TOWARDS SELF-TUNING PARAMETER SERVERS

LIU CHUN YIN

MPhil

The Hong Kong Polytechnic University

2018

The Hong Kong Polytechnic University

Department of Computing

Towards Self-Tuning Parameter Servers

Liu Chun Yin

A thesis submitted in partial fulfillment of the requirements

for the degree of

Master of Philosophy

November 2017

CERTIFICATE OF ORIGINALITY

I hereby declare that this thesis is my own work and that, to the best of my knowledge and belief, it reproduces no material previously published or written, nor material that has been accepted for the award of any other degree or diploma, except where due acknowledgement has been made in the text.

. . . .

Liu Chun Yin

November 2017

Abstract

Machine Learning (ML) has driven advances in many applications in recent years. Nowadays, it is common to see industrial-strength machine learning jobs that involve billions of model parameters, petabytes of training data, and weeks of training. Good efficiency, i.e., fast completion time of running a specific ML job, therefore, is a key feature of a successful ML system. While the completion time of a long-running ML job is determined by the time required to reach model convergence, practically that is largely influenced by the values of various system settings. In this thesis, we present techniques towards building *self-tuning parameter servers*. Parameter Server (PS) is a de-facto system architecture for large-scale machine learning; and by self-tuning we mean while a long-running ML job is iteratively training the expert-suggested model, the system is also iteratively learning which setting is more efficient for that job and applies it online. We have implemented our three techniques, namely, (1) online ML job optimization framework, (2) online ML job progress estimation, and (3) online ML system reconfiguration, on top of TensorFlow. Experiments show that our techniques can reduce the completion times of long-running TensorFlow jobs from $1.7\times$ to $5.1\times$.

Acknowledgements

It is a pleasure to thank the many people who helped me a lot during my study.

First and foremost I would like to thank my former supervisor, Dr. Eric Lo. He has been supportive since I was a degree student until now. He opened my eyes to do research and guided me a lot during my study period. He has given me chances to participate in different kind of research projects. With his patience, his encouragement and advice, I have learnt a lot about how to do a meaningful research and how to express ideas in an efficient way.

I would like to thank my chief supervisor, Dr. Ken Yiu, who take care of my last half of study period without hesitation. He has given me a lot of advice on my research.

I would like to express my gratitude to my parents for their support over my whole life. This thesis would not exist without them.

I also thank Dr. Ben Kao and Dr. Kevin Yip for their trust and guidance. They shared their own research experience with me and gave me chances to participate in different projects.

Last but not least, I need to thank my colleagues for their assistance. Without them, I can 't complete my study in a stimulating and joyful environment. I am especially indebted to Bo Tang, Andy Shen, PengFei Zhang, XuXuan Zhou, Andy He, Petrie Wong, Qiang Zhang, Ziquang Feng, Wenjian Xu, Bai Ran. Bo was helpful for giving me advice on the alogrithms of this thesis.

Table of contents

List of Figures

List of Tables

Chapter 1

Introduction

Machine Learning (ML) has driven advances in many applications in recent years [2, 19, 67]. At the core of many ML jobs is an *iterative-convergent process* (Figure 1.1) — given an expert-suggested model, the model parameters are iteratively refined by computing the refinements based on the training data. Recently, the *Parameter Server* (PS) architecture [1, 18, 21, 46, 60, 79] has emerged as the de-facto system architecture to support large-scale distributed machine learning. The PS architecture advocates the separation of working units as "servers" and "workers", where the servers collectively maintain the model state and the workers duly "pull" the latest version of the model from the servers, scan their own part of training data to compute the model refinements, and "push" the model updates back to the servers for aggregation. The PS architecture has the beauty to eschew the network communications between the workers so that it can scale-out to both big model and big data.

With the PS architecture, nowadays industrial-strength machine learning

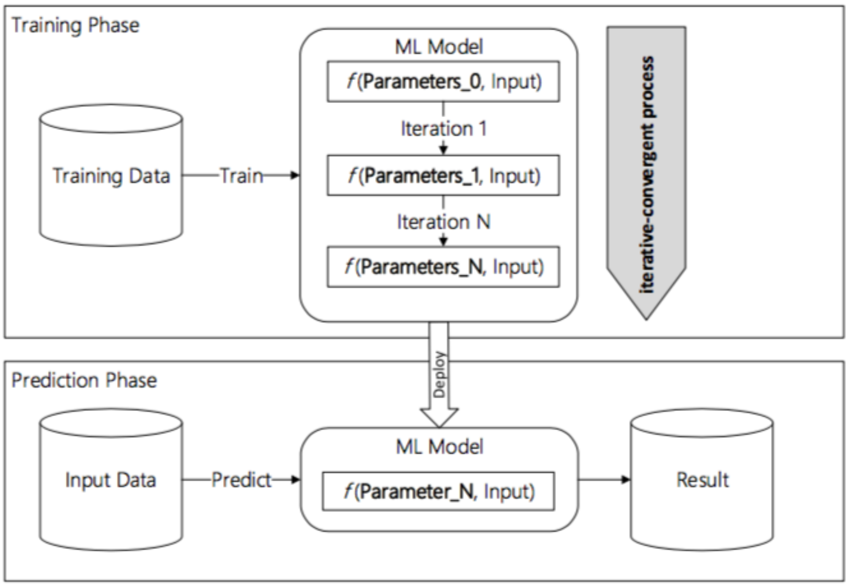


Figure 1.1. Iterative-convergent process

applications may involve petabytes of training data [1], billions of parameters [21], clusters of machines [84], thousands of iterations [64], and weeks of training [46]. Good efficiency, i.e., fast completion time of running a specific ML job, is therefore one of the key features of a successful ML system. Generally, the completion time of a long-running machine learning job is determined by the time required to reach model convergence. Practically, however, the completion time is largely influenced by the values of the various system knobs such as the server-worker ratio (i.e., how many hardware threads are dedicated to the servers and workers), the device placement (i.e., which operation shall be shipped to the GPU for processing and which shall stay in the CPU), and the parallelism degree (e.g., the model replication factor, the model partitioning scheme). Today, unfortunately, the burden falls on the users who submit the ML jobs to specify the knob values.

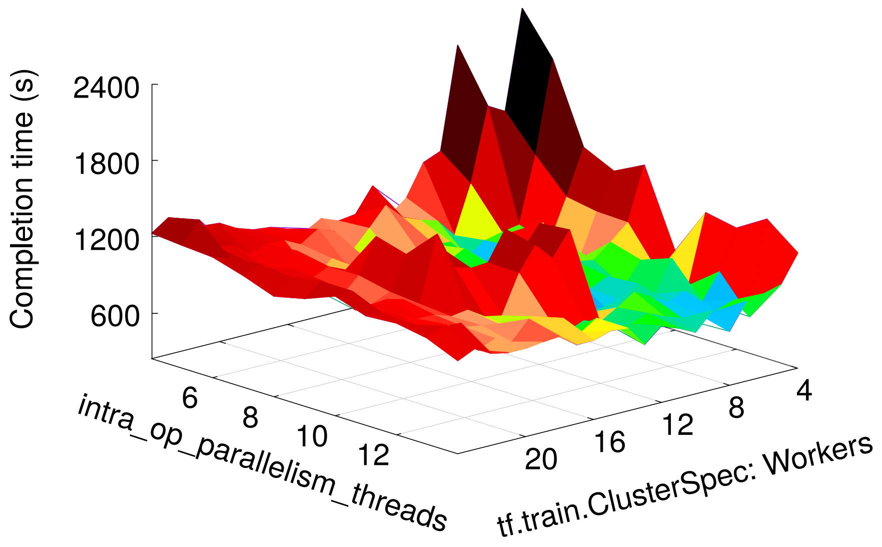


Figure 1.2. A 2D response surface for a Convolutional Neural Network TensorFlow **job**

Unearthing the right set of knob values that achieve optimal completion time has way surpassed human abilities. Part of what makes that so enigmatic is that the response surfaces of ML jobs are highly complex. Figure 1.2 shows a response surface of a Convolutional Neural Network job on CIFAR-10 dataset [40] running on TensorFlow [1], a PS-style ML system. The two system knobs involved are: *tf.train.ClusterSpec::worker* and *intra_op_parallelism_threads*, which respectively vary the server-worker ratio and the thread affinity of operations (i.e., the mapping between TensorFlow operations and hardware threads). We can observe that the response surface is complex and nonmonotonic, and the optimal lies at where human can't easily find. What adds to the challenge is that, the completion time of a ML job, unlike traditional data processing, is a complex interplay between *statistical efficiency* (how many iterations are needed until convergence to a given tolerance) and *hardware efficiency* (how efficiently

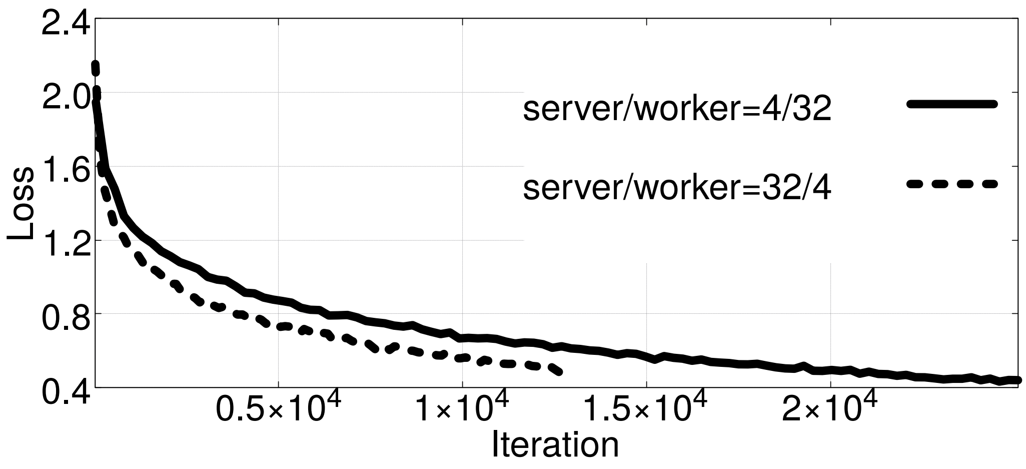


Figure 1.3. Different system settings would influence statistical efficiency

those iterations are carried out) [83]. Consider the server-worker ratio as an example. On the one hand, more workers would increase the hardware efficiency by having a higher degree of data parallelism. On the other hand, that might hurt the statistical efficiency when servers accept asynchronous updates from workers, since more concurrent workers update the global model would make the model more inconsistent, causing it more iterations to convergence. Figure 1.3 evidences such a case. It shows that varying just one system setting (different server-worker ratio) would already yield a 2.5× difference in statistical efficiency.

Configuring distributed ML systems to reduce the long-running execution time of ML jobs currently requires system expertise – something many ML users may lack. Even for system experts, the dependencies between the knobs (e.g, changing one knob may nullify the benefits of another) makes the whole task nontrivial if that is not downright impossible. Furthermore, this manual tuning task must be repeated whenever the expert-suggested model or hardware resources changes.

In this thesis we present techniques towards building *self-tuning parameter servers.* By self-tuning, we mean when a long-running ML job is iteratively training the expert-suggested model, the system is also iteratively learning which setting is more efficient for that job and dynamically applies it. Our goal is to free ML users from the system details and progressively unearth and apply better and better system settings for a job as it proceeds. In our field, online self-tuning tools only exist for *physical database design* [12], where various physical structures like indexes are dynamically added or removed when the workload change under *the same system setting* [72]. It has been shown that tuning system configuration requires its own performance model, since the influences of system knobs cannot be captured by the abstract cost-model of the query optimizer [5, 23]. Auto-tuning a database (DB) system configuration is still an active research [5, 9, 11, 22, 23, 38, 41, 42, 57, 70, 71, 76, 82] and so far two of them can (almost) completely keep human out of the loop: iTuned [23] and OtterTune [5]. One remarkable difference between tuning DB systems and tuning ML systems is that the former could hardly afford online tuning while the latter can. Specifically, in the quest of the best configuration, iTuned [23] and OtterTune [5] establish a tuning session to execute the target workload on a sandbox/standby environment using different candidate system settings X and collect their corresponding execution metrics Y (e.g., running time) as the training examples. The tuning sessions cannot be done on a live environment because a bad candidate setting would immediately hurt the throughput and latency. ML programs, in contrast, generally run up to thousands of iterations. Therefore, it is perfectly fine to try some different settings (including some bad ones, as the negative examples) in their early iterations — as long as the completion time of the long-running job can be shorten in toto.

