M.S. THESIS

TreeML: Taming Hyper-parameter Optimization of Deep Learning with Stage Trees

Stage Tree를 활용한 딥러닝의 초 매개변수 최적화

2021 년 8 월

서울대학교 대학원

컴퓨터 공학부

신 안 재

TreeML: Taming Hyper-parameter Optimization of Deep Learning with Stage Trees

Stage Tree를 활용한 딥러닝의 초 매개변수 최적화

지도교수 전 병 곤

이 논문을 공학석사학위논문으로 제출함

2021 년 6 월

서울대학교 대학원

컴퓨터 공학부

신 안 재

신안재의 공학석사 학위논문을 인준함

2021 년 7 월

위 원 장	염 헌 영
부위원장	전 병 곤
위 원	이 영 기

Abstract

Hyper-parameter optimization is crucial for pushing the accuracy of a deep learning model to its limits. A hyper-parameter optimization job, referred to as a study, involves numerous trials of training a model using different training knobs, and therefore is very computation-heavy, typically taking hours and days to finish.

We observe that trials issued from hyper-parameter optimization algorithms often share common hyper-parameter sequence prefixes. Based on this observation, we propose TreeML, a hyper-parameter optimization system that reuses computation across trials to reduce the overall amount of computation significantly. Instead of treating each trial independently as in existing hyper-parameter optimization systems, TreeML breaks down the hyper-parameter sequences into stages and merges common stages to form a tree of stages (a stage tree). TreeML maintains an internal data structure, search plan, to manage the current status and history of a study, and employs a critical path based scheduler to minimize the overall study completion time. TreeML is applicable to not only single studies, but multi-study scenarios as well. Evaluations show that TreeML's stage-based execution strategy outperforms trial-based methods for several models and hyper-parameter optimization algorithms, reducing end-to-end training time by up to 2.76× (3.53×) and GPU-hours by up to 4.81× (6.77×), for single (multiple) studies.

Keywords: Deep Learning System, Hyper-parameter, Hyper-parameter Tuning
Student Number: 2019-24157

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Deep learning (DL) models have made great leaps in various areas including image classification [1, 2, 3], object detection [4], and speech recognition [5, 6]. However, such benefits come at a cost; training DL models require heavy datasets and long computations, which may take up to a week [7] even on hundreds of GPUs [7]. This cost becomes even more significant when we take hyper-parameter optimization into account. Since hyper-parameters can impact the trained models' quality, investigating the hyper-parameter search space often requires hundreds to thousands of training with different hyper-parameter settings [8]. Consequently, naively running hyper-parameter optimization requires an exceedingly large number of GPUs, and it is crucial to explore the hyper-parameter search space as efficiently as possible.

Training modern DL models requires changing hyper-parameter values on-the-fly during training to reach state-of-the-art accuracy, as they aim to minimize high-dimensional, non-convex loss functions. The learning rate hyper-parameter governs the training speed of a DL model. As a result, the DL community widely

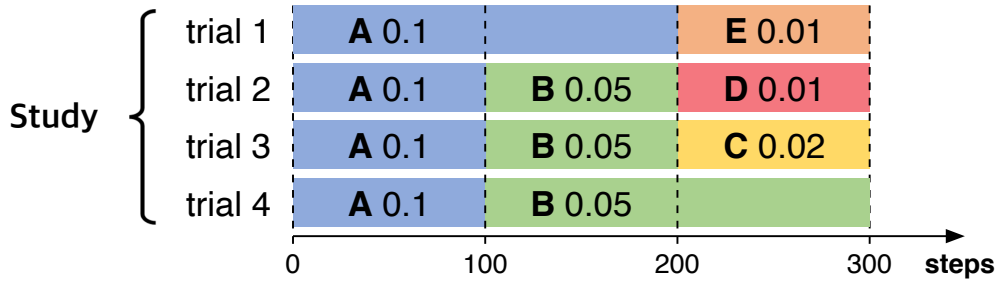


Figure 1.1: A hyper-parameter optimization study of trials that share common computations. A single hyper-parameter, learning rate, is being explored within the search space $\{0.1, 0.05, 0.02, 0.01\}$. Each trial is split into several stages. Each stage is labeled with an id (A-E) and its parameter value.

uses the learning rate as a sequence, and all DL frameworks provide various learning rate sequences that developers can plug-in to their code. Moreover, many papers also use other hyper-parameters as sequences to train DL models [7, 9, 10, 11, 12, 13]. However, existing hyper-parameter optimization systems [14, 15, 16, 17] do not consider hyper-parameters as *sequences* of values.

Tuning hyper-parameters as sequences creates an optimization opportunity of sharing common computations. Figure 1.1 shows a hyper-parameter optimization job, which we call a *study*. This study consists of four separate instances, or *trials*, each associated with different learning rate sequences. The first 100 training steps for all four trials can be shared, as they are operating on the same learning rate value, 0.1. We use the term *stage* to refer to this sharable execution unit. Similarly, for step range $[100, 200)$, trials 2, 3, and 4 have a common stage for learning rate 0.05. Instead of handling such common stages independently, we can execute them only once and share them across trials to avoid redundant computation and reduce the amount of resource (GPU-hours) used. We can merge the common stages and regard the set of trials as a tree

of stages – a *stage tree*. This framework-agnostic abstraction can express the computation dependencies of stages as a directed tree. Existing systems lack this key abstraction, missing the opportunity to eliminate redundant computation across trials.

However, building a system that handles trials as a stage tree to share computation is challenging because of the dynamic characteristics of hyper-parameter optimization. First, as trials are added and removed on-the-fly, the system must dynamically determine which stages to share across trials. Depending on the specific hyper-parameter sequences and trial submission timings, a newly added trial may or may not be able to reuse the result of an intermediate stage; the system must efficiently manage the states (checkpoints and evaluation metrics) of such stages so that no trial needlessly executes a stage that would otherwise be sharable. Second, stages must be scheduled in an online manner. In order to minimize the study's completion time, the system requires an online scheduling algorithm that allocates GPUs to stages while taking common stages into account.

