit in the GPU memory. For the video classification task, we can only fit one video in one mini-batch, so the mini-batch size is fixed to one.

MLtuner decides whether the training progress of each application is noisy or not, based on its initial progress. For noisy cases (which will be the neural network applications), it downsamples each progress trace into 10 samples, and for non-noisy cases (which will be the matrix factorization application) it downsamples each progress trace into 3 samples.

MLtuner adjusts the tunables when the training branch has been running for more than $10\times$ of the tunable searching time, or (only for the neural network applications) when the model hits the convergence criteria,

[3] We did not define the convergence criteria in terms of loss change between iterations, because different tunable choices will greatly affect the per-iteration convergence rate, and thus make the model "converge" at different losses.

which is when the validation accuracy plateaus for the last five tests. We bound the time for doing each adjustment to be less than 50% of the current runtime, and stop the adjustment early when exceeding the bound. That is because, if the model has already converged, we will not be able to get further converging progress with any tunable choice, leaving the adjustment time unbounded will cause the training to run forever. If the adjustment is triggered by the convergence criteria, we will test the model on the validation set for one more time, after the adjustment, and stop the training if the peak accuracy does not change.
By default, MLtuner uses the BayesianOptSearcher, but for comparison purpose, some of our experiments also show the result of using the GridSearcher. The MarginalSearcher is used for all tunable adjustments.

5.1.3 Cluster setup

There are two clusters used in our experiments. We used a *GPU cluster* for the deep neural network experiments, and a *CPU cluster* for the matrix factorization experiments.
The GPU cluster has 8 machines. Each machine has a NVIDIA Titan X GPU, with 12 GB of GPU device memory. In addition to the GPU, each machine has one E5-2698Bv3 Xeon CPU (2.0 GHz, 16 cores with 2 hardware threads each), and 64 GB of RAM, and is installed with 64-bit Ubuntu 16.04, CUDA toolkit 8.0, and cuDNN v5. The machines are inter-connected via a 40 Gbps Ethernet interface.
The CPU cluster has 32 machines. Each machine has four quad-core AMD Opteron 8354 CPUs (16 physical cores in total) and 32 GB of RAM, and is installed with 64-bit Ubuntu 14.04. The machines are inter-connected via a 20 Gb Infiniband interface.

5.2 Auto-tuning with MLtuner

This section evaluates the robustness and the overhead of auto-tuning with MLtuner, on our example machine learning applications.

5.2.1 ILSVRC12 image classification w/ Inception-BN

In this set of experiments, we trained the Inception-BN network on the ILSVRC12 dataset. We will compare the validation accuracies of training with auto-tuning, and training with the state-or-art SGD learning rate tuning algorithms, including AdaGrad [12], AdaRevision [27], AdaDelta [47], RMSProp [42], Nesterov [29], and Adam [20]. Because these SGD learning rate tuning algorithms still require the selection of the initial learning rate, we used auto-tuning (with GridSearcher) to pick the initial tunable choice (including learning rate, batch size, and data staleness bound) for those setups, but we only adjust the tunables for the auto-tuning setups. We have also manually verified that the initial learning rates picked for the SGD learning rate tuning algorithms are very close to ideal. We did three runs for each of the two auto-tuning setups. Table 4 summaries the achieved model accuracies after convergence, and Figure 5 shows the convergence trace.
The results show that adjusting the tunables improves the model accuracy. This result echoes people's finding that, when training deep neural networks, it is necessary to decrease the learning rate during the training [40, 19, 39, 23, 16, 1, 48]. When the validation accuracy plateaus, it is not necessarily because the model has converged, but sometimes it's because the current learning rate is too large for the SGD algorithm to make any further progress. Auto-tuning is able to get around 69.5% validation accuracy on this task, which is the same as (actually a little bit higher than) the reported validation accuracies for the same task by open-sourced systems, such as MXNet [5].
Surprisingly, none of the SGD learning rate tuning algorithms is able to achieve the expected model accuracy. Actually in the original papers of those algorithms, none of them evaluated their performance on this