To our knowledge, this is the first work to discuss about live tuning ML jobs towards shorter completion times. The principled contributions of this thesis are as follows:

1. **Online Job Optimization Framework** In online tuning, we always hope to unearth the optimal setting as soon as possible (so as to apply that in the next iteration immediately) while minimizing the number of iterations to unearth it. Therefore, an online job optimization framework should make a decision between trying a potentially better system setting or applying a known good setting at the beginning of each iteration. The framework must **learn**, meaning the quality of its suggestion should improve along the iterations by collecting training examples. Furthermore, the framework must learn **quick** and with **confidence**, so that it can start giving suggestion with confidence once some training examples are collected. The framework shall also be **noise resilience**. Specifically, once the job has started, what we want is no longer as simple as "*what is the estimated completion time if I start this job using setting X?*". Instead, what we want is "*after executing all these n iterations using m different system settings, <u>now</u> what is the <u>latest</u> estimated <u>remaining</u> completion time if I switch to setting X' starting from the next iteration?*". Time estimates have noise and the framework shall be resilient to that. Last, the framework should be **system agnostic**, so it can be easily adopted to different system implementations (e.g., TensorFlow [1], Angel [35,36] , Petuum [79]). Summarizing the above, our first contribution is an online job optimization

framework based on *Bayesian Optimization* (BO) [10,55,69], which is able to address all the above requirements in one go.

2. **Online Progress Estimation** One key input to the online job optimization framework is the *estimated remaining completion time* of an in-flight ML job. That is essentially an *ML job progress estimation* problem. Progress estimation (for long-running analytical queries) has been a challenging problem in DB systems [17, 44, 45, 47, 56] and this thesis would be the first to study this issue on ML systems. The novelty here is **statistical progress estimation** (i.e., how many **more** iterations converge after executing that many). While there are studies about convergence rate (e.g., [49, 64, 66]), they are *theoretical bounds* based on an *offline* setting. Our contribution here is to **turn those theoretical bounds into live estimates** for feeding the online job optimization framework.

3. **Online Reconfiguration** None of the above would be meaningful unless there is a way to reconfigure the system to a new setting online. Online reconfiguration is **not** system agnostic but in this thesis we discuss different approaches to achieve that. The discussions together cover the mainstream ML system implementations that are based on the PS architecture.

4. **Experimentation on TensorFlow** As an initial effort towards the goal of online ML job tuning, we have implemented our techniques on top of TensorFlow [1], an open-source PS-style ML library/system. Experiments show that our techniques in practice can reduce the long-running completion times of TensorFlow jobs by 1.7×-5.1×.

This thesis results from a joint research project with Dr. Tang Bo, Dr. Eric Lo, Mr. Andy Shen, Mr. PengFei Zhang, Mr. XuXuan Zhou. I am the main student of this project. I am supervised by Dr. Tang Bo and Dr. Eric Lo. Mr. Andy Shen, Mr. PengFei Zhang and Mr. XuXuan Zhou helped in preparing and running the experiments.

Next, we put down the preliminary and background for this thesis (Chapter 2), followed by our main contributions (Chapter 3). We give a review of related work afterwards (Chapter 4) and conclude this thesis with a vision that go beyond self-tuning — building *self-improving* ML systems (Chapter 5).

Chapter 2

Background and Preliminary

2.1 Iterative-Convergent ML Algorithms

ML jobs come in many forms, such as logistic regression and deep neural networks. Nonetheless, almost all seek a set of parameters (values) to a global model A that best summarizes or explains the training data $\mathcal{D}$ as measured by an explicit objective function such as likelihood or risk. Such jobs are usually solved by *iterative-convergent* algorithms and can be abstracted as the following *additive* operation:

$$A^i = A^{i-1} + \alpha\Delta(A^{i-1}, D)$$

where, A^i is the state of the model parameters at iteration i and the update function Δ computes the parameter updates based on the data $D \subseteq \mathcal{D}$, which are added to form the new model state of the next iteration based on a learning

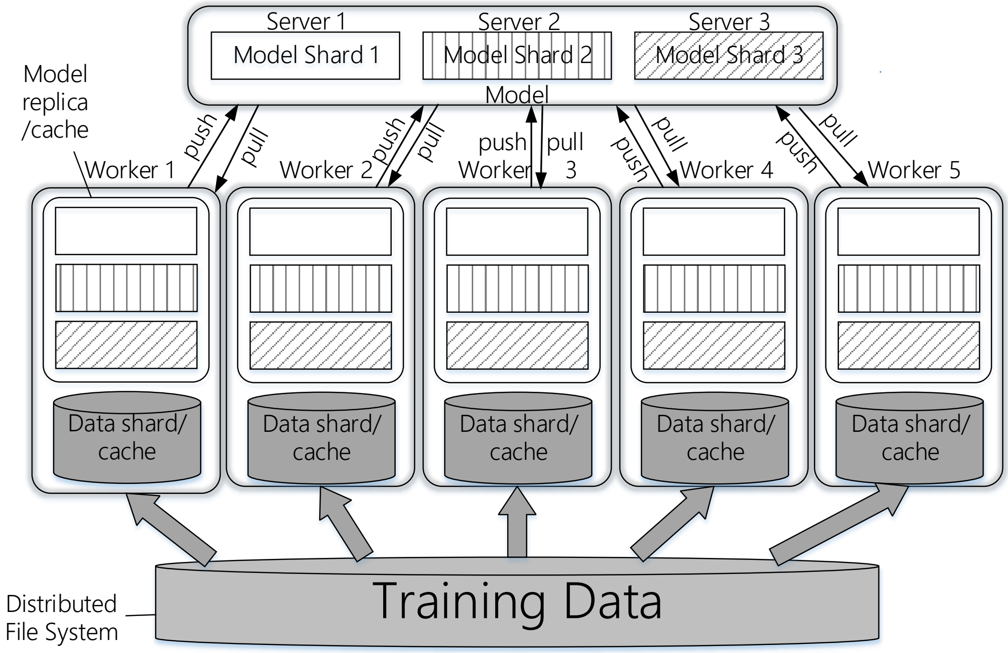


Figure 2.1. Parameter server architecture: full model replication/caching

rate α. This operation iterates itself until A *converges*, i.e., stops changing or the objective function returns a value smaller than a threshold ϵ. Recent works (e.g., [53, 65, 80]) have shown that many industrial-strength tasks require more than 10^6 iterations or months to reach convergence.

Gradient Descent (GD) is arguably the most popular family of iterative-convergent optimization algorithms. GD is applicable to most of the supervised, semi-supervised, and unsupervised ML problems. By its name, GD is a class of first-order methods whose update function Δ is based on computing *gradients* from the data. Batch GD (BGD), Stochastic GD (SGD), Mini Batch GD (MGD), SVRG++ [7], PSGD [86] are some example GD family members. In these algorithms, a pass over the entire dataset is called an *epoch*.

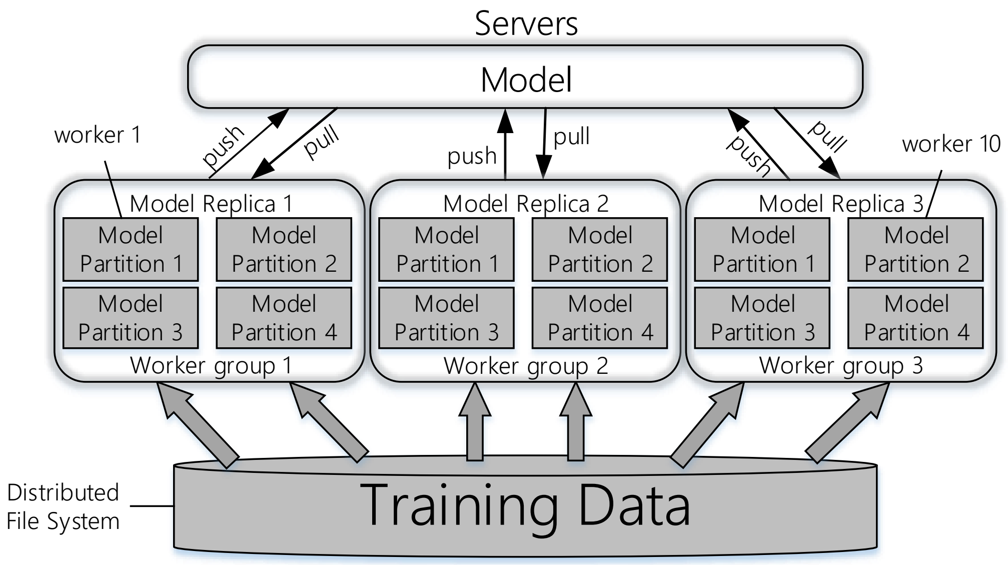


Figure 2.2. Worker group knob

2.2 Parameter Server Architecture

Recently, it is not uncommon to see models with billions of parameters [21, 46] and training using billions of data examples [1]. To support learning at that scale, the *Parameter Server* (PS) architecture [1, 18, 21, 46, 60, 79] has become the de facto standard to distribute the workload across clusters of machines (e.g., Figure 2.1). In it, the model parameters are stored and distributed in "servers". Workers "pull" (part of) the latest model from the server(s), perform local computation like calculating stochastic gradient by accessing their part of training data, and then "push" the updates back to the server(s) whose parameters need to be updated. Servers update the global model by aggregating the local updates from workers (e.g., averaging the stochastic gradients from workers). As opposed to traditional MPI message-passing and MapReduce-like frameworks where pairwise communications between workers are needed in order to exchange each other's parameter updates, the PS architecture has the beauty of only requiring communications between workers and servers, thereby mitigating the network bottleneck. In PS architecture, the server-worker ratio is a key knob.

Another knob in PS-style ML systems is the *consistency protocol*, i.e., how to synchronize the model between servers and workers:

– Bulk Synchronous Parallel (BSP). Systems like GraphX [25] and MLlib [52] enforce a global barrier after each iteration, and thus guarantees that all updates from the previous iteration can be seen by all workers in the current iteration. Under BSP, distributed machine learning follows the same execution logic as sequential algorithm on a single machine, making the proof of convergence straightforward. However, the hardware efficiency of BSP systems are prone to the straggler problem. That is, when some stragglers are significantly slower than other machines due to data skewness, all workers cannot proceed to the next iteration until all stragglers are done.

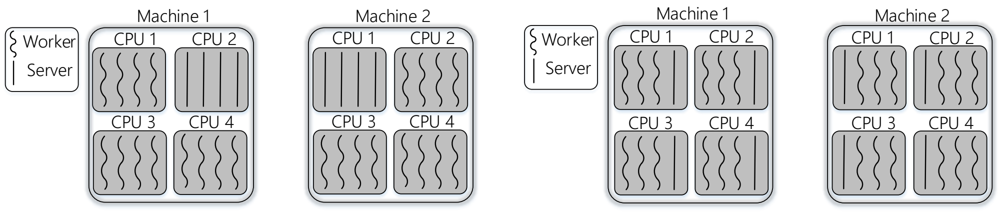
– Asynchronous Parallel (ASP). Systems like DistBelief [21], Hogwild! [64,84] and TensorFlow [1] completely remove the synchronization barrier where workers proceed without waiting for each other, making ASP systems to have a higher hardware efficiency than BSP systems in general. However, the statistical efficiency of ASP systems are prone to the straggler problem – they might lose their convergence guarantee in the presence of stragglers [85]. Under ASP (or SSP below), a worker finishes a push of update to the server is regarded as the end of one iteration.