To this end, we present TreeML, a hyper-parameter optimization system that finds and reuses redundant computations in hyper-parameter optimization jobs. TreeML uses a *search plan*, a tree-like data structure with append-only edges, to manage and reuse stage states for common stages in a clear, consistent fashion. Once added, edges are invariant to trial operations, allowing us a static structure for considering only current trials when sharing stages. TreeML also employs an online scheduler that considers critical paths among stages to minimize the overall completion time. The scheduler extracts a stage tree snapshot from the search plan and iteratively analyzes critical paths, removing the critical path from the stage tree and repeating the process with the remaining stages. The system schedules each critical path as a whole by batching the stages in the

same path, subsequently reducing the checkpoint saving and loading overheads when sharing computations.

We evaluated TreeML with three popular DL models (ResNet56, MobileNetV2, and BERT-Base) and three well-known hyper-parameter optimization algorithms (SHA, ASHA, grid search) on a cluster of 40 GPUs. Our evaluations show that TreeML outperforms Ray Tune, a black-box optimization system, reducing the end-to-end training time and GPU-hours of a single study up to 2.76× and 4.81×, respectively. For multi-study scenarios, TreeML can share redundant computations across studies and reduce the end-to-end training time and GPU-hours by up to 3.53× and 6.77×, respectively.

Chapter 2

Background and Motivation

In this section, we present a brief overview of hyper-parameters and its optimization. Then, we motivate the need for the abstraction of hyper-parameter sequences and discuss the challenges of applying such abstraction in a system.

2.1 Hyper-Parameter Optimization

Hyper-parameter optimization refers to the act of training multiple instances of a machine learning model with slightly differing training knobs, such as learning rate and batch size. We use the term *study* to refer to a single optimization run of a model over a certain search space of parameters. Each sub-procedure of a study associated with a set of parameters sampled from the given search space is called a *trial*. Specifically, a trial defines what hyper-parameter sequence the model should use to train. A trial can be split into one or more disjoint subsequences, referred to as stages in this paper.

There are many types of hyper-parameters as well as many possible values

for each hyper-parameter. The search space is often enormous, and the number of trials is usually in the hundreds and even thousands [8, 18, 19]. Therefore, hyper-parameter optimization is crucial in training deep learning models for high model quality. The model quality of trials with different hyper-parameter values may differ significantly, even if settings other than the hyper-parameters such as the model architecture and input data are kept the same across all trials [20].

Hyper-parameter sequences. Learning rate, one of the most critical hyper-parameters in DL, is a tunable value that controls how much model weights should be updated proportionally to its error. If the learning rate value is too small, the training process becomes very slow, and if the value is too large, the model weights may fail to converge to a stable value. As the model trains, the loss landscape changes and the learning rate must be adjusted, respectively, resulting in a sequence of learning rate values. Over a decade, the DL community have discovered many heuristics [1, 21, 22, 23, 24, 25, 26, 27, 28], which are supported natively by most DL frameworks [29, 30, 31].

Recent works have also applied this sequence heuristic to other hyper-parameters as well, such as batch size [12], drop-out ratio [32], optimizer [7], momentum [13], image augmentation parameters [9], training image input size [10], input sequence length [11], and network architecture parameters [10]. As training modern DL models involves minimizing high-dimensional, non-convex loss functions, we predict this trend of hyper-parameter sequences to become even more popular throughout the community.

The hyper-parameter sequences are usually manually sampled by researchers [19]. When manually tuning hyper-parameters, a common heuristic to discover a well-performing trial is local search, slightly modifying a previously attempted hyper-parameter sequence that showed good results. As a result, promising trials

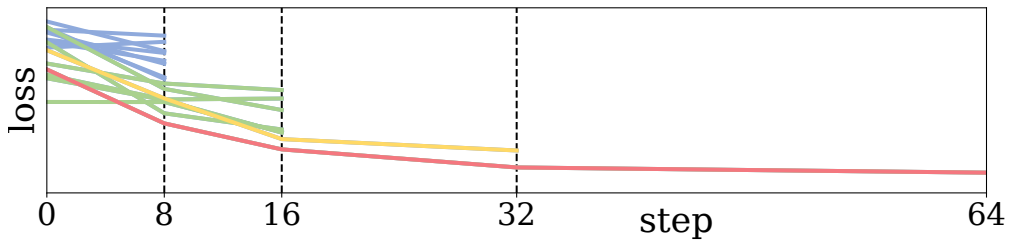


Figure 2.1: An illustration of Successive Halving (SHA) when reduction factor is 2. The search starts with 16 trials (lines). Only the trials with lower loss values are trained further over the decision boundaries (vertical lines). For every boundary, only half of the trials can proceed.

often share common subsequences in their hyper-parameter values.

Hyper-parameter optimization. Hyper-parameter optimization allocates resources to each trial in a non-uniform way. Promising trials with lower loss are trained more, and inferior trials are stopped early. For example, Successive Halving (SHA)[33] is a popular way to allocate more resource to trials which have better accuracy than others. We provide an example run of SHA in Figure 2.1. SHA has multiple decision boundaries, depicted as vertical dashed lines. SHA trains all trials until the decision boundary but advances only the trials with relatively lower loss value. In SHA, every boundary is a synchronous barrier; all trials are trained until the border before tested against other trials. However, ASHA [8], an asynchronous variant of SHA, only compares a trial only against completed trials. Therefore, a trial can advance to the next boundary without waiting for other trials to complete. SHA and ASHA compare trials after training a fixed amount of iterations, but some algorithms such as the median-stopping rule, dynamically kill trials whenever they perform poorly than expected[15].