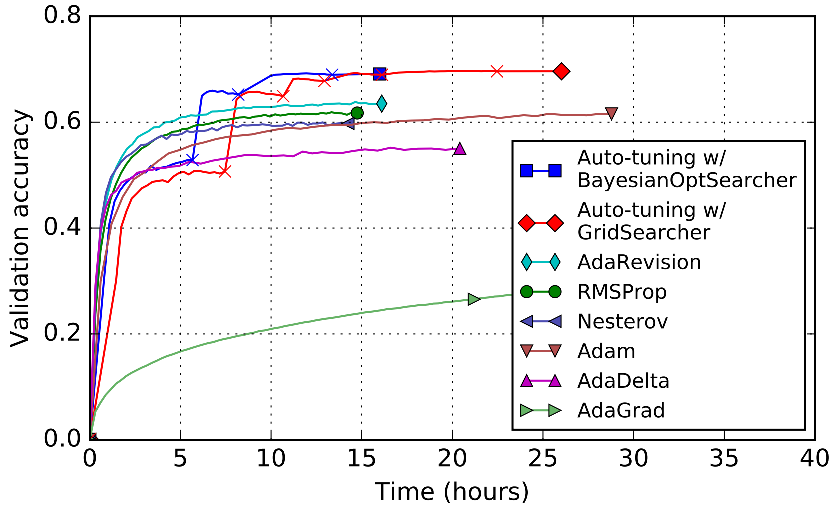


Figure 5: ILSVRC12 with Inception-BN validation accuracies. We only show the result of the first run for each setup, and the other runs have similar behavior. The "x" markers show the point where we adjust the tunables.

ILSVRC12 image classification task. They only showed results on small datasets, such as Cifar10 or MNIST [24]. We think the reason why they don't work well for this task is that, because of the complexity of the model and the size of the data, this task needs carefully scheduled learning rate decrease during the training, in order to get *full convergence*. Those algorithms, however, are designed mostly for adjusting the relative learning rates of each individual model parameter values. They try to use relatively large learning rates for the model parameters with small gradients, and relatively small learning rates for the model parameters with large gradients. They are shown to work well for sparse problems, such as matrix factorization [45], but are not good at adjusting the overall global learning rate according to the model convergence. This problem has already been found by many other people. For example, in the experimental setups described by Szegedy et al. [40], they still needed to decrease the global learning rate during the training, even though they were using the RMSProp algorithm.

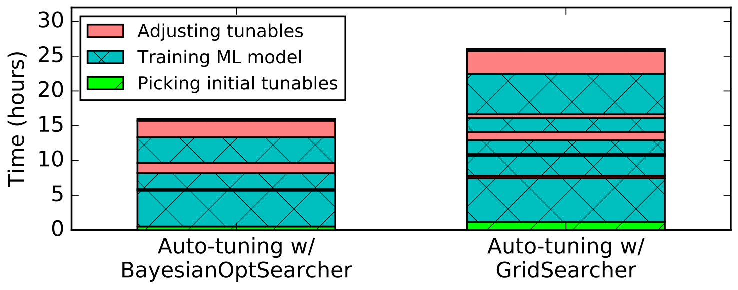


Figure 6: ILSVRC12 Inception-BN runtime break down.