– Stale Synchronous Parallel (SSP). Some systems (e.g., [35, 74]) exploit the SSP [32] strategy in which a worker is allowed to proceed to the next iteration as long as it does not exceed the slowest one by more than s iterations. SSP systems sit between ASP systems ($s = \infty$) and BSP systems (where $s = 0$). As such, they explicitly expose yet another knob, s, for users to tune.

2.3 Parallelism in Distributed Learning

PS-style ML systems have different levels of parallelisms.

- Data parallelism parallelizes a workload by assigning the data partitions (shards) to different workers. Standard data partitioning knob values include range partitioning and hash partitioning.
- Model parallelism is a unique in ML systems. Under the PS architecture, the servers and workers could have separate model parallelism strategies. On the server side, the model is usually partitioned using standard partitioning schemes (e.g., hash partitioning). On the worker side, the model can be replicated, partitioned, or both. Figure 2.1 shows a case where the full model is replicated on each worker whereas the model is partitioned into three shards on the server side to balance the model push/pull loads from the workers [21]. Figure 2.2 shows another case, where the full model is replicated on each "worker group", and within each group the full model is partitioned across all workers of that group [18, 80]. For that case, the worker group size g is a knob. $g = 1$ means full replication – each group has only 1 worker and each gets a full model replica to work on. $g = n$



(a) server threads on the same CPU (b) server threads on different CPUs

Figure 2.3. Thread affinity on 2 NUMA machines: 4 servers 12 workers

means full partitioning and no replication – all n available workers are in the same group and each gets a partition of the model to work on. For the case in Figure 2.2, $g = 4$.

– Hardware parallelism refers to the type of parallelism defined by the machine architecture and hardware multiplicity. For example, TensorFlow has a "device placement" knob to specify which operation shall be executed on which GPU/CPU. Furthermore, it is important to set the "thread affinity" knob so as to affix the worker/server placement to the hardware threads. Figure 2.3 shows two different thread affinity designs based on the same cluster with the same worker-server ratio. It is easy to imagine that the same ML job executed on the two would have different completion times because of the non-uniform memory access (NUMA) nature of modern servers.

Chapter 3

Towards Self-Tuning Parameter Servers

3.1 Online Job Optimization Framework

Figure 3.1 shows our proposed framework to carry out online ML job optimization. In this thesis, we endeavor to devise off-the-shelf techniques that could be integrated with as many existing ML systems as possible. Therefore, the framework aims to reuse the ML system front-end so that ML users do not need to learn a new API but can enjoy the completion time speedup brought by auto-tuning.

On receiving an ML job J from the front-end, the job will be *instrumented* as J' before sending to the ML system back-end so that it would emit various *per-iteration* metrics (e.g., execution time, loss) to a repository during its execution. Before starting an iteration, the *Tuning Manager* will (1) update a

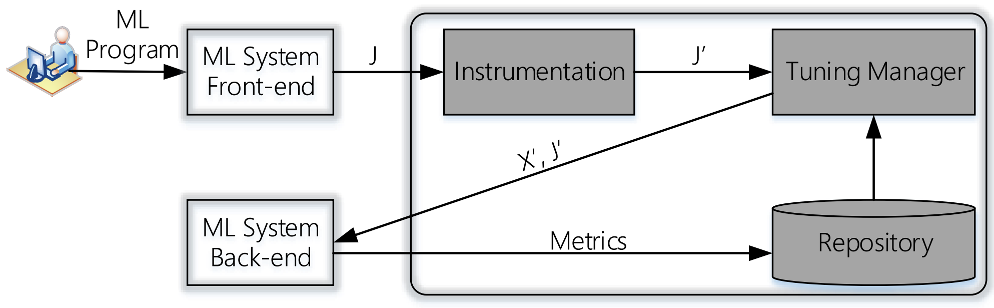


Figure 3.1. Online job optimization framework

statistical model based on the newly collected metrics, (2) recommend a new configuration of system setting X', and (3) reconfigure the system to setting X'. Practically, the Tuning Manager might not return a new recommendation after every iteration but execute a certain number of iterations for each setting in order to understand its statistics efficiency (e.g., the live convergence rate).

3.1.1 Problem Formulation

Given an ML job J, one key goal of the tuning manager is to find the optimal or near-optimal system setting X^* that minimizes the remaining completion time. Let $X = \langle c_1 = v_1, \ldots, c_d = v_d \rangle$ be a system setting, where each c_i is a configurable system parameter with value v_i. We use $T(\langle X, \ell_j \rangle)$ to denote the remaining completion time of the job if we switch to setting X where the model has reached a loss down to ℓ_j. $\langle X, \ell_j \rangle$ is simply a $(d+1)$-dimensional vector that includes both the system setting values and the loss of the model.

Then, the problem is, given a model whose loss is ℓ_j, find the X that minimizes $T(\langle X, \ell_j \rangle)$. Knowing $T(\langle X, \ell_j \rangle)$ ahead would be infeasible. Therefore, we employ Bayesian Optimization (BO) [10, 55, 69] to search for an approximation solution with significantly smaller cost. In what follows, we drop X and $\boldsymbol{\ell}_j$ if the context is clear and represent $T(\langle X, \boldsymbol{\ell}_j \rangle)$ as T.

We summarized all used notations in Table 3.1.

Notation	**Meaning**
X	system setting
X^*	optimal or near-optimal system setting
$\boldsymbol{\ell}_i$	the loss of the first iteration of using setting X_i
$\langle X, \boldsymbol{\ell}_j \rangle$	$(d+1)$-dimensional vector that includes both the system setting values and the loss of the model
$T(\langle X, \boldsymbol{\ell}_j \rangle)$	the remaining completion time of the job if we switch to setting X where the model has reached a loss down to $\boldsymbol{\ell}_j$
t_i^j	the execution time of j-th iteration
l_i^j	the loss of iteration j
Y_i	the estimated remaining completion time at setting X_i
$\langle j, X_i, t_i^j, l_i^j \rangle$	collected execution metrics
$\langle X_i, \boldsymbol{\ell}_i, Y_i \rangle$	training data of Bayesian Optimization (BO)
w^j	model parameters after j-th iteration
w^*	the optimal model parameters

Table 3.1. Notation table

3.1.2 Bayesian Optimization

Bayesian Optimization (BO) is a strategy for optimizing a black-box objective function that is unknown beforehand but observable through conducting experiments, like our T. By modeling T as a Gaussian Process [63], BO can return an estimate of T given any X and $\boldsymbol{\ell}$. The estimate and its confidence interval would progressively improve with the number of observations.

BO has the ability to suggest the next setting using a pre-defined *acquisition function* that also gets updated with more observations. There are many choices of acquisition function such as (i) Probability of Improvement (PI) [69], which picks the next setting that can maximize the probability of improving the current best; (ii) Expected Improvement (EI) [63], which picks the next setting that can maximize the expected improvement over the current best; (iii) Upper Confidence Bound (UCB) [20], which picks the one that has the smallest lower bound in its certainty region. Different acquisition functions have different strategies to balance between *exploring* T (so that it tends to suggest a setting in an unknown region of the response surface) and *exploiting* the knowledge so far (so that it tends to suggest a setting that lies in a known high performance region). In this thesis, we choose EI because it has shown to be more robust than PI, and unlike UCB, it is parameter-free. Using BO with EI has the ability to learn the objective function quickly and always return the expected optimal setting.

BO itself is noise resilient. Specifically, what we can collect from experiments is actually T':

$$T' = T + e \tag{3.1}$$

where e is a Gaussian noise with zero mean, i.e., $e \sim \mathbb{N}(0, \sigma^2)$. Since T', T, and e are Gaussian, we can infer T and its confidence interval [6]. As we will discuss in Section 3.2, the observation noise comes from the fact T' is not a direct measurement but a product between (i) *per-iteration execution time* (hardware efficiency) and (ii) *estimated number of iterations left* (statistical progress). Although (i) could be directly observed, (ii) has to be based on turning a theoretical bound into an estimate based on the current progress (Section 3.2).

There are also some other reasons that we choose BO. First, BO has an advantage of being non-parametric, meaning it does not impose any limit on T, making our techniques useful for a variety of ML systems and platforms (system agnostic). Furthermore, it can deal with non-linear response surface but require far fewer samples (quick learner with confidence) than others with similar power like deep network. Lastly, BO has a good track record on tuning database systems [5, 23].

In the following, we briefly describe how to model T as a Gaussian Process (GP). For details about GP, we refer readers to [63]. In a nutshell, GP models $T(\langle X, \boldsymbol{\ell}_j \rangle)$ by a mean function $m(\cdot)$ and a covariance kernel function $k(\cdot, \cdot)$:

$$\begin{aligned}
m(\langle X_i, \boldsymbol{\ell}_i \rangle) &= \mathbb{E}[T(\langle X_i, \boldsymbol{\ell}_i \rangle)] \\
m(\langle X_j, \boldsymbol{\ell}_j \rangle) &= \mathbb{E}[T(\langle X_j, \boldsymbol{\ell}_j \rangle)] \\
k(\langle X_i, \boldsymbol{\ell}_i \rangle, \langle X_j, \boldsymbol{\ell}_j \rangle) &= \mathbb{E}[(T(\langle X_i, \boldsymbol{\ell}_i \rangle) - m(\langle X_i, \boldsymbol{\ell}_i \rangle)) \\
&\quad (T(\langle X_j, \boldsymbol{\ell}_j \rangle) - m(\langle X_j, \boldsymbol{\ell}_j \rangle))]
\end{aligned}$$

where the kernel function makes sure $T(\langle X_i, \boldsymbol{\ell}_i \rangle)$ and $T(\langle X_j, \boldsymbol{\ell}_j \rangle)$ have large covariance if $\langle X_i, \boldsymbol{\ell}_i \rangle$ and $\langle X_j, \boldsymbol{\ell}_j \rangle$ are similar and have small covariance otherwise. In this work, we adopt Matérn covariance [54] as the kernel function because it does not require strong smoothness.

There is a closed-form for the EI acquisition function from [37]. Let t be the minimum value of T observed from the current best setting so far. For each candidate setting X and a model with loss $\boldsymbol{\ell}$, we can compute its *expected improvement* of using X (over the current best known setting) as:

$$EI(\langle X, \boldsymbol{\ell} \rangle) = \begin{cases} (t - m(\langle X, \boldsymbol{\ell} \rangle))\Phi(Z) + \sigma(\langle X, \boldsymbol{\ell} \rangle)\phi(Z) & \text{if } \sigma(X_i) > 0 \\ 0 & \text{if } \sigma(X_i) = 0 \end{cases}$$

$\sigma(\langle X, \boldsymbol{\ell} \rangle) = \sqrt{k(\langle X, \boldsymbol{\ell} \rangle, \langle X, \boldsymbol{\ell} \rangle)}$, $Z = \frac{t - m(\langle X, \boldsymbol{\ell} \rangle)}{\sigma(\langle X, \boldsymbol{\ell} \rangle)}$, and Φ and ϕ are the CDF and PDF of standard normal, respectively.

3.1.3 Using BO in Online Tuning

We propose to divide the execution of an ML job into two phases: *initialization* and *online tuning.*

Initialization Phase The *initialization* phase refers to the early iterations of the job. Its goal is to quickly bring in a small set of representative settings and their execution metrics to set up the BO. Initially, the job starts the first a iterations using the setting X_0, which is the default or the one given by the user. Iterations after that will select the one with the highest expected improvement EI, from an initial *orthogonal sample* $\mathcal{S}$ [81] of candidate settings from the setting space. Orthogonal sampling is chosen over random sampling because the latter is often ineffective when only a few samples are collected from a fairly high dimensional space. So, assume the initial sample size is m and each setting runs a iterations, the first $a + m \cdot a$ iterations belong to the initialization phase.