Multiple studies potentially share common computation as well. Hyper-parameter optimization is a feedback-driven exploratory process where the user

constantly tries new search spaces and tuning heuristics [17, 34]. In this process, the user modifies the previous study to create a new one, thereby running multiple studies that share the same model and dataset. As a result, a hyper-parameter optimization system can benefit from sharing computation across multiple studies.

2.2 Challenges of Sharing Computations in Hyper-Parameter Optimization Jobs

As promising trials usually share common sequences in their hyper-parameter configurations, it makes sense to build a system that performs such common computations only once, avoiding redundant computations. Several systems have been proposed throughout the literature that applies computation sharing to increase computation efficiency, including machine learning systems [35, 36, 37, 38] that target a static set of jobs with configurations known beforehand, as well as systems from the big data domain [39, 40, 41, 42, 43, 44] that assume an online setting where jobs are dynamically submitted. Unfortunately, sharing computations in hyper-parameter optimization jobs involves new challenges due to the workload characteristics of hyper-parameter optimization.

C1: Dynamic computation sharing of trials. As opposed to static settings where all computations are known from the start, hyper-parameter optimization studies operate in a more online manner in which trials are constantly added and removed during a study. Thus, new common computations may emerge at runtime, and existing common computations may expire. This complicates matters, as any non-common computation can become a common computation in the future, and vice versa. Moreover, the unique pattern of sharable computations across trials motivates an abstraction tailored to hyper-parameter

optimization jobs (Section 3). A hyper-parameter optimization system must take such uncertainties into account and employ a computation sharing mechanism that adapts to such dynamics (Section 4.2).

C2: Online stage scheduling. The scheduling order of stages impacts the total completion time of trials because each stage saves a different amount of execution time, depending on the number of trials that share the stage. However, the exact saved time is unpredictable, as trials are added and removed dynamically. Moreover, depending on the scheduling algorithm, sharing computation may incur large overheads because model checkpoints must be saved to and loaded from the disk to be shared across trials. A hyper-parameter optimization system must schedule stages on-the-fly while considering the effects of computation sharing as well as the possible overheads (Section 4.3).

Chapter 3

Stage Tree

We now propose an abstraction for identifying common computations in hyper-parameter optimization trials: the *stage tree*. The stage tree abstraction is not a direct solution to solving the challenges described in Section 2.2. Rather, stage trees provide the basis for TreeML's two core system techniques (Sections 4.2 and 4.3).

We first briefly explain how individual hyper-parameter sequences are expressed. Users express sequences as mathematical functions with a non-negative integer domain. Then two subsequences are identical if they share the same function and domain. The sequences can be elementary functions such as *cosine*, or *exponential*, but also piecewise functions. For example, learning rate warmup [24] is a technique to increase the learning rate linearly for a few steps and then decaying the value using a different function. To express piecewise functions, elementary functions can be concatenated such as trial 1 in Figure 1.1. Each elementary function corresponds to a stage, which can be further split for merging.

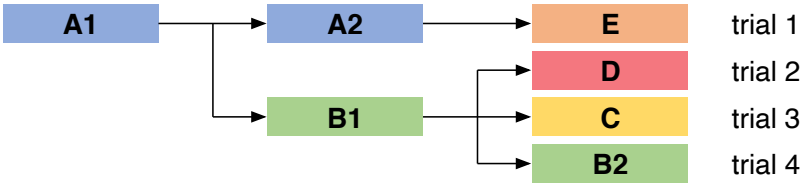


Figure 3.1: A stage tree formed from the trials of Figure 1.1. Stage $A1$ can be executed once to serve all four trials, while stage $B1$ can be shared by three trials. A stage can be split into shorter stages to match the length of a stage from another study that shares the same hyper-parameter value.

By merging common stages across trials in Figure 1.1, we get the tree-shaped arrangement of stages in Figure 3.1. In this form, it is evident that stages $A1$ and $B1$ can be shared by multiple trials. We refer to this form as a *stage tree*. The stage tree is mainly used to identify schedulable units when it comes to executing a hyper-parameter optimization study. Conveniently, a stage can be considered as a schedulable unit, while edges between stages express scheduling dependencies.

During the course of a study, the shape of the stage tree constantly changes as new trials arrive and old trials are deleted. When new trials arrive, new stages may be added to a stage tree, while existing stages can be split into shorter stages of smaller step ranges. Stages can even be deleted if the given hyper-parameter optimization algorithm decides to kill certain trials.

Figure 3.2 depicts how the stage tree from Figure 3.1 transforms when a new trial is added. Stage A of the new trial (Trial 5) cannot be merged into stage $A1$ or stage $A2$ in Figure 3.1, because neither of them has a matching step range (steps 0-150). Instead, stage $A2$ needs to be divided into stages $A3$ (steps 100-150) and $A4$ (steps 150-200), and then the new trial's last stage, F, is appended to $A3$. All stages that came after $A2$ in the original stage tree are

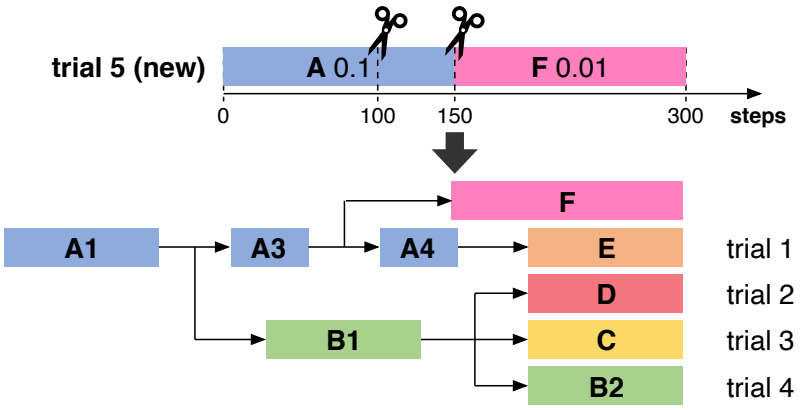


Figure 3.2: An illustration of a stage tree transformation when a new trial is added to the stage tree in Figure 3.1. Both the first stage in trial 5 and stage $A2$ in Figure 3.1's stage tree must be split into smaller stages, in order to merge trial 5 into the stage tree. As a result, trial 5 shares stages $A1$ and $A3$ with trial 1.

modified to follow $A4$ in the new stage tree.