Figure 6 summaries the overhead of auto-tuning, by breaking the total runtime into three parts: time for picking the tunable choice at the beginning, time for adjusting the tunables, and the actual training time with

Setup	Accuracy (%)
AdaGrad [12]	30.8
AdaRevision [27]	63.8
AdaDelta [47]	55.2
RMSProp [42]	60.0
Nesterov [29]	60.0
Adam [20]	61.6
Auto-tuning w/ GridSearcher	**69.8 $\pm$ 0.3**
Auto-tuning w/ BayesianOptSearcher	**69.0 $\pm$ 0.3**

Table 4: ILSVRC12 with Inception-BN validation accuracies. We did three runs for each auto-tuning setups, and showed their average accuracy and standard deviation.

the selected tunables. The results show that auto-tuning has relatively low overhead, and with either the BayesianOptSearcher or the GridSearcher, the time for picking the initial tunable choice is negligible. With the default BayesianOptSearcher, MLtuner spent 11.2 hours on the actual training, and 4.8 hours on searching or adjusting the tunables (40% extra runtime). MLtuner with GridSearcher spent 18.7 hours training, and 7.3 hours tuning (39% extra runtime).
From Figure 6, we find a major part of the tuning time is spent on the last tunable adjustment. That is because MLtuner always tries to adjust the tunables when the model hits the convergence criteria, hoping to get further convergence with some other tunable choice. It keeps doubling the trail time, until it finds a converging branch. However, if the model has already converged, none of the branches will have any further convergence, so this searching procedure can end up taking very long (or even forever, if we don't bound the searching time). If we only look at the time for MLtuner to reach the converged accuracy, without counting the last adjustment attempt after convergence, auto-tuning with BayesianOptSearcher has only 20% searching overhead, and auto-tuning with GridSearcher has only 18% searching overhead. The GridSearcher setup has longer training time, because it had longer searching time at the beginning and started the adjustments later, so it was less likely to give up and declare convergence during the tunable adjustments.

5.2.2 Cifar10 image classification with AlexNet

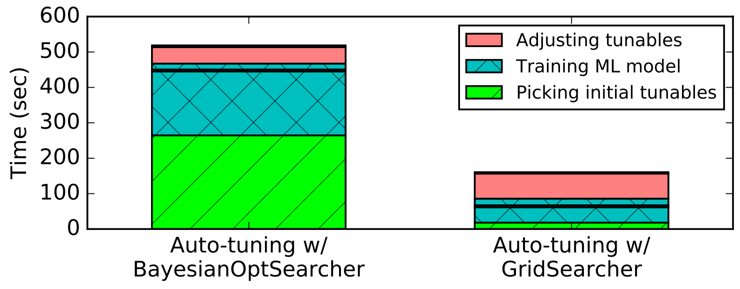


Figure 7: Cifar10 AlexNet runtime break down. Only the first run of each setup is shown, and the other runs have similar behavior.

In this set of experiments, we trained the AlexNet network on the Cifar10 dataset. We did five runs for each auto-tuning setup, from different initial model parameter states, all shown in Figure 8. Figure 7 shows the runtime break down. Similar to the ILSVRC12 task, most of the tunable adjustment time is spent on the last adjustment attempt after convergence. If we include the last adjustment attempt, auto-tuning with BayesianOptSearcher has 178% overhead, and auto-tuning with GridSearcher has 176% overhead. If we don't include the last adjustment attempt, because it did not improve the model accuracy at all, auto-tuning with BayesianOptSearcher has 108% overhead, and auto-tuning with GridSearcher has 51% overhead. Surprisingly, the GridSearcher has less searching overhead than the BayesianOptSearcher, for picking the tunable choice at the beginning. That is because, for this relatively small dataset, MLtuner is able to identify the good tunable choices with very short trial time. The GridSearcher spends only about 0.1 seconds to try each tunable choice, while the BayesianOptSearcher needs to run the Bayesian optimization algorithm to generate each trial tunable choice, and that takes up to 10 seconds. As a result, even though the GridSearcher tried more tunable choices than the BayesianOptSearcher did (176 vs. an average of 30 for picking the initial choice), it spent less time searching. Previously, people have found tuning machine learning with grid searching is inefficient [35, 37], because, to evaluate each hyperparameter choice, they needed to train the model to completion. MLtuner, on the other hand, tries each tunable choice with just a short amount of trial time, so grid searching becomes acceptable and sometimes even superior. The BayesianOptSearcher setup has longer training time, because MLtuner decides when to adjust tunables based on the searching time, so the BayesianOptSearcher setup spent more time running the initial tunable choice before adjustment.