Figure 3.2a illustrates the execution metrics that would be inserted into the repository after trying m different settings, with $a = 5$ iterations. Each record in the collected execution metrics is a quadruple $\langle j, X_i, t_i^j, l_i^j \rangle$, with X_i indicates that

$\langle j, X_i, t_i^j, l_i^j \rangle$
$\langle 0, X_0, t_0^0, l_0^0 \rangle$
$\ldots$
$\langle 4, X_0, t_0^4, l_0^4 \rangle$
$\langle 5, X_1, t_1^5, l_1^5 \rangle$
$\ldots$
$\langle 9, X_1, t_1^9, l_1^9 \rangle$
$\ldots$
$\langle n, X_m, t_m^n, l_m^n \rangle$

(a) collected execution metrics

$\langle X_i, \boldsymbol{\ell}_i, Y_i \rangle$
$\langle X_0, \boldsymbol{\ell}_0, Y_0 \rangle$
$\langle X_1, \boldsymbol{\ell}_1, Y_1 \rangle$
$\langle X_2, \boldsymbol{\ell}_2, Y_2 \rangle$
$\ldots$
$\langle X_m, \boldsymbol{\ell}_m, Y_m \rangle$

(b) training data

Figure 3.2. Execution metrics and training data

iteration j was executed using setting X_i, t_i^j indicates the execution time of that iteration, and l_i^j indicates the loss of the model after that iteration. Hereafter, we also refer l_i^j as "the loss of iteration j", which is:

$$l_i^j = f(\mathsf{w}^j) - f(\mathsf{w}^*) \tag{3.2}$$

where f is the learning function specified in the expert-suggested model (e.g., the labeling function, in classification), w^* is the optimal model parameter (while w^* is unknown, $f(\mathsf{w}^*)$ is known from the ground truth, e.g., the labels of the training data), and w^j is the updated model parameter after this iteration j.

The loss of *one* iteration alone is insufficient to judge whether a setting has good statistical efficiency. Furthermore, even a lousy setting could improve the loss a lot in early iterations whereas an optimal setting could hardly improve the loss if it is applied in late iterations. Consequently, the execution metrics would be preprocessed into triples $\langle X_i, \boldsymbol{\ell}_i, Y_i \rangle$, where $\boldsymbol{\ell}_i$ is the loss of the *first* iteration of using setting X_i, i.e., $\boldsymbol{\ell}_i = l_i^{ia}$ (e.g., l_0^0, l_1^5 in Figure 3.2a), and Y_i is the estimated remaining completion time assuming that the remaining iterations will use X_i

and resume from where the model has reached a loss down to $\boldsymbol{\ell}_i$. We set l_i^{ia} as $\boldsymbol{\ell}_i$, but not or any other values like $l_i^{(i+1)a-1}$ (i.e., the loss of the last iteration of using setting X_i; e.g., l_0^4, l_1^9 in Figure 3.2a), because BO is going to predict what setting it shall use if it picks up the job from a model whose loss is l_i^{ia}. Figure 3.2b shows the training data after preprocessing.

The initialization phase takes a total of $n = a + m \cdot a$ iterations and it ends with feeding all training data to the BO (i.e., to learn σ, parameter values of the kernel function, etc.). After that is the *online tuning* phase.

Online Tuning Phase In this phase, a new set of orthogonal sample S will be generated at the beginning of every a iterations. The set of candidate settings at the beginning of those iterations would then be $S \cup \mathbb{X}$, where $\mathbb{X}$ is the set of settings that has been selected by BO and *executed* so far. Then, the next recommended setting X' is the one with the highest EI:

$$X' = \arg\max_{X \in S \cup \mathbb{X}} EI(\langle X, \boldsymbol{\ell}_j \rangle)$$

Note that the sample size S in the online tuning phase can be way larger than the size of the initial sample $\mathcal{S}$ because the EIs in this phase are obtained through calculation but not actual execution. Furthermore, depending on the reconfiguration cost R_{cost} (Section 3.3), a reconfiguration only happens when $EI(\langle X', \boldsymbol{\ell}_j \rangle) - R_{cost} > 0$. In other words, if a reconfiguration costs more than it will potentially save, that reconfiguration would not take place. Overall, the online tuning phase goes on until the job finishes.

Values of a and m We end this section by discussing how to set the values of a

and m. Our major purpose is to avoid our techniques being parametric if possible. The main usage of a is to deduce the statistical progress (convergence rate). So, we set a, the number of iterations executed for each setting, be the number of workers so as to assume that each worker has already pushed the update to the server at least once, by default. We discuss how that default value can be overridden based on the application's semantic in Section 3.3. Nonetheless, we remark that no systems can reach 100% parameter-free. For example, OtterTune, iTuned, and our approach also at least need to specify m, the initial sample size. While the initial sample size used in Ottertune is unclear, iTuned evidences that a very small sample size is enough. In this thesis, we adopt $m = 20$. In Chapter 5, we put down interesting directions about how to possibly eliminate this very last knob by *transfer learning*.

3.2 Online Progress Estimation

One key input to the online job optimization framework is the estimated remaining completion time Y_i, assuming that the j-th iteration has just finished, the current model has a loss of l_j, and the remaining iterations will continue to use setting X_i.

Estimating Y_i has to base on both the collected quadruples $\langle j, X_i, t_i^j, l_i^j \rangle$ and statistical progress estimation techniques. More specifically, Y_i can be estimated as:

$$Y_i = \bar{t_i} \times r_j$$

which is a product between (i) *per-iteration execution time* $\bar{t_i}$ (hardware efficiency) and (ii) *estimated number of iterations left* r_j (statistical progress). $\bar{t_i}$ could be directly computed as the average of the recorded iteration times of using that setting X_i, e.g., $\bar{t_1}$ can be computed as $(t_1^5 + t_1^6 + \cdots + t_1^9)/5$ in Figure 3.2a. In this section, we present two approaches to obtain r_j, the remaining number of iterations required to reach model convergence.

3.2.1 Stateless Progress Estimation

The machine learning community has well studied the convergence properties of various machine learning algorithms [51, 58, 64, 66]. For instance, if the learning function f is strongly convex and L-smooth[1], then the convergence rate of **serial** SGD is $O(\frac{1}{\epsilon})$ [58], where ϵ is the user-specific convergence threshold. More specifically, referring to Eq. 3.2, if $l_i^j \leq \epsilon$, then the model is regarded as converged. In other words, let k be the total number of iterations required to reach a loss ϵ, then $k \geq \frac{\mathcal{H}_1}{\epsilon}$, where $\mathcal{H}_1$ is the hidden constant in $O(\frac{1}{\epsilon})$.

So far, all convergence analysis are based on an *offline* setting: (1) the learning **always** starts from an initial model whose parameter w^0 which are all 0's in theoretical analysis [66]; and (2) the system setting never changes in the course of the job. A recent work [39] has leveraged the above to predict the actual value of k by establishing an offline tuning session like iTuned and OtterTune. First, the losses of some iterations of an ML job are collected, e.g., $\langle 0, l^0 \rangle, \langle 1, l^1 \rangle, \langle 2, l^2 \rangle, \cdots$. Then those $\langle j, l^j \rangle$ pairs are fitted to a function $j = \frac{\mathcal{H}_1}{l^j}$

[1] A convex function f is L-smooth when $||\nabla f(\mathsf{w}^a) - \nabla f(\mathsf{w}^b)|| \leq L||\mathsf{w}^a - \mathsf{w}^b||, \ \forall \ \mathsf{w}^a, \mathsf{w}^b \in \mathbb{R}^n$, where $||\cdot||$ denotes the Euclidean norm and $\nabla f(\mathsf{w})$ denotes the gradient of function $f : \mathbb{R}^d \rightarrow \mathbb{R}$ at w.

to deduce $\mathscr{H}_1$. Finally, k, the number of iterations that a new ML job required to converge to a loss ϵ is derived as $k = \frac{\mathscr{H}_1}{\epsilon}$.

We can extend the approach above to online predict r_j. For example, if we also focus on SGD, then we (i) use the convergence rate of **parallel** SGD instead of that of serial SGD and (ii) use only the losses collected during the iterations related to the current setting of interest, X_i, to do the fitting. Specifically, the convergence rate of parallel SGD can be represented by the function

$$j = \frac{\mathscr{H}_2}{l_i^j} \log \frac{d}{l_i^j} \tag{3.3}$$

where d is any constant greater than Ld_0, with $d_0 = ||\mathsf{w}^0 - \mathsf{w}^*||^2$ and L is the Lipschitz constant [64]. Since w^* is unknown until the training is done, Ld_0 becomes a hidden constant d that has to be greater than l_i^j, or else the log inside the fitting function becomes negative. Recall from Eq. 3.2 that l_i^jcan be arbitrary large. So the off-the-shelf safest way is to set d as the maximum value in domain.

Although correct, we regard this approach only as a baseline because the resulting estimates would not be that reliable — (i) the large d would dominate the fitting function even after the log and (ii) d is static in a way that would not get tighten even with the update of the model state. More specifically, d is static because it is stateless — it is always L times the constant distance between the initial model w^0 and the optimal model w^* (c.f. $d_0 = ||\mathsf{w}^0 - \mathsf{w}^*||^2$). That explains why we name this approach a *stateless* approach. In Section 3.2.2 up next, we present a *stateful* approach that takes the latest model state into account so that the value d keeps tighten as the model approaches convergence, resulting in far more accurate estimations.

As a summary, r_j is estimated by: (i) fitting the function $j = \frac{\mathscr{H}_2}{l_i^j} \log \frac{d}{l_i^j}$ using l_i^j from iterations of using X_i and determine the value of $\mathscr{H}_2$. Referring to Figure 3.2a as an example, after the j=9-th iteration, we can fit $\langle 5, l_1^5 \rangle$ to $\langle 9, l_1^9 \rangle$ to determine the constant $\mathscr{H}_2$ specifically for setting X_1. (ii) Substitute $l_i^j$with the value ϵ to the fitting function and regard the result as k, i.e., $k = \frac{\mathscr{H}_2}{\epsilon} \log \frac{d}{\epsilon}$. (iii) Since k is an estimated value about the **total** number of iterations required but in fact j iterations have been elapsed, we set $r_j = k - j$.

3.2.2 Stateful Progress Estimation

Our goal now is to derive a better value of d for Eq. 3.3 to satisfy two requirements:

– (R1) $\frac{d}{l_i^j} > 1$; otherwise $\log \frac{d}{l_i^j}$ is negative.

– (R2) d is tighter than Ld_0; hopefully, gets tighten with the model updates.

In the following, we show that we can satisfy both (R1) and (R2) by setting $d = \boldsymbol{\ell}_i$, where $\boldsymbol{\ell}_i$ is the observed loss of the first iteration of using setting X_i (c.f. Section 3.1.3):

1. $l_i^j = f(\mathsf{w}^j) - f(\mathsf{w}^*)$ by Eq. 3.2
2. $f(\mathsf{w}^j) - f(\mathsf{w}^*) \leq \frac{L}{2} ||\mathsf{w}^j - \mathsf{w}^*||^2$ by [64], which upper bound the loss.
3. $\mathbb{E}[||\mathsf{w}^j - \mathsf{w}^*||^2] \leq \mathbb{E}[||\mathsf{w}^0 - \mathsf{w}^*||^2]$ by [64], which expects the model parameter w^j of any executed iteration is closer to the optimal w^* than the initial model w^0.