Chapter 4

TreeML System Design

In this section, we introduce TreeML, a hyper-parameter optimization system that incorporates stage trees to run studies while automatically reusing computation for sharable stages. TreeML addresses the challenge of dynamic computation sharing (**C1**) by maintaining an internal data structure, *search plan*, to track all submitted trials and efficiently reuse model checkpoints and evaluation metrics for shared stages (Section 4.2). TreeML also implements a scheduling algorithm (**C2**) that considers critical paths in stage trees to minimize the overall makespan of the study (Section 4.3).

4.1 Overview

TreeML consists of various components to serve studies that dynamically send hyper-parameter optimization trials. A study application, whether an automated optimization algorithm or an interactive shell, communicates with the TreeML master via a *client library*. Instead of eagerly partitioning a trial into stages,

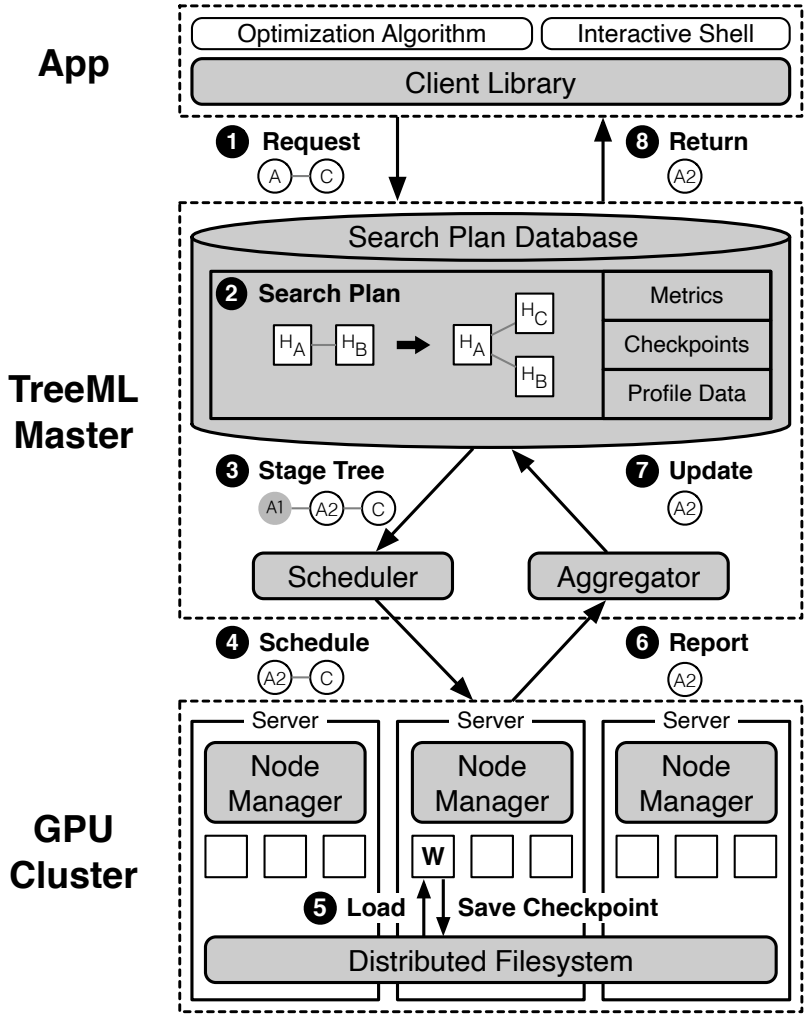


Figure 4.1: TreeML system architecture. Trial requests are issued by study applications, scheduled by the TreeML Master, and trained on the GPU cluster via workers (shown as **W**).

TreeML stores the trial information in the *search plan database*, in the form of a global *search plan*, so that new trials do not effect existing stages. After the search plan is updated, a transient stage tree is generated from the search plan and passed on to the *scheduler*, which in turn determines which stages need to

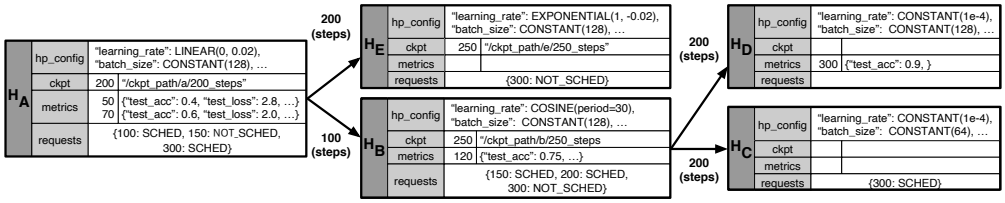


Figure 4.2: A search plan example of hyper-parameter configurations. Each node stores various fields, including hyper-parameter value functions for each hyper-parameter (`hp_config`) and a dictionary that marks the current stages that are waiting to be executed under this configuration (`requests`). Edges across nodes indicate sequential dependencies, e.g., H_B occurs after training a model for 100 steps under H_A, while H_C occurs after training a model for 100 more steps under H_B (a total of 200 preceding steps).

be run. The scheduler notifies the *node managers*, one on each GPU server, to run stages. The *aggregator* continuously collects evaluation metrics from the running stages to update the search plan database.