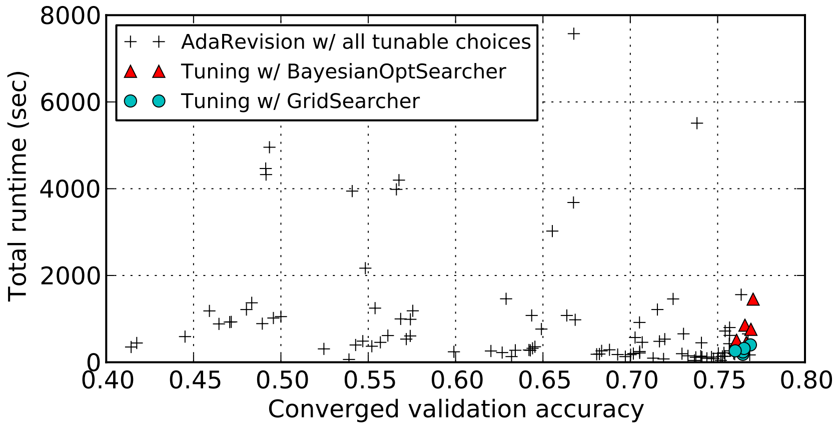
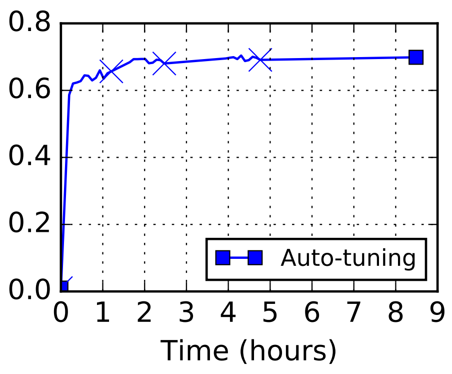


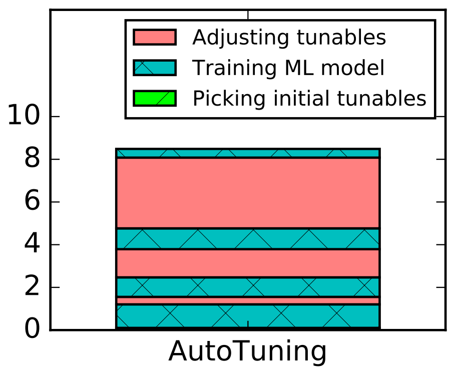
Figure 8: Cifar10 AlexNet converged accuracy and convergence time. The baseline used AdaRevision with each of the all 176 tunable choices in range, and 79 of them diverged (not shown in this graph). We did five runs for each of the auto-tuning setups, and showed the results of all of them in the graph.

Even though MLtuner needs to spend extra time tuning the tunables, we think being able to automatically find the good tunables is worth this cost. The ideal training time is actually not achievable by any constant tunable choice. Instead, it can only be achieved by changing the tunables during the training. Figure 8 shows the total runtimes and the converged model accuracies of using each of the all 176 tunable choices in range (with discretized learning rate). In order to compare with the state-of-art approach, we used AdaRevision [27] to tune the learning rates for this baseline, and compared it with auto-tuning. The result shows that, even with AdaRevision, the tunable choice not only affects the convergence time, but also affects the converged model accuracy a lot. By picking the good tunable choices and adjusting them on the fly, our auto-tuning setups robustly converged to the best model accuracy, and the convergence times are close to that achieved by AdaRevision using the ideal best tunable choice.

5.2.3 Video classification



(a) Validation accuracy.



(b) Runtime (hours).

Figure 9: Video classification with RNN.