4. Take expectation on both sides of (2), and then combine that with (3), we have:

$$\mathbb{E}[f(\mathsf{w}^j) - f(\mathsf{w}^*)] \leq \frac{L}{2}\mathbb{E}[||\mathsf{w}^j - \mathsf{w}^*||^2] \leq \frac{L}{2}\mathbb{E}[||\mathsf{w}^0 - \mathsf{w}^*||^2] = \frac{L}{2}d_0 \leq Ld_0$$

5. Take expectation on both sides of (1), and then combine that with (4), we have:

$$\mathbb{E}[l_i^j] = \mathbb{E}[f(\mathsf{w}^j) - f(\mathsf{w}^*)] \leq Ld_0, \quad \forall j = ia + z, \quad \forall z \in [1, a)$$

6. Now, consider another view of (2):

$$\mathbb{E}[f(\mathsf{w}^j) - f(\mathsf{w}^*)] \geq \mathbb{E}[f(\mathsf{w}^{j+z}) - f(\mathsf{w}^*)], \quad \forall z > 0$$

then we can regard:

$$\mathbb{E}[\boldsymbol{\ell}_i] \geq \mathbb{E}[l_i^j], \quad \forall j = ia + z, \quad \forall z \in [1, a)$$

which follows [58] to assume the loss of the first iteration of using setting X_i is expected to be larger than the loss of the subsequent iterations and more importantly:

$$\frac{\mathbb{E}[\boldsymbol{\ell}_i]}{\mathbb{E}[l_i^j]} \geq 1$$

So, during online tuning, we can set $d = \boldsymbol{\ell}_i$, it satisfies (R1) by (6) and satisfies (R2) by (5). As a result, the fitting function now becomes:

$$j = \frac{\mathscr{H}_2}{l_i^j} \log \frac{\boldsymbol{\ell}_i}{l_i^j} \tag{3.4}$$

Referring to Figure 3.2a as an example again and assume l_1^5 to l_1^9 are 0.9, 0.8, 0.7, 0.6, 0.5 respectively. Then, after the j=9-th iteration, we fit $\langle 5, 0.9 \rangle$ to $\langle 9, 0.5 \rangle$ to determine the constant $\mathscr{H}_2$ using $\boldsymbol{\ell}_i = 0.9$.

We call this approach a *stateful* approach because the estimation of k now takes into account the model state $\boldsymbol{\ell}_i$ instead of the constant initial state. Figure 3.3a visualizes the importance of turning the fitting function from stateless (using d in the log) to stateful (using $\boldsymbol{\ell}_i$ in the log), where the online prediction of k could yield a significant difference.

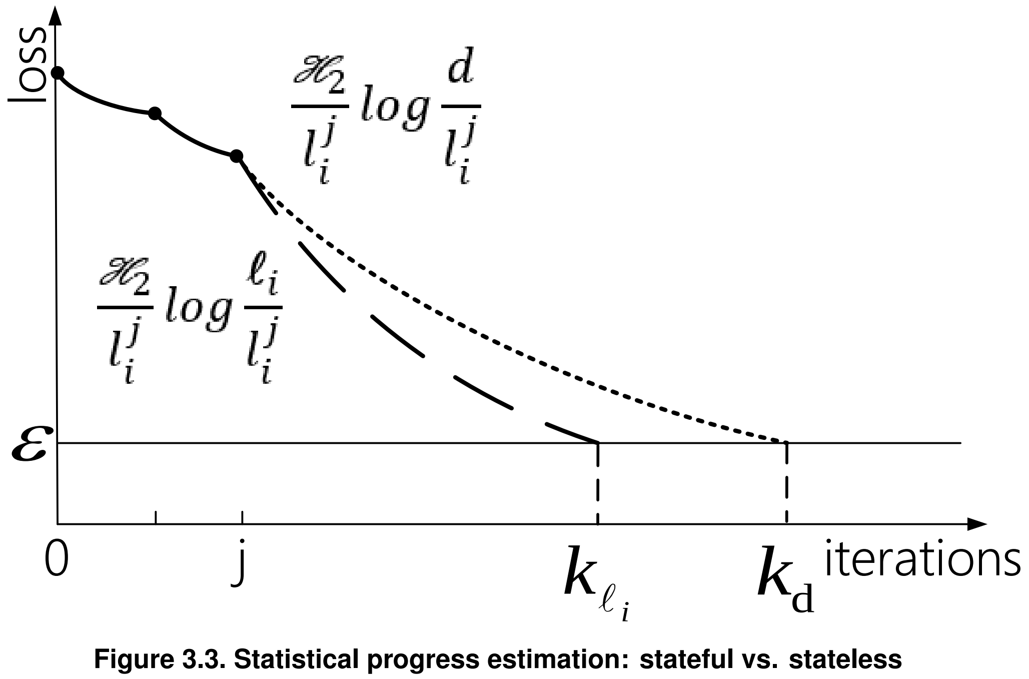

loss
$\frac{\mathscr{H}_2}{l_i^j} log \frac{d}{l_i^j}$
$\frac{\mathscr{H}_2}{l_i^j} log \frac{\ell_i}{l_i^j}$
$\mathcal{E}$
0
j
k_{ℓ_i}
k_d
iterations

Figure 3.3. Statistical progress estimation: stateful vs. stateless

3.3 Online Reconfiguration

Online reconfiguration changes the system setting in the course of an MJ job. Under the PS architecture, the following physical changes could be triggered by a reconfiguration:

- (Type I) Data Relocation: Reconfigurations like turning a worker node to a server node and moving data from CPU to GPU within a worker would involve data relocation. Here, we further bifurcate data relocation into:
  - (Type I-a) Training Data Relocation
  - (Type I-b) Model Data Relocation
- (Type II) System Setting Reconfiguration: For example, in **TensorFlow**, there is a knob to turn on or off the function inlining optimization. This kind of knobs would not trigger any data relocation.

In terms of online reconfiguration facilities, most ML systems already have one that can be re-used by us — the checkpointing and recovery feature (e.g., the save & restore in TensorFlow). In most circumstances, that feature is collectively implemented by four techniques:

1. Checkpointing (CKP): This saves the model state (e.g, the model parameter w^j, the current iteration number j) to a persistent storage. Usually, this would not save any system settings (e.g., whether function inlining is on or off) because those values are stored separately in a system configuration/property file/in-memory data structure. Moreover, checkpointing

does not involve the training data because there is a master copy of the training data in the shared storage (e.g., HDFS).

2. System Setting Recovery (SSR): This is built-in as part of the recovery process, in which the system is reinitialized based on the setting specified in the configuration/property file/data structure.

3. Model Data Recovery (MDR): This is the other part of the build-in recovery process, in which the model state is restored to the servers based on the system setting.

4. Training Data Recovery (TDR): Because the training data is read only and stored in the shared storage. Therefore, on recovery, the workers would simply fetch the missing data from the shared storage directly.

Existing ML systems implement their checkpointing and recovery process as a CKP and a full SSR+MDR+TDR, respectively. For online reconfiguration purpose, we can however implement that more efficiently by mix-and-match those techniques. One such reconfiguration scheme is as follows:

Reconfiguration Scheme 1 :

– For Type II reconfiguration only, change the system configuration file and invoke SSR.

– For Type I-a reconfiguration only, just invoke TDR.

– For Type I-b reconfiguration only, invoke CKP+MDR.

– For any combination of the above, invoke the union of their actions.

The advantages of this scheme is its readiness — almost all open-source ML systems that we have examined, including TensorFlow [1] , Petuum [79], Angel [36], PS-lite [46] can support this online reconfiguration scheme with minimal effort. One issue of this scheme, however, is its save and restore overhead for some reconfiguration types. Nonetheless, that is seldom a big issue in nowadays setting because many industrial-strength ML jobs issue checkpoints regularly anyway. For example, the default checkpoint frequency of TensorFlow is 10 minutes [2]. More frequent checkpoints could be found from use cases like real-time model serving (e.g., [28]) and continuous training (e.g., [8]). In those cases, one can simply set the reconfiguration frequency as the application's checkpointing frequency so that the online reconfiguration overlaps with the inevitable checkpointing cost. That is, they can override the default value of a, the number of iterations between two settings, based on their requirements on the model freshness.

In the following, we present another reconfiguration technique, namely, On-Demand-Model-Replicatoin, specifically design for Type I-b reconfiguration.

On-Demand-Model-Relocation (ODMR) In our experience of applying our techniques on TensorFlow, a lion share of reconfiguration cost attributes to Type I-b, i.e., the cost of relocating some model parameters from one node to another node (e.g., when a recommendation suggests to increase the number of servers). Consequently, we design a technique, namely, On-Demand-Model-Relocation, that can achieve Type I-b model data relocation at almost no cost.

The idea of ODMR is to carry out parameter relocation *reactively*. Con-

[2]`https://www.tensorflow.org/api_docs/python/tf/train/Supervisor`

cretely, on receiving a Type I-b request, the system only invokes SSR to reflect the decision of moving a parameter from a source to a destination. The actual parameter movement takes place only when a parameter is pulled from the source server and pushes back to the destination server. Suppose there are two servers S_1 and S_2 and they originally manage parameters $\{w_1, \ldots, w_6\}$ and $\{w_7, \ldots, w_{12}\}$, respectively. Now, assume a reconfiguration suggests to add one more server S_3 so that the three servers, S_1, S_2, and S_3 manage parameters $\{w_1, \ldots, w_4\}$, $\{w_7, \ldots, w_{10}\}$, and $\{w_5, w_6, w_{11}, w_{12}\}$, respectively. So, when a worker requests a parameter that is supposed to be relocated, e.g., w_{12}, we simply let the worker to pull from the old destination S_2. After the workers have computed the updates, they push both their original values and the updates to the new destination S_3. The reasons of pushing the original value are that (1) the destination S_3 does not have the original value o, so sending the updates u alone is not enough and (2) the original value "flags" the servers that this push is special and to avoid possibly repeated counting — the first time the server receives the message $\langle o, u_1 \rangle$ it should create a new parameter with value $o + u_1$, but the second time it receives a message $\langle o, u_2 \rangle$, it should act like receiving a normal push with u_2.

The ODMR approach has the merit of carrying out Type I-b relocation at zero cost. Unfortunately it is invasive — it requires modifications to the underlying ML systems to support this. Systems that support ODMR however should implement it because of its efficiency. The following reconfiguration scheme summarizes the use of ODMR for different cases:

Reconfiguration Scheme 2 :

– For Type II reconfiguration only, change the system configuration file and invoke SSR.

– For Type I-a reconfiguration only, just invoke TDR.

– For Type I-b reconfiguration only, then invoke ODMR.

– For any combination of the above, invoke the union of their actions.

We did a small survey on the source-code of TensorFlow, Petuum, Angel, and PS-lite to check their "friendliness" with respect to incorporating ODMR/Scheme 2, we found out that:

System	Friendliness with Scheme 2
Angel	Yes, with minimal effort
Petuum	Yes, with some effort
PS-lite	Yes, with some effort
TensorFlow	No, almost needs to turn on its head

Angel is most ready to use Scheme 2 off-the-shelf because it not only has an explicit push/pull API, it also exposes user-defined functions called `psFunc` for developers to customize the push and pull functions. With that, one can implement Scheme 2 on Angel non-invasively. Petuum and PS-lite can also use Scheme 2 because they also have explicit push/pull API. However they do not expose facilities like `psFunc` as in Angel. Implementing Scheme 2 on TensorFlow would turn on its head because TensorFlow was deliberately architected to make the push and pull API implicit and many system settings are immutable except through the system save-and-restore facility [1]. Nonetheless, TensorFlow has the largest community among the four ML systems that we have studied. Therefore, as the initial effort towards the vision of self-tuning ML systems, we have decided to build our first prototype on TensorFlow. Implementing our techniques on others is our future work.

3.4 Prototype Implementation

We implemented our techniques on top of TensorFlow v1.3 (TF). The implementation includes a user-level library written in Python 2.7 in order to abstract out the system setting of a TensorFlow program. We implemented the Tuning Manager and the repository using Python.

In details, TensorFlow has a front-end written in Python and a back-end written in C++.