Figure 4.1 shows the overall flow of processing a trial in TreeML. The study application initiates the execution of a trial by submitting the trial to TreeML via the client library (①). Once a trial arrives at the system, the hyper-parameter sequence configuration of the trial is immediately compared with the search plan in the search plan database, and the search plan is adjusted accordingly (②). In case metrics that satisfy the trial are already present, then TreeML immediately returns the evaluation metrics of the trial back to the application. Otherwise, the search plan database generates a stage tree and notifies the scheduler to run new stages.

The scheduler decides which stages to run by examining the stage tree generated from the current search plan (③). Stages are given to GPU workers for execution (④), and the workers start computation by loading checkpoints

from the distributed filesystem (⑤). Workers periodically report evaluation metrics to the aggregator through the node manager. Each server has a node manager to gather metrics locally before passing them to the aggregator for reducing inter-server data traffic (⑥). The aggregator, upon receiving a set of metrics, updates the search plan (⑦). After repeating the scheduler-aggregator cycle multiple times, the final stage for a trial will eventually terminate, and the metrics are sent back to the application (⑧). Even if the trial has not finished yet, the application may request for metrics of intermediate stages at any time; TreeML will promptly return the metrics if they are available in the database.

4.2 Search Plan

4.2.1 Search Plan Data Structure

When a trial is submitted, TreeML must check if it can reuse an existing model checkpoint that shares hyper-parameter configurations. TreeML uses a data structure called *search plan* to maintain this information. A search plan is a tree that stores the hyper-parameter configuration history of submitted trials as well as model checkpoints and evaluation metrics. Each tree node in a search plan represents a hyper-parameter configuration starting from a certain training step. An edge between nodes indicates that the hyper-parameter configuration of the child node is appended to the configuration of the parent node, to form a hyper-parameter sequence. The number of training steps required to move from a parent node hyper-parameter configuration to a child node is annotated on the connecting edge. A path in a search plan represents a trial. Search plans have append-only edges; trial additions or removals do not remove existing edges. Such characteristics make individual paths invariant to other paths, or trials.

An example of a search plan is drawn in Figure 4.2. H_A, the root node of this

search plan, indicates a configuration of training a freshly initialized model (no parent node) with a linear learning rate ($\texttt{LINEAR}(x; a, b) = a + bx$) and constant batch size ($\texttt{CONSTANT}(x; a) = a$). Likewise, H_E indicates a configuration of an exponential learning rate ($\texttt{EXPONENTIAL}(x; a_0, \gamma) = a_0\gamma^x$) and constant batch size, starting from a model checkpoint that has been trained with H_A for 10 steps (note the directed edge between H_A and H_E).

Unlike stage trees, a search plan node is not a scheduling unit. The existence of a node does not necessarily imply that a trial, configured by that node, is currently running in the system. Rather, a node holds various statistics gathered by the system regarding the corresponding hyper-parameter configurations. TreeML can tell that a trial for that node has finished running by checking the following node fields:

- *hp_ config*: Hyper-parameter configurations for each target hyper-parameter. Widely used functions for hyper-parameter values, such as `CONSTANT`, `EXPONENTIAL`, `COSINE`, and `STEP`, are allowed.

- *ckpt*: A dictionary of file paths for checkpoints that were trained under this configuration.

- *metrics*: Intermediate values for evaluating the quality of the model checkpoint, like test/validation accuracy and loss.

- *requests*: A dictionary holding integers representing trial requests as keys, and state variables (`SCHED` or `NOT_SCHED`) as values. Each integer indicates the number of steps a model needs to be trained before evaluating it. For example, in Figure 4.2, the number 150 in H_A's *requests* field indicates that a trial requires training with H_A's hyper-parameter configuration for 150 steps. The state variable marks if that request has currently been

scheduled or not (Section 4.3). Note that a single request maps to a single trial; we use both terms interchangeably throughout the paper.

Adding a new trial to the search plan is done as follows. When a new trial arrives, the system traverses the search plan for a path that matches the trial's hyper-parameter sequence. If the trial has no matching path, new nodes are added to the search plan. Then we check the *ckpt* and *metrics* fields of the leaf node and immediately return the appropriate results in case no training is needed (e.g., there already is a metric that matches the request). In case results aren't already available, a new entry is added to the *requests* field of the node.

Revisiting the example illustrated in Figure 3.2 where a new trial submission requires splitting an existing stage $A2$ and adding a new stage F, TreeML handles this case by adding a search plan node corresponding to F as a child of H_A in Figure 4.2. H_A itself does not need to be modified. TreeML also marks the new node's *requests* field with the number 300, the step count of the new trial.

Removing a trial is done in a similar manner as adding a trial, except that nodes are not added nor deleted (to maintain the append-only edge property). The system traverses the search plan to find the node of the request that corresponds to the trial. If the request object is marked as `NOT_SCHED`, then we can simply delete the request from the database since the request has not yet been scheduled. On the other hand, if the request is marked `SCHED`, the database signals the scheduler to abort computation for the corresponding stage.

Going from search plans to stage trees.

While search plans are effective for managing the current status and history of a hyper-parameter study, stages are more straightforward as a scheduling unit for a system scheduler component to interact with. Thus, we use search

Algorithm 1 Build Stage Tree

```
1: function BuildStageTree(requests R)
2:     Initialize empty lookup table, L
3:     Initialize empty set of stages, S
4:     for r ∈ R do
5:         FindLatestCheckpoint(r, L)
6:     for end, start in L do
7:         S.put(Stage(start, end))
8:     return BuildTree(S)
9: function FindLatestCheckpoint(r, L)
10:    if r.node == null || r ∈ L then return
11:    for s ∈ {r.step - 1, r.step - 2, ..., r.node.init_step} do
12:        if checkpoint_exists(r.node, s) then
13:            L[r] = (r.node, s)
14:            return
15:    r_p = (r.node.parent, r.node.init_step)
16:    L[r] = r_p
17:    FindLatestCheckpoint(r_p, L)
```

plans as the basic format for carrying out trial additions and removals, but ultimately generate stage trees when a scheduling decision needs to be made. The generated stage trees are transient representations, used solely for creating scheduling units (stages), and are not kept in the system like search plans.