This set of experiments shows the performance of MLtuner on the video classification task. Figure 9(a) shows model accuracy trace. By automatically picking and adjusting tunables, MLtuner reached a validation accuracy of 70.4%, which is the same as the accuracy number reported by Cui et al [9]. Figure 9(b) summaries the runtime break down. MLtuner spent about 1.7 hours tuning before convergence, 3.4 hours training, and 3.3 hours doing the last adjustment attempt after convergence. The tuning overhead is 147% if we include the last adjustment attempt, or 50% if we don't include the last adjustment attempt.

5.2.4 Matrix factorization

Because the model parameters of the MF task has uneven update frequency, we used AdaGrad [12] to decide the per-parameter learning rates, and used auto-tuning to select the initial learning rate for AdaGrad, as well as the data staleness bound. The MF task uses a loss threshold of 2e-7 as the convergence criteria, so we will just compare the time for each setup to reach this loss threshold, and we do not adjust tunables after convergence. We did three runs for each auto-tuning setup and showed their average and standard deviation. Figure 10 shows the runtime break down of training with auto-tuning, and training with the ideal best tunable choice. Considering both the overhead of tunable searching and the extra training time for using suboptimal tunable choices, auto-tuning spent 155% more time than ideal. Compared to the neural network applications, auto-tuning has a slightly higher overhead for this MF task, because the MF task converges in only about 75 clocks (iterations), while our MLtuner system needs to try each tunable choice for at least 3 clocks to measure its convergence speed. We think a possible fix to this issue is to break the whole iteration into multiple clocks, and let the training system report the training progress more frequently, so that MLtuner will be able to identify good tunable choices in shorter time.
Even though MLtuner spent around double the ideal runtime, the ideal best tunable choice is usually not achievable through manual tuning, and using bad tunable choices can cost much longer runtime. Figure 11

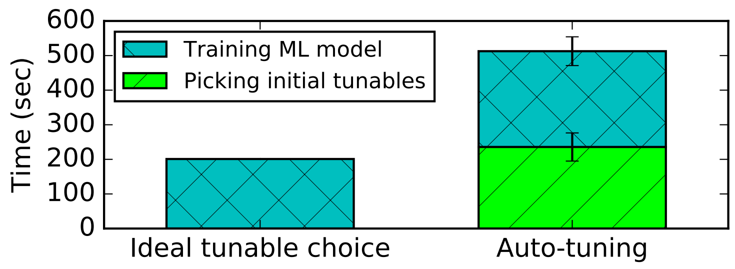


Figure 10: MF runtime break down.

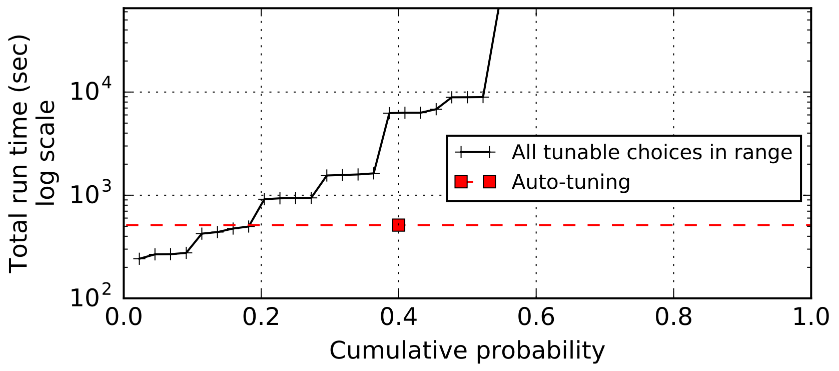


Figure 11: MF runtimes with different tunable choices, and with auto-tuning. The graph shows the cumulative probabilities of finishing the task in less than certain runtime.

shows the runtimes of using each of the 44 tunable choices in range (with discretized learning rate). The result shows that, even considering the searching overhead, auto-tuning converges 10× faster than 55% of the tunable choices in range, and 100× faster than 45% of the tunable choices in range. So being able to automatically find the good tunable choices is worth the tuning cost.