Currently TensorFlow exposes all system settings through the class constructors and class attributes of the core classes. Table 3.2 (left) shows how ML users specify those settings within the program. We have implemented a Python module SelfTF so that users no longer need to specify the system settings anymore. Table 3.2 (right) shows the corresponding TensorFlow with SelfTF installed.

We modified TensorFlow's back-end (in C++) so that it can support reconfiguration scheme 1. We refer this prototype implementation as SelfTF in this section. Currently, SelfTF can only support SGD.

We end this section by recalling the need to estimate the reconfiguration cost R_{cost} for the online tuning phase (Section 3.1). With the discussion above, it becomes clear that R_{cost} depends on the scheme and the type of reconfiguration technique. Nonetheless, empirically we observe that the cost variance of

TensorFlow	SelfTF

# Manually define the configuration of the TensorFlow training server 	
cluster = tf.train.ClusterSpec({"ps": parameter_servers_list, "worker": workers_list})
 	
server_config = tf.ConfigProto(
  inter_op_parallelism_threads=8,
  intra_op_parallelism_threads=8,
  ....
)
 	
server = tf.train.Server(
  cluster,
  job_name=FLAGS.job_name,
  task_index=FLAGS.task_index,
  config=server_config)
	# Configuration is managed by SelfTF
	
	server =
	  SelfTF.create_training_server())

# Graph operation device placement by TensorFlow 	
with tf.device(tf.train.replica_device_setter(
    worker_device="/job:worker/task:%d" % FLAGS.task_index,
    cluster=cluster)):
	# Graph operation device placement by SelfTF
	with tf.device(
	  SelfTF.device_placement()):

# TensorFlow graph definition *# Define a variable* tf.get_variable(partitioner= tf.fixed_size_partitioner(10)) *# Other operations* tf.add(...) *# End of graph definition*	*# TensorFlow graph definition* *# Define a variable* tf.get_variable(partitioner= **SelfTF.get_variable_partitioner()**) *# Other operations* tf.add(...) *# End of graph definition*

Table 3.2. Example of SelfTF **module**

each technique (e.g., MDR) is fairly stable and thus we can simply deduce the individual costs from the execution metrics collected during the initialization phase.

3.5 Experimental Evaluation

3.5.1 Experiment Setup

Machine Learning Models and Datasets We evaluate SelfTF with three widely used machine learning models: (i) l_2 regularized Logistic Regression (LogR), (ii) Support Vector Machine (SVM), and (iii) Convolutional Neural Network (CNN). For CNN, we used a convolutional neural network with five layers. The first convolutional layer filters the 24×24 input image with 64 kernels of size 5×5. The second convolutional layer takes as input the (response-normalized and pooled) output of the first convolutional layer and filters it with 64 kernels of size 5×5. The third and the fourth layer are fully-connected layers with 394 and 192 neurons, respectively. The last layer is a 10-way softmax output layer. This CNN model has also been used in [18, 43].

We used CIFAR-10 as [40] the training data for CNN and a malicious URL dataset [48] as the training data for both LogR and SVM models. Table 3.3 shows the characteristics of the datasets we used.

Dataset	ML model(s)	# of records	# of features	Size
CIFAR-10	CNN	60,000	1,024	160M
URL	LogR, SVM	2,396,130	3,231,961	4G

Table 3.3. Training datasets

Metrics Following [35], we used the variance of the objective values of the last five iterations to determine whether SGD steadily converges. The convergence thresholds ϵ for LogR, SVM, and CNN are 0.07, 0.07, and 0.5, respectively.

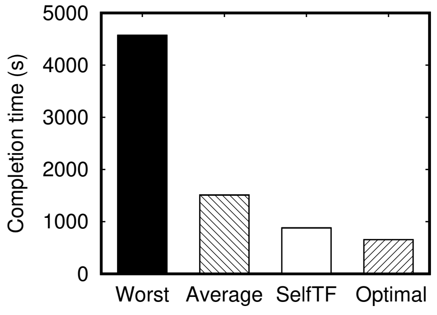
We report the completion time as the wall-clock time of a machine learning job. The completion time includes the time used for data loading and result outputting. Furthermore, we decompose the completion time as the statistical efficiency and hardware efficiency.

Computing Cluster We performed all the experiments on a cluster of 36 identical commodity servers, connected by Ethernet. The network bandwidth is 1Gbps. The computing nodes run a 64-bit Centos 7.3, with the training datasets on HDFS 2.6.0. Each node is Intel Xeon E5-2620 system with 8 cores CPU running at 2.1 GHz, 64GB of memory, and 800GB SSD.

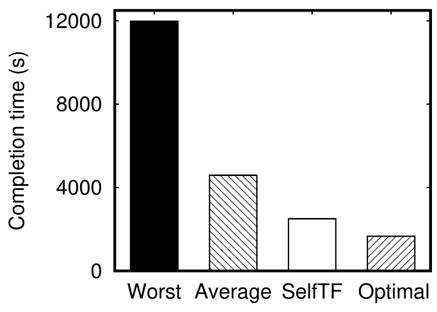
Comparison In the experiments, we implemented a brute-force solution that exhausts the setting space for 60 days and report:

1. Worst: the worst completion time among all examined settings
2. Average: the average completion time among all examined settings.
3. Optimal: the best completion time among all examined settings.

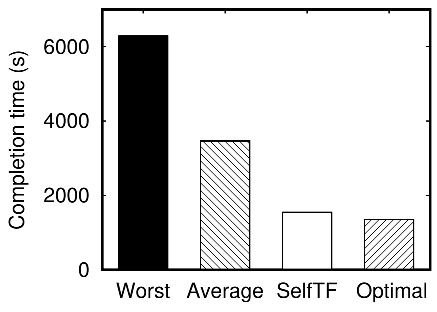
We especially remark the brute-force solution is only for comparison but impractical. For example, Optimal is impractical because nobody would run the same job multiple times to identify the best system settings. We also remark that



(a) CNN



(b) LogR



(c) SVM

Figure 3.4. End-to-end completion time comparison

TensorFlow, although popular, is more like a software library than a system — currently users need to specify most system parameters (e.g., the worker-server ratio) and there are **no** default values for those. Therefore, while Worst and Optimal are unlikely to happen, Average could more or less reflect the real cases.

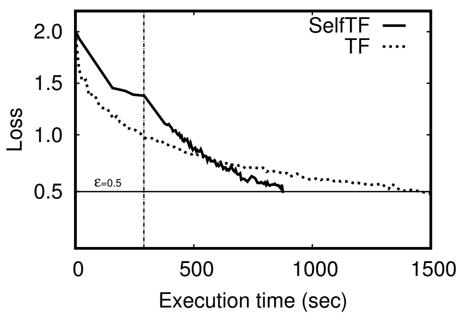
3.5.2 Overall Performance Evaluation

Figure 3.4 compare the completion time of SelfTF on CNN, LogR, and SVM. The major observations are:

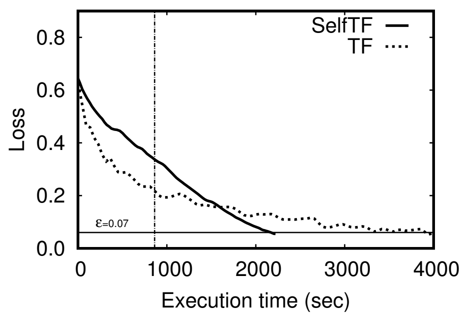
1. SelfTF has a performance on a par with Optimal, meaning SelfTF has the ability to unearth a near-optimal setting during its on-job-training.
2. SelfTF has about 1.7× to 2.2× speedup when compared with Average, meaning SelfTF saves ML users much time on average.
3. SelfTF is about 4.1× to 5.1× faster than Worst, which shows that an ML user should never try her luck when determining the setting but adopt SelfTF whenever possible.

3.5.3 Convergence Analysis

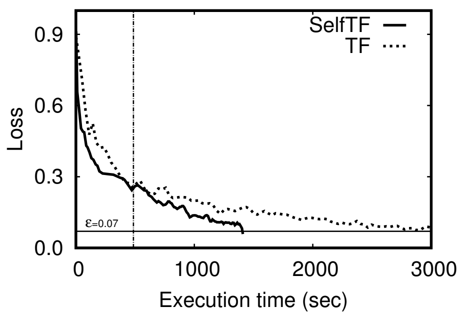
We next study how SelfTF internally behaves within an ML job. Figure 3.5 show that the convergence rate of SelfTF with respect to the job training time. In the figure, we include the convergence rate of one random setting in Average for comparison and we label that as TF. We also indicate the point where SelfTF switches from its initialization phase to the online tuning phase (with a vertical dotted line).



(a) CNN



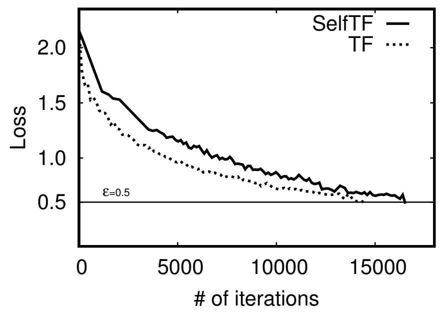
(b) LogR



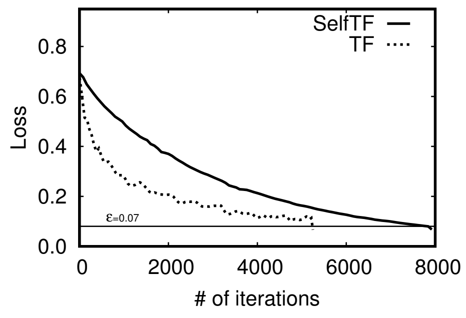
(c) SVM

Figure 3.5. Model convergence rate vs. job training time

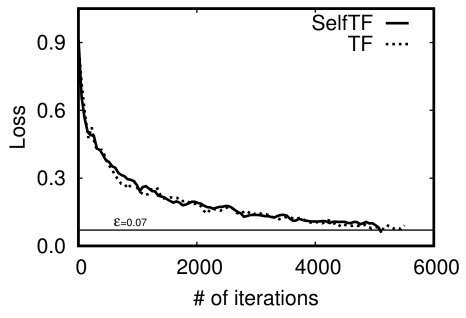
From the figure, we observe that SelfTF might have a slower convergence rate during the initialization phase (for CNN and LogR) because it was trying different settings and some of those might include bad ones. Nonetheless, we know that is worth because that lets SelfTF unearth a near-optimal setting and save significant running time afterwards. The case in CNN is more apparent — the convergence rate improves significantly once online tuning phase has started.



(a) CNN



(b) LogR



(c) SVM

Figure 3.6. Model convergence rate vs. statistical efficiency

3.5.4 Statistical Efficiency versus Hardware Efficiency

We have been emphasizing that the completion time of an MJ job is a complex interplay between statistical efficiency and hardware efficiency and a setting good at one might be a bad setting overall. Figure 3.6 evidences that. In particular in CNN and LogR, SelfTF has chosen settings that need slightly more iterations to convergence but the table below shows that those settings actually have better hardware efficiency. Of course, we iterate that is actually a correct decision because SelfTF outperforms TF in end-to-end completion time.

	SelfTF		TF	
	# of iterations	time per iteration	# of iterations	time per iteration
CNN	16,590	0.05	14,504	0.11
LogR	7,924	0.32	5,360	0.83
SVM	5,063	0.30	5,802	0.59

3.5.5 Reconfiguration Overhead

The table below shows the percentage of time a SelfTF job spent on reconfiguration. It shows that our reconfiguration scheme for TensorFlow has not incurred any significant overhead to the jobs, not to mention that those reconfiguration overheads indeed cover some TensorFlow's checkpointing cost.