The generated stage tree serves all `NOT_SCHED` requests in the search plan; `SCHED` requests have already been processed by the scheduler, and thus do not need to be served again. Every request corresponds to a path in the stage tree where the start of the path starts from a existing checkpoint. For example, to

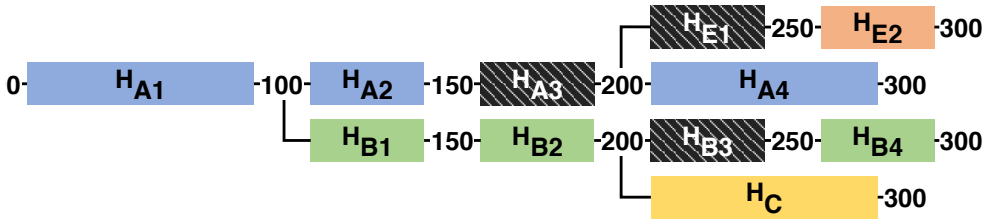


Figure 4.3: A stage tree generated from the search plan in Figure 4.2. The numbers below each stage indicate the step to start and stop training. Shaded stages indicate stages with checkpoints where training can be resumed from.

serve the request 300 of H_E in Figure 4.2, we first follow the edge from the preceding node (H_A), indicating that the request requires training for 200 steps with H_A's hyper-parameter configuration. Then, the request requires training for an additional 100 steps with H_E's hyper-parameter configuration, for a total of 300 steps. Note that since there already is a checkpoint for H_A at 200 steps, we don't actually have to perform any training with H_A.

Algorithm 1 describes the process of generating a stage tree from a search plan. The algorithm first checks all requests and breaks them down into smaller units that utilize available checkpoints (line 5). The lookup table L maps a child request object to a parent request object that is needed to reach the child. The request object is a tuple of a search plan node (e.g., H_E in Figure 4.2) and the number of training steps required to fulfill the request (e.g., 300 in H_E). Next, each <child, parent> pair is translated as a stage (line 7) that loads a model checkpoint from the parent, trains the model, and saves a new checkpoint at the child. Lastly, the function *BuildTree* is called (line 8) to construct a stage tree from the stages by connecting consecutive stages.

Figuring out the closest parent to a child is done with a helper function *FindLatestCheckpoint*. This function receives a request object and the lookup

table L as input. If no appropriate checkpoints are available in the current node, the function is recursively called with its parent node (line 17). Also, the parent node is added to the lookup table (line 16). It is worth noting that the lookup table is also used as a memoization mechanism (line 10).

Figure 4.3 illustrates the stage tree generated from the search plan in Figure 4.2 via Algorithm 1. A stage will be executed by resuming from the nearest available checkpoint, where available checkpoints are marked as shaded areas in the figure. For example, the stage denoted by H_{E2} with steps 250 to 300 in the figure will be trained after resuming from H_{E1}'s checkpoint at 250 steps (as seen in H_E's *ckpt* field).

4.2.2 Search Plan Database

TreeML stores all search plans that are currently being served in the search plan database. When a new trial is added, TreeML updates the search plan as described in Section 4.2.1. The various field entries in any node of the search plan, including checkpoints, metrics, and runtime profile data, can also be updated by the aggregator component.

Checkpoint caching The database stores pointers to each checkpoint, as well as metadata such as file size, reference count, and last used time. These checkpoints have two purposes: sharing computations and recovering from failures. Depending on its primary use, checkpoints are either *central* or *peripheral*. Workers automatically create central checkpoints at the end of a stage for computation reuse in child stages[1]. Additionally, users can configure the system to periodically create peripheral checkpoints in the middle of stages. Long stages can recover from peripheral checkpoints in case of failures.

[1]The *central* checkpoint of a leaf stage is actually optional, but useful when extending a trial by training more steps, which is a common pattern.

Trial additions and removals can cause a central checkpoint to become a peripheral one, or vice versa. Therefore, TreeML manages the two types of checkpoints in the same cache. The cache policy can evict any of the two checkpoints according to a cost-benefit ratio, proposed in Nectar [40].

Multiple search plans Trials may have no overlapping hyper-parameter sequences at all, in which case they cannot be represented with a single search plan. To cover such cases, the search plan database holds multiple search plans; when a new trial arrives, TreeML adds it to the search plan that has a matching root node hyper-parameter configuration. Managing multiple search plans also has the benefit of allowing TreeML to serve more than one study at once – studies on different models as well as different input datasets. As different search plans are mutually independent, TreeML does not require any kind of synchronization mechanism between search plans.

4.3 Scheduler

TreeML schedules computation on GPUs with stages as the basic scheduling unit. Since the number of stages that can run concurrently at a given moment usually exceeds the number of available GPUs in the cluster, TreeML utilizes a scheduler component to determine the stages to be run. The scheduler allocates GPUs to execute stages, and preempts stages associated with requests that have been canceled by the client.

The scheduler takes stage trees generated from the current search plans as inputs and schedules stages on GPU workers. A simple scheduling method would be to do a breadth-first traversal through all stage trees and schedule each stage one by one until all GPU workers have been assigned stages. However, we have found that this method leads to a large job makespan, due to stages on the critical path of the stage trees being scheduled relatively later than non-critical

path stages.