5.3 Comparing to hyperparameter optimization

This section explicitly compares our auto-tuning approach with the traditional hyperparameter optimization (HyperparamOpt) approach, where each tunable choice is evaluated by training the model to completion. Similar to the approach described by Snoek et al. [35], our HyperparamOpt experiments used Bayesian optimization to generate the trial tunable choices. We used their open-sourced Bayesian optimization implementation, called Spearmint [35]. Since Snoek et al. [35] did not provide a stopping condition for deciding when to stop the searching, we used the same stopping condition as is used by our BayesianOptSearcher, which is when the convergence accuracies of the top five best tunable choices differ by less than 10%.
Table 5 summarizes the total task completion times with the HyperparamOpt approach, and and with auto-tuning (using the BayesianOptSearcher). For the Cifar10 image classification task, HyperparamOpt spent 5000 seconds, trying 11 tunable choices (each running to completion), and auto-tuning is faster than HyperparamOpt by 8× (and 22× faster with the GridSearcher). HyperparamOpt is not able to finish the ILSVRC12 task or the matrix factorization task in reasonable amount of time, because, for both tasks,

Application	Auto-tuning	HyperparamOpt
ILSVRC12	14 hours	>35 hours
Cifar10	620 seconds	5000 seconds
MF	510 seconds	>16 hours

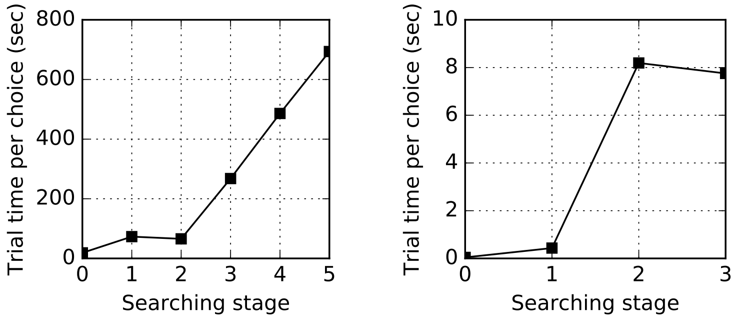
Table 5: Total task completion time with auto-tuning and with HyperparamOpt.

HyperparamOpt spent more than 10 hours training with the first tunable choice proposed by Bayesian optimization, and, unfortunately, this first proposed tunable choice had a learning rate of 10^{-5}, which was too small and caused extremely long convergence time.

5.4 Discussion of design choices

This section justifies some of our design choices.

Adjust tunables with the MarginalSearcher. As is mentioned in Section 4.3, MLtuner uses MarginalSearcher to adjust the tunables during the training. That is because the per-chocie tunable trial time is so long that we cannot afford using the BayesianOptSearcher or the GridSearcher.



(a) ILSVRC12. (b) Cifar10.

Figure 12: Tunable trial time per choice for each searching stage. The searching stage #0 stands for picking the initial tunable choice at the beginning, and the other searching stages stand for adjusting the tunables. Both graphs used the GridSearcher to pick the initial tunable choice, because we don't want the trial time to be affected by the long decision time of the BayesianOptSearcher.

Figure 12 summarizes the per-choice trial times of searching the initial tunable choice at the beginning, and adjusting the tunables. The result shows that, when we adjust the tunables, the per-choice tunable trial time can be over an order of magnitude longer than picking the initial tunables (37$\times$ longer for ILSVRC12, and 200 $\times$ longer for Cifar10). That is because, the training progress slopes are usually much flatter when MLtuner adjusts the tunables than at the beginning, so we need much longer trial times to get stable converging progress.

6 Conclusions

MLtuner automatically tunes the tunables that can have major impact of the performance and effectiveness of ML applications. Experiments with three real ML applications on two real ML systems show that it works, outdoing most reasonable settings and performing within 40–178% of having manually chosen the best settings (but without the risk). It does better in cases, like the DNN examples, where runtime changes to settings leads to better outcomes than any static settings.