Job	% of time on reconfiguration
CNN	12.7%
LogR	4.3%
SVM	9.1%

3.5.6 Stateful vs. Stateless Progress Estimation

Lastly, we compare the power between the stateless approach and the stateful approach in estimating the statistical progress by showing the completion time differences between them. We use completion time instead of accuracy/error in this experiment because the latter would need running SelfTF on **each** setting **completely till convergence** for **each reconfiguration point** — that simply does not worth the time to collect those numbers if reporting the completion time between the two approaches can tell us the efficiency differences. Figure 3.7 shows the experimental results. It clearly shows that a stateful approach is always better than a stateless approach by the completion time difference.

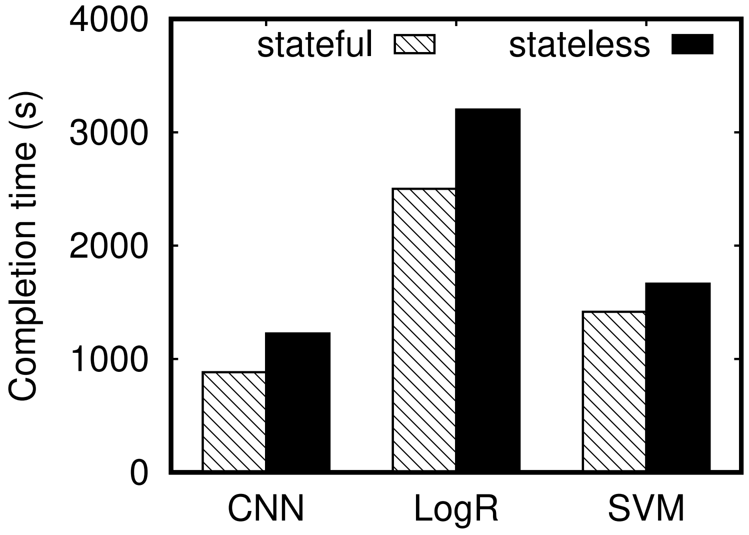


Figure 3.7. Estimation techniques among different models

Chapter 4

Related Work

Self-tuning database systems has been an active research topic for almost two decades [15, 16, 34, 77]. Nonetheless, most of them focused on choosing the best logical or physical design (e.g., index) of a database (e.g., [3, 4, 26]). Tuning system configuration is a different challenge because the influences of system knobs cannot be captured by the internal cost-model of the query optimizer [5, 23]. Existing works mostly adopt a feedback-driven approach to adjust the knobs [5, 9, 11, 22, 23, 38, 41, 42, 57, 70, 71, 76, 82]. So far, however, only iTuned [23] and OtterTune [5] can (almost) completely keep the human out of the loop (e.g., don't need to specify which knob to be tuned). The idea of self-tuning actually goes beyond database systems. There are projects about tuning MapReduce system configurations [29, 31] and selecting the best (e.g., cheapest) cloud containers for a specific workload [6, 30, 33]. To our best knowledge, none of them focus on online tuning an in-flight ML job. Furthermore, most existing works dedicate a specific tuning session to collect execution metrics and train the performance model; the query results obtained in the tuning session are however disposed. In this work, the execution metrics are collected while the ML job is making real progress – there is no waste of resources.

The (short) history of PS architecture began with systems that were specifically designed for LDA topic modeling [68] and deep network [18,21]. Afterwards, general-purpose ML systems also adopt the PS architecture [46, 79]. Compared with auto-tuning DB systems, auto-tuning ML systems is in infancy. In [80], an offline tuner specifically designed for Adam [18], a close-source ML system, was presented. The work manually established an analytical cost-model based on Adam's architecture and design. Latest works [39] and [59] discusses the automatic selection of different GD algorithms by manually creating an analytical cost-model and the automatic placement of operators on CPU/GPU using reinforcement learning, respectively. Our scope is way broader than only those. More importantly, we target online tuning, i.e., a job is executed using better and better system settings as it proceeds. In contrast, [39] targets offline tuning — first decide on which GD algorithm to use and never change that even though a job may last for hours or weeks. In [62], experiments based on a BSP PS system show that changing the cluster resources online could influence the completion time of ML jobs, which supports the arguments of this thesis. In machine learning, auto hyper-parameter tuning (e.g., tuning the number of the hidden layers in a deep neural network) that finds the best model is a grand challenge [27]. Currently, most ML users follow a trial-and-error process to find their "right" model: (i) pick an initial setting for the hyper-parameters, and then train the model for a fixed amount of time (e.g., days); (ii) if the final accuracy is not desirable, then choose another set of hyper-parameter values and repeat the process, until the model has reached the user's expectation. Under this trial-and-error cycle, what we propose in this thesis would significantly reduce the time of each trial, thereby expediting the ideal model seeking process.

Performance modeling and progress estimation are interesting problems in their own right. For example, Ernest [73] trains a performance model for machine learning applications. However, even that latest work has only put the estimation of statistical efficiency as a future work. Progress indicator is an enlightening feature in analytical systems because that lets users know when will they obtain the results [17, 44, 45, 47, 56]. In this thesis, we have pioneered the first progress indicator for ML systems through giving initial solutions to the statistical progress estimation problem.

Our work bear a resemblance to *query reoptimization* for long-running queries [50,78]. Nonetheless, the contexts are entirely different because query reoptimization focuses on switching execution strategies of individual operators under the *same system setting* while we focus on switching to another system setting in the midst of an iterative job. There has been some discussions about live reconfiguration in cloud databases (e.g., [24]) and in-memory transactional systems (e.g., [75]). Our On-Demand-Model-Relocation (ODMR) technique is inspired by the on-demand data migration method of the former but we customize that for model data relocation under the PS architecture. There are also works to reduce the costs of checkpointing and system suspend-and-resume (e.g., [13,14]). Those works are orthogonal to us and any advance in that area could inspire improvement on our reconfiguration techniques.

Chapter 5

Conclusion and Future Work

In this thesis, we make a case for building an online tuner for ML systems. We show that the performances of machine learning (ML) systems, like database (DB) systems, are also subjected to the values of many system parameters. However, unlike DB systems, ML systems can afford online on-job training and tuning because of the long-running nature of ML. To this end, we propose an online job optimization framework that is suitable to all ML systems. We also develop initial solutions to solve the online statistical progress estimation problem. We discuss an array of existing and new techniques to support online reconfiguration on existing ML systems. As an initial effort to showcase our techniques, we have implemented a prototype on top of TensorFlow. Experiments show that various ML tasks gain speedup by a factor of 1.7× to 5.1×.

The area of self-tuning ML systems is still in infancy. Our next step is to extend our idea to other ML systems (e.g., Angel [36]), optimization algorithms (e.g., SVRG++ [7]), and on platforms with heterogenous machines (e.g., [35]).

We will also study the use of *transfer learning* [61] to eliminate the initialization phase. Specifically, when the framework receives a new ML job J, it shall search the repository and locate a previous job $\hat{J}$ that is most similar to J. Then it shall transfer all candidate settings $\mathcal{X}_{\hat{J}}$ that $\hat{J}$ had ever picked to be J's candidate settings. OtterTune [5] has also leveraged a similar idea when facing new DB workloads. By using this transfer learning idea, we might eliminate the whole initialization phase and proceed to online tuning phase directly. We expect the effectiveness of this idea would increase with the number of jobs optimized by the framework, thereby achieving a vision of *self-improving* ML systems.

Bibliography

[1] 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 Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *OSDI*, pages 265–283, 2016.

[2] Gediminas Adomavicius and Alexander Tuzhilin. Toward the next generation of recommender systems: A survey of the state-of-the-art and possible extensions. *TKDE*, 17(6):734–749, 2005.

[3] Sanjay Agrawal, Surajit Chaudhuri, and Vivek R. Narasayya. Automated selection of materialized views and indexes in SQL databases. In *VLDB*, pages 496–505, 2000.

[4] Sanjay Agrawal, Vivek R. Narasayya, and Beverly Yang. Integrating vertical and horizontal partitioning into automated physical database design. In *SIGMOD*, pages 359–370, 2004.

[5] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. Automatic database management system tuning through large-scale machine learning. In *SIGMOD*, pages 1009–1024, 2017.

[6] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics. In *NSDI*, pages 469–482, 2017.

[7] Zeyuan Allen Zhu and Yang Yuan. Improved SVRG for non-strongly-convex or sum-of-non-convex objectives. In *ICML*, pages 1080–1089, 2016.

[8] Denis Baylor, Eric Breck, Heng-Tze Cheng, Noah Fiedel, Chuan Yu Foo, Zakaria Haque, Salem Haykal, Mustafa Ispir, Vihan Jain, Levent Koc, et al. Tfx: A tensorflow-based production-scale machine learning platform. In *SIGKDD*, pages 1387–1395, 2017.

[9] Peter Belknap, Benoît Dageville, Karl Dias, and Khaled Yagoub. Self-tuning for SQL performance in oracle database 11g. In *ICDE*, pages 1694–1700, 2009.

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

[11] Kurt P. Brown, Michael J. Carey, and Miron Livny. Goal-oriented buffer management revisited. In *SIGMOD*, pages 353–364, 1996.

[12] Nicolas Bruno and Surajit Chaudhuri. An online approach to physical design tuning. In *ICDE*, pages 826–835, 2007.

[13] Tuan Cao, Marcos Vaz Salles, Benjamin Sowell, Yao Yue, Alan Demers, Johannes Gehrke, and Walker White. Fast checkpoint recovery algorithms for frequently consistent applications. In *SIGMOD*, pages 265–276, 2011.

[14] Badrish Chandramouli, Christopher N Bond, Shivnath Babu, and Jun Yang. Query suspend and resume. In *SIGMOD*, pages 557–568, 2007.

[15] Surajit Chaudhuri and Vivek R. Narasayya. An efficient cost-driven index selection tool for microsoft SQL server. In *VLDB*, pages 146–155, 1997.

[16] Surajit Chaudhuri and Vivek R. Narasayya. Self-tuning database systems: A decade of progress. In *VLDB*, pages 3–14, 2007.

[17] Surajit Chaudhuri, Vivek R. Narasayya, and Ravishankar Ramamurthy. Estimating progress of long running SQL queries. In *SIGMOD*, pages 803–814, 2004.

[18] Trishul M. Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. Project adam: Building an efficient and scalable deep learning training system. In *OSDI*, pages 571–582, 2014.

[19] Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel P. Kuksa. Natural language processing (almost) from scratch. *Journal of Machine Learning Research*, 12, 2011.

[20] Emile Contal, David Buffoni, Alexandre Robicquet, and Nicolas Vayatis. Parallel gaussian process optimization with upper confidence bound and pure exploration. In *ECML PKDD*, pages 225–240, 2013.

[21] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc'Aurelio Ranzato, Andrew W. Senior,

Paul A. Tucker, Ke Yang, and Andrew Y. Ng. Large scale distributed deep networks. In *NIPS*, pages 1232–1240, 2012.

[22] Karl Dias, Mark Ramacher, Uri Shaft, Venkateshwaran Venkataramani, and Graham Wood. Automatic performance diagnosis and tuning in oracle. In *CIDR*, pages 84–94, 2005.

[23] Songyun Duan, Vamsidhar Thummala, and Shivnath Babu. Tuning database configuration parameters with ituned. *PVLDB*, 2(1):1246–1257, 2009.

[24] Aaron J. Elmore, Vaibhav Arora, Rebecca Taft, Andrew Pavlo, Divyakant Agrawal, and Amr El Abbadi. Squall: Fine-grained live reconfiguration for partitioned main memory databases. In *SIGMOD*, pages 299–313, 2015.

[25] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. Graphx: Graph processing in a distributed dataflow framework. In *OSDI*, pages 599–613, 2014.

[26] Himanshu Gupta, Venky Harinarayan, Anand Rajaraman, and Jeffrey D. Ullman. Index selection for OLAP. In *ICDE*, pages 208–219, 1997.

[27] Isabelle Guyon, Imad Chaabane, Hugo Jair Escalante, Sergio Escalera, Damir Jajetic, James Robert Lloyd, Núria Macià, Bisakha Ray, Lukasz Romaszko, Michèle Sebag, Alexander R. Statnikov, Sébastien Treguer, and Evelyne Viegas. A brief Review of the ChaLearn AutoML Challenge: Anytime Any-dataset Learning without Human Intervention. In *Workshop on Automatic Machine Learning*, pages 21–30, 2016.