Instead, the scheduler computes the critical path of all given trees and schedules the longest critical path on a worker. At this point, all *request* entries in the search plan associated with this path are marked as `SCHED`. With multiple workers, the scheduler repeatedly finds the next longest path among unscheduled stages of all stage trees and schedules the path of stages on an idle worker. The longest path of a stage tree is the path that has the longest estimated execution time; the execution time of an individual stage is estimated by multiplying the number of steps of that stage by the execution time per step (profiled beforehand when a search plan node is newly added).

We also observed that the stage transition overhead for a worker is significant due to checkpoint saving and loading, when scheduling a path of stages on the worker. If two consecutive stages (connected as parent and child in the stage tree) are scheduled on the same worker, then there is no need to load the corresponding central checkpoint from the distributed filesystem before running the child stage because the checkpoint would still be present in GPU memory. The checkpoint save still needs to happen for other child stages, but can be done in the background, in parallel with training. To mitigate these overheads, the scheduler batches consecutive stages in the critical path and dispatches them as a single scheduling unit to a worker. The larger scheduling granularity improves locality by avoiding checkpoint overheads and further minimizes the end-to-end training time of a study.

The scheduler does not store any information regarding the execution states of stages. The scheduler operates in a stateless manner, relying entirely on the search plan to identify the stages that need to be run and the stages that have already run. After processing a stage tree, the scheduler simply releases the stage tree. Any stage batches (i.e., stage paths) that are yet to be scheduled on

a worker (due to all workers being busy) are put in a separate queue; as soon as a worker becomes idle, the aggregator is notified to update the search plan, and the scheduler sends the batch at the head of the queue to the idle worker, unless the queue is empty.

When the scheduler is triggered again later by another trial addition/removal to schedule more stages, the scheduler takes a new stage tree freshly generated from the latest search plan and repeats the whole scheduling process from the start. Note that triggering the scheduler while it is scheduling a stage tree does not affect the current scheduling; all unscheduled stage batches will be enqueued into the queue, behind the previous batches.

Chapter 5

Implementation

We have implemented TreeML in 5K lines of Python code. Communication between the TreeML master and node managers is done via the pub/sub interface provided by Apache Kafka 2.4.1 and Apache ZooKeeper 3.4.13. MySQL 8.0 is used to store system states in the search plan database. Kafka, ZooKeeper, and MySQL all run in Docker containers. Additionally, we use GlusterFS 6.9 as the distributed file system for saving and sharing checkpoints between nodes. Our current implementation of TreeML utilizes the deep learning framework PyTorch 1.5.0 to train DNN models, though TreeML's design is not tied to any specific framework.

5.1 Data Pipeline

We implemented a custom data pipeline for PyTorch that is compatible with stages. Two major updates were done. First, we modified the checkpoint mechanism of PyTorch's default data pipeline to include the current permutation of the dataset as a part of the checkpoint. This way, the data pipeline is able

to save its current position in the dataset when a stage terminates, and later resume from the same position for the next stage. Second, we added a feature to change the batch size of the data pipeline. When the batch size is changed, the data pipeline will flush every preprocessed batch from the queue, and relaunch the background threads so that they produce the correct batch samples.

5.2 Cost Estimator

Since hyper-parameters affect the GPU resource requirements of a stage, we implemented a cost estimator that uses linear regression to estimate a stage's completion time and number of GPUs required on profiling results. The scheduler uses this estimator to analyze the critical paths. Initially, when no historical data is available, the estimator predicts both latency and GPU requirement as one. We observe that cold predictions underestimate the GPU requirements of a stage, causing out of memory(OOM) errors. To effectively mitigate OOM errors, the stage is rescheduled with the double number of GPUs than the previous attempt.

5.3 Client Library

We implement a client library that serves three purposes. First, the client library is the entry point to TreeML. It serves as a thin communication layer for the study to add new trial requests. Second, the client library includes popular hyper-parameter optimization algorithms [33, 45, 8, 15, 46]. Lastly, the client library provides the API to express hyper-parameter sequences. The API is a collection of several *parametric families*. Each family represents a set of functions with identical parameters. Users can use the family to create new sequences. For example, the sinusoidal parametric family is a function that returns a new cosine function when given magnitude, period, and phase.

```
class MyTrainer(Trainer):
  ...
  def setup(self, hp):
  # hp is a dictionary of updated values
    if "lr" in hp:
      for group in self.optimizer.param_groups:
        group["lr"] = hp["lr"]
    if "bs" in hp:
      if self.train_loader:
        del self.train_loader
      self.train_loader = DataLoader(
        self.train_dataset,
        batch_size=hp["bs"]
      )
  ...
```

Figure 5.1: An example that updates the learning rate (lr) and batch size (bs) in the custom `Trainer` that the user should override. TreeML passes into `setup` the values of sequential hyper-parameters that should be updated.

To run a study in TreeML, users must first decide the model and dataset they want to use in the study, the types and values of hyper-parameters to tune, and the tuning algorithm to use. The training logic, which describes all things needed for training a model such as setting the values of each hyper-parameter, is defined by overriding the base `Trainer` class TreeML provides. The values of each hyper-parameter used in the `Trainer` will be drawn from the search space defined in Python by the user. The tuning algorithm specifies how to spawn, pause, or terminate trials that compose the study. Users may implement their own strategies, or simply choose from the tuners we provide. We will now take a closer look at each step a user must take to run a study in TreeML.