References

[1] Training deep net on 14 million images by using a single machine. `http://mxnet.readthedocs.io/en/latest/tutorials/imagenet_full.html`.

[2] A. Ahmed, M. Aly, J. Gonzalez, S. Narayanamurthy, and A. J. Smola. Scalable inference in latent variable models. In *WSDM*, 2012.

[3] J. Bergstra, D. Yamins, and D. D. Cox. Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. *ICML (1)*, 28:115–123, 2013.

[4] J. S. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl. Algorithms for hyper-parameter optimization. In *Advances in Neural Information Processing Systems*, pages 2546–2554, 2011.

[5] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang. MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

[6] T. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman. Project Adam: Building an efficient and scalable deep learning training system. In *OSDI*, 2014.

[7] H. Cui, J. Cipar, Q. Ho, J. K. Kim, S. Lee, A. Kumar, J. Wei, W. Dai, G. R. Ganger, P. B. Gibbons, G. A. Gibson, and E. P. Xing. Exploiting bounded staleness to speed up big data analytics. In *USENIX ATC*, 2014.

[8] H. Cui, A. Tumanov, J. Wei, L. Xu, W. Dai, J. Haber-Kucharsky, Q. Ho, G. R. Ganger, P. B. Gibbons, G. A. Gibson, and E. P. Xing. Exploiting iterative-ness for parallel ML computations. In *SoCC*, 2014.

[9] H. Cui, H. Zhang, G. R. Ganger, P. B. Gibbons, and E. P. Xing. GeePS: Scalable deep learning on distributed GPUs with a GPU-specialized parameter server. In *Proceedings of the Eleventh European Conference on Computer Systems*, 2016.

[10] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, A. Senior, P. Tucker, K. Yang, Q. V. Le, et al. Large scale distributed deep networks. In *NIPS*, 2012.

[11] J. Donahue, L. A. Hendricks, S. Guadarrama, M. Rohrbach, S. Venugopalan, K. Saenko, and T. Darrell. Long-term recurrent convolutional networks for visual recognition and description. *arXiv preprint arXiv:1411.4389*, 2014.

[12] J. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, (Jul), 2011.

[13] M. Feurer, A. Klein, K. Eggensperger, J. Springenberg, M. Blum, and F. Hutter. Efficient and robust automated machine learning. In *Advances in Neural Information Processing Systems*, pages 2962–2970, 2015.

[14] R. Gemulla, E. Nijkamp, P. J. Haas, and Y. Sismanis. Large-scale matrix factorization with distributed stochastic gradient descent. In *SIGKDD*, 2011.

[15] G. Gibson, G. Grider, A. Jacobson, and W. Lloyd. PRObE: A thousand-node experimental cluster for computer systems research. *USENIX ;login:*, 2013.

[16] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. *arXiv preprint arXiv:1512.03385*, 2015.

[17] Q. Ho, J. Cipar, H. Cui, S. Lee, J. K. Kim, P. B. Gibbons, G. A. Gibson, G. R. Ganger, and E. P. Xing. More effective distributed ML via a Stale Synchronous Parallel parameter server. In *NIPS*, 2013.

[18] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8), 1997.

[19] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

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

[21] B. Komer, J. Bergstra, and C. Eliasmith. Hyperopt-sklearn: automatic hyperparameter configuration for scikit-learn. In *ICML workshop on AutoML*, 2014.

[22] A. Krizhevsky and G. Hinton. Learning multiple layers of features from tiny images. 2009.

[23] A. Krizhevsky, I. Sutskever, and G. E. Hinton. ImageNet classification with deep convolutional neural networks. In *NIPS*, 2012.

[24] Y. LeCun, C. Cortes, and C. J. Burges. The mnist database of handwritten digits, 1998.