[28] Xinran He, Junfeng Pan, Ou Jin, Tianbing Xu, Bo Liu, Tao Xu, Yanxin Shi, Antoine Atallah, Ralf Herbrich, Stuart Bowers, et al. Practical lessons from predicting clicks on ads at facebook. In *Proceedings of the Eighth International Workshop on Data Mining for Online Advertising*, pages 1–9. ACM, 2014.

[29] Herodotos Herodotou, Fei Dong, and Shivnath Babu. Mapreduce programming and cost-based optimization? crossing this chasm with starfish. *PVLDB*, 4(12):1446–1449, 2011.

[30] Herodotos Herodotou, Fei Dong, and Shivnath Babu. No one (cluster) size fits all: Automatic cluster sizing for data-intensive analytics. SoCC, page 18, 2011.

[31] Herodotos Herodotou, Harold Lim, Gang Luo, Nedyalko Borisov, Liang Dong, Fatma Bilgen Cetin, and Shivnath Babu. Starfish: A self-tuning system for big data analytics. In *CIDR*, pages 261–272, 2011.

[32] Qirong Ho, James Cipar, Henggang Cui, Seunghak Lee, Jin Kyu Kim, Phillip B. Gibbons, Garth A. Gibson, Gregory R. Ganger, and Eric P. Xing. More effective distributed ML via a stale synchronous parallel parameter server. In *NIPS*, pages 1223–1231, 2013.

[33] Botong Huang, Shivnath Babu, and Jun Yang. Cumulon: optimizing statistical data analysis in the cloud. In *SIGMOD*, pages 1–12, 2013.

[34] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. Database cracking. In *CIDR*, pages 68–78, 2007.

[35] Jiawei Jiang, Bin Cui, Ce Zhang, and Lele Yu. Heterogeneity-aware distributed parameter servers. In *SIGMOD*, pages 463–478, 2017.

[36] Jie Jiang, Lele Yu, Jiawei Jiang, Yuhong Liu, and Bin Cui. Angel: a new large-scale machine learning system. *National Science Review*, 2017.

[37] Donald R. Jones, Matthias Schonlau, and William J. Welch. Efficient global optimization of expensive black-box functions. *Journal of Global Optimization*, 13(4):455–492, Dec 1998.

[38] B. Dageville K. Dias S. Joshi K. Yagoub, P. Belknap and H. Yu. Oracle‘s sql performance analyzer. *IEEE Data Engineering Bulletin*, 2008.

[39] Zoi Kaoudi, Jorge-Arnulfo Quiane-Ruiz, Saravanan Thirumuruganathan, Sanjay Chawla, and Divy Agrawal. A cost-based optimizer for gradient descent optimization. In *SIGMOD*, pages 977–992, 2017.

[40] Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical report, 2009.

[41] S. Kumar. Oracle database 10g: The self-managing database. *White Paper*, Feb, 2003.

[42] E. Kwan, S. Lightstone, A. Storm, and L. Wu. Automatic configuration for ibm db2 universal database. *Technical report, IBM*, Jan, 2002.

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

[44] Jiexing Li, Arnd Christian König, Vivek R. Narasayya, and Surajit Chaudhuri. Robust estimation of resource consumption for SQL queries using statistical techniques. *PVLDB*, 5(11):1555–1566, 2012.

[45] Jiexing Li, Rimma V. Nehme, and Jeffrey F. Naughton. Toward progress indicators on steroids for big data systems. In *CIDR*, 2013.

[46] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *OSDI*, pages 583–598, 2014.

[47] Gang Luo, Jeffrey F Naughton, Curt J Ellmann, and Michael W Watzke. Toward a progress indicator for database queries. In *SIGMOD*, pages 791–802, 2004.

[48] Justin Ma, Lawrence K Saul, Stefan Savage, and Geoffrey M Voelker. Identifying suspicious urls: an application of large-scale online learning. In *ICML*, pages 681–688, 2009.

[49] Horia Mania, Xinghao Pan, Dimitris Papailiopoulos, Benjamin Recht, Kannan Ramchandran, and Michael I Jordan. Perturbed iterate analysis for asynchronous stochastic optimization. *arXiv preprint arXiv:1507.06970*, 2015.

[50] Volker Markl, Vijayshankar Raman, David E. Simmen, Guy M. Lohman, and Hamid Pirahesh. Robust query processing through progressive optimization. In *SIGMOD*, pages 659–670, 2004.

[51] Qi Meng, Wei Chen, Yue Wang, Zhi-Ming Ma, and Tie-Yan Liu. Convergence analysis of distributed stochastic gradient descent with shuffling. *NIPS*, 2017.

[52] Xiangrui Meng, Joseph K. Bradley, Burak Yavuz, Evan R. Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, D. B. Tsai, Manish Amde, Sean Owen, Doris Xin, Reynold Xin, Michael J. Franklin, Reza Zadeh, Matei Zaharia, and Ameet Talwalkar. Mllib: Machine learning in apache spark. *Journal of Machine Learning Research*, 17:34:1–34:7, 2016.

[53] Ofer Meshi, Mehrdad Mahdavi, and Alex Schwing. Smooth and strong: Map inference with linear convergence. In *NIPS*, pages 298–306. 2015.

[54] Budiman Minasny and Alex. B. McBratney. The matrn function as a general model for soil variograms. *Geoderma*, 128(3):192 – 207, 2005.

[55] Jonas Mockus. *Bayesian approach to global optimization: theory and applications.* Springer Science & Business Media, 2012.

[56] Kristi Morton, Abram L. Friesen, Magdalena Balazinska, and Dan Grossman. Estimating the progress of mapreduce pipelines. In *ICDE*, pages 681–684, 2010.

[57] Dushyanth Narayanan, Eno Thereska, and Anastassia Ailamaki. Continuous resource monitoring for self-predicting DBMS. In *13th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 239–248, 2005.

[58] Arkadi Nemirovski, Anatoli Juditsky, Guanghui Lan, and Alexander Shapiro. Robust stochastic approximation approach to stochastic programming. *SIAM Journal on optimization*, 19(4):1574–1609, 2009.

[59] Khanh Nguyen, Hal Daumé III, and Jordan Boyd-Graber. Reinforcement learning for bandit neural machine translation with simulated human feedback. *arXiv preprint arXiv:1707.07402*, 2017.

[60] Beng Chin Ooi, Kian-Lee Tan, Sheng Wang, Wei Wang, Qingchao Cai, Gang Chen, Jinyang Gao, Zhaojing Luo, Anthony KH Tung, Yuan Wang, and Zhongle Xie. Singa: A distributed deep learning platform. In *ACM Multimedia*, pages 685–688, 2015.

[61] Sinno Jialin Pan and Qiang Yang. A survey on transfer learning. *TKDE*, 22(10):1345–1359, October 2010.

[62] Xinghao Pan, Shivaram Venkataraman, Zizheng Tai, and Joseph Gonzalez. Hemingway: Modeling distributed optimization algorithms. *CoRR*, 2017.

[63] Carl Edward Rasmussen and Christopher K. I. Williams. *Gaussian Processes for Machine Learning (Adaptive Computation and Machine Learning)*. The MIT Press, 2005.

[64] Benjamin Recht, Christopher Re, Stephen J. Wright, and Feng Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *NIPS*, pages 693–701, 2011.

[65] Tom Schaul, Dan Horgan, Karol Gregor, and David Silver. Universal value function approximators. In *ICML*, pages 1312–1320, 2015.

[66] Shai Shalev-Shwartz and Shai Ben-David. *Understanding machine learning: From theory to algorithms*. Cambridge university press, 2014.

[67] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Vedavyas Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy P. Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[68] Alexander J. Smola and Shravan M. Narayanamurthy. An architecture for parallel topic models. *PVLDB*, 3(1):703–710, 2010.

[69] Jasper Snoek, Hugo Larochelle, and Ryan P. Adams. Practical bayesian optimization of machine learning algorithms. In *NIPS*, pages 2960–2968, 2012.

[70] Adam J. Storm, Christian Garcia-Arellano, Sam Lightstone, Yixin Diao, and Maheswaran Surendra. Adaptive self-tuning memory in DB2. In *VLDB*, pages 1081–1092, 2006.

[71] Wenhu Tian, Patrick Martin, and Wendy Powley. Techniques for automatically sizing multiple buffer pools in DB2. In *Proceedings of the 2003 conference of the Centre for Advanced Studies on Collaborative Research*, pages 294–302, 2003.

[72] Gary Valentin, Michael Zuliani, Daniel C. Zilio, Guy M. Lohman, and Alan Skelley. DB2 advisor: An optimizer smart enough to recommend its own indexes. In *ICDE*, pages 101–110, 2000.

[73] Shivaram Venkataraman, Zongheng Yang, Michael J. Franklin, Benjamin Recht, and Ion Stoica. Ernest: Efficient performance prediction for large-scale advanced analytics. In *USENIX*, pages 363–378, 2016.

[74] Jinliang Wei, Wei Dai, Aurick Qiao, Qirong Ho, Henggang Cui, Gregory R. Ganger, Phillip B. Gibbons, Garth A. Gibson, and Eric P. Xing. Managed communication and consistency for fast data-parallel iterative analytics. In *SoCC*, pages 381–394, 2015.

[75] Xingda Wei, Sijie Shen, Rong Chen, and Haibo Chen. Replication-driven live reconfiguration for fast distributed transaction processing. In *USENIX*, pages 335–347, 2017.

[76] Gerhard Weikum, Christof Hasse, Alex Moenkeberg, and Peter Zabback. The COMFORT automatic tuning project, invited project review. *Inf. Syst.*, 19(5):381–432, 1994.

[77] Gerhard Weikum, Axel Moenkeberg, Christof Hasse, and Peter Zabback. Self-tuning database technology and information services: From wishful thinking to viable engineering. VLDB, pages 20–31, 2002.

[78] Wentao Wu, Jeffrey F. Naughton, and Harneet Singh. Sampling-based query re-optimization. In *SIGMOD*, pages 1721–1736, 2016.

[79] Eric P. Xing, Qirong Ho, Wei Dai, Jin Kyu Kim, Jinliang Wei, Seunghak Lee, Xun Zheng, Pengtao Xie, Abhimanu Kumar, and Yaoliang Yu. Petuum:

A new platform for distributed machine learning on big data. In *SIGKDD*, pages 1335–1344, 2015.

[80] Feng Yan, Olatunji Ruwase, Yuxiong He, and Trishul Chilimbi. Performance modeling and scalability optimization of distributed deep learning systems. In *KDD*, pages 1355–1364, 2015.

[81] Kenny Q. Ye. Orthogonal column latin hypercubes and their application in computer experiments. *Journal of the American Statistical Association*, 93(444):1430–1439, 1998.

[82] Dong Young Yoon, Ning Niu, and Barzan Mozafari. Dbsherlock: A performance diagnostic tool for transactional databases. In *SIGMOD*, pages 1599–1614, 2016.

[83] Ce Zhang and Christopher Ré. Dimmwitted: A study of main-memory statistical analytics. *PVLDB*, 7(12):1283–1294, 2014.

[84] Huan Zhang, Cho-Jui Hsieh, and Venkatesh Akella. Hogwild++: A new mechanism for decentralized asynchronous stochastic gradient descent. In *ICDM*, pages 629–638, 2016.

[85] Shen-Yi Zhao and Wu-Jun Li. Fast asynchronous parallel stochastic gradient descent: A lock-free approach with convergence guarantee. In *AAAI*, pages 2379–2385, 2016.

[86] Martin Zinkevich, Markus Weimer, Lihong Li, and Alex J. Smola. Parallelized stochastic gradient descent. In *NIPS*, pages 2595–2603. 2010.