[25] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su. Scaling distributed machine learning with the parameter server. In *OSDI*, 2014.

[26] D. Maclaurin, D. Duvenaud, and R. P. Adams. Gradient-based hyperparameter optimization through reversible learning. In *Proceedings of the 32nd International Conference on Machine Learning*, 2015.

[27] B. McMahan and M. Streeter. Delay-tolerant algorithms for asynchronous distributed online learning. In *NIPS*, 2014.

[28] J. Močkus. On bayesian methods for seeking the extremum. In *Optimization Techniques IFIP Technical Conference*. Springer, 1975.

[29] Y. Nesterov. A method of solving a convex programming problem with convergence rate o (1/k2). In *Soviet Mathematics Doklady*, 1983.

[30] J. Ngiam, A. Coates, A. Lahiri, B. Prochnow, Q. V. Le, and A. Y. Ng. On optimization methods for deep learning. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pages 265–272, 2011.

[31] F. Pedregosa. Hyperparameter optimization with approximate gradient. *arXiv preprint arXiv:1602.02355*, 2016.

[32] R. Power and J. Li. Piccolo: Building fast, distributed programs with partitioned tables. In *OSDI*, 2010.

[33] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei. ImageNet large scale visual recognition challenge. *International Journal of Computer Vision*, 2015.

[34] A. Senior, G. Heigold, K. Yang, et al. An empirical study of learning rates in deep neural networks for speech recognition. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 6724–6728. IEEE, 2013.

[35] J. Snoek, H. Larochelle, and R. P. Adams. Practical bayesian optimization of machine learning algorithms. In *NIPS*, 2012.

[36] K. Soomro, A. R. Zamir, and M. Shah. Ucf101: A dataset of 101 human actions classes from videos in the wild. *arXiv preprint arXiv:1212.0402*, 2012.

[37] E. R. Sparks, A. Talwalkar, D. Haas, M. J. Franklin, M. I. Jordan, and T. Kraska. Automating model search for large scale machine learning. In *SoCC*, 2015.

[38] I. Sutskever, J. Martens, G. E. Dahl, and G. E. Hinton. On the importance of initialization and momentum in deep learning. *ICML*, 2013.

[39] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. *arXiv preprint arXiv:1409.4842*, 2014.

[40] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna. Rethinking the inception architecture for computer vision. *arXiv preprint arXiv:1512.00567*, 2015.

[41] C. Thornton, F. Hutter, H. H. Hoos, and K. Leyton-Brown. 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*, pages 847–855. ACM, 2013.

[42] T. TielemanWang and G. Hinton. Divide the gradient by a running average of its recent magnitude. COURSERA: Neural Networks for Machine Learning, 2012.

[43] M. Vartak, P. Ortiz, K. Siegel, H. Subramanyam, S. Madden, and M. Zaharia. Supporting fast iteration in model building. *NIPS ML Systems Workshop*, 2015.

[44] O. Vinyals, A. Toshev, S. Bengio, and D. Erhan. Show and tell: A neural image caption generator. *arXiv preprint arXiv:1411.4555*, 2014.

[45] J. Wei, W. Dai, A. Qiao, Q. Ho, H. Cui, G. R. Ganger, P. B. Gibbons, G. A. Gibson, and E. P. Xing. Managed communication and consistency for fast data-parallel iterative analytics. In *SoCC*, 2015.

[46] J. Yue-Hei Ng, M. Hausknecht, S. Vijayanarasimhan, O. Vinyals, R. Monga, and G. Toderici. Beyond short snippets: Deep networks for video classification. In *CVPR*, 2015.

[47] M. D. Zeiler. Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.

[48] H. Zhang, Z. Hu, J. Wei, P. Xie, G. Kim, Q. Ho, and E. Xing. Poseidon: A system architecture for efficient GPU-based deep learning on multiple machines. *arXiv preprint arXiv:1512.06216*, 2015.