First, users should implement the training logic by overriding the base `Trainer` class TreeML provides. Users should write functions for initializing training (e.g. defining the model or loading the dataset), training for one logical

```
def get_search_space():
  hp = {
    "lr": [
      Constant(0.1),
      Exponential(0.1, 0.95)
    ],
    "bs": [
      Constant(128),
      MultiStep(128, [40], 2)
    ]
  }
  return GridSearchSpace(hp)
```

Figure 5.2: Defining a search space consisting of learning rate (lr) and batch size (bs) sequences in Python using the function definitions provided by TreeML. Two different sequences were defined for each hyper-parameter, resulting in four trials.

iteration (which may consist of multiple steps), evaluating the model trained so far and returning the metrics, saving, and loading checkpoints. One logical training iteration, executed by one call to the `Trainer`'s `train` function, should be long enough to avoid overheads, but short enough to regularly report progress. Often, a logical training iteration is set as one pass through the dataset. Whenever a hyper-parameter value is initialized or updated within a stage, TreeML calls the `Trainer`'s `setup` function with a dictionary containing updated values. Then, using theses values in `setup`, the user should make according changes to the appropriate attributes of the `Trainer`. Figure 5.1 illustrates a `setup` example.

Then, the user should define the search space they wish to explore using TreeML's implementation of well-known functions. Figure 5.2 displays a simple example that creates a search space over two types of hyper-parameters to use with the `MyTrainer` class defined previously in Figure 5.1. Unlike in existing frameworks, users can directly express hyper-parameters in the search space as

sequences, without having to embed the sequences as part of the training logic. Notice the matching keys between the search space and the `hp` dictionary passed into `MyTrainer`'s `setup`. Trials are sampled from this search space as a grid here, resulting in a total of four trials, but users who wish to implement conditional hyper-parameter spaces can optionally pass in a function to `GridSearchSpace` to filter out certain trials.

The last step is to create a study and a tuner. A study is defined by specifying the dataset, the command to run a trial, the checkpoint path, and the hyper-parameter set. The hyper-parameter set contains the types of hyper-parameters that are tuned in the study. For tuners, we provide several hyper-parameter optimization algorithms such as Successive Halving (SHA) [33], Hyperband [45], Asynchronous Successive Halving (ASHA) [8], median-stopping [15], and PBT [46] in the client library. Figure 5.3 illustrates how to create a study with a search space containing two types of hyper-parameters, and tune the study with a tuner that early-stops trials on milestones based on a certain metric.

TreeML's client library heavily utilizes Python's `asyncio` library. Instead of creating a new thread for each request, the library creates coroutines which are handled by the default Python event-loop. The tuning algorithms provided by TreeML take advantage of `asyncio` primitives, such as *wait_all* (block until all coroutines have finished) and *wait_any* (block until at least one coroutine has finished), to implement their logic.

Typically, hyper-parameter optimization algorithms submit several requests in parallel. In such situations, the client library batches the requests to reduce processing overhead at the search plan database.

```
hp_set = ["lr", "bs"]
study = Study(remote_url).create(
  dataset, command, ckpt_path, hp_set
)

schedule = Schedule.from_milestones(
  (5, 8), (10, 4)
)
tuner = EarlyStopTuner(
  schedule, search_space,
  metric.ExtractSingleNumber(
    "test_acc"
  )
)
tuner(study)

# Users can tune a study multiple times on different
    tuners
tuner2(study)

# Users can directly evaluate a certain trial on a
   specified step
study.eval(hp_config, step)
```

Figure 5.3: Running a study with an example tuner that trains 8 trials for 5 logical training iterations, early-stops 4 trials, and trains the remaining 4 trials up to 10 logical iterations. The killing decision is made based on the test accuracy as specified in the last argument to `EarlyStopTuner`.

Chapter 6

Evaluation

In this section, we first compare TreeML with Tune [14], a black-box hyper-parameter optimization framework built top of Ray [47]. We conducted four single study experiments comparing Tune and TreeML 6.1, and two multi-study experiments, each with a varying number of studies that run in parallel 6.2. Finally, we demonstrate the effect of our scheduling policy via comparison with other policies 6.3.

Environment Each experiment uses a homogeneous GPU cluster of five Amazon EC2 p2.8x instances, each with 8 NVIDIA Tesla K80 GPUs. A distributed file system using GlusterFS [48] is set up on Amazon EBS volumes. All experiment scripts are implemented in PyTorch 1.5.0 [29]. In all of our experiments, we measure the *end-to-end time* (the elapsed time from the start of the experiment to the end) and the *GPU-hours* (the sum of elapsed time each GPU was held for training).

For fair evaluation, we have made the following changes to Tune. We re-

Type	Function	Models		
		R	M	B
learning rate	MultiStep, CyclicLR, Warmup+MultiStep Warmup+Exponential, Warmup+Cosine,	✓	✓	✓
batch size	Constant, MultiStep	✓	✓	
momentum	Constant, MultiStep	✓		
weight decay	Constant	✓		
optimizer	Constant	✓	✓	
cutout size	Constant, MultiStep		✓	
input seq. length	Constant, MultiStep			✓

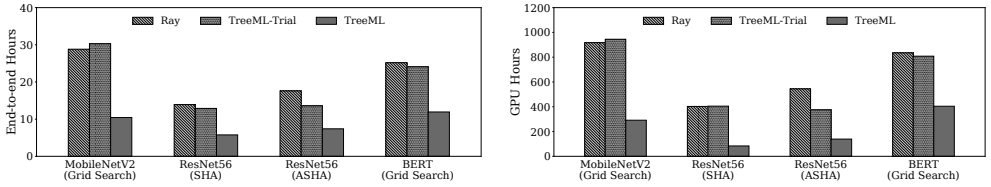
Table 6.1: Hyper-parameter types, functions and their memberships. Functions denote possible sequences samples. `R`, `M`, `B` each denote the search space of ResNet56, MobileNetV2, and BERT-Base. For example, the ResNet56 search space consists of five hyper-parameters.

implement the ASHA [8] algorithm to match the behavior specified in the original paper. Also, we alter Tune's runtime and API so that the system evaluates the model only whenever TreeML does. Model evaluation is relatively cheaper than training one epoch but causes huge overheads when done every batch. Tune's original implementation runs model evaluation every iteration creating huge overhead when tuning the BERT-Base model, which is trained in units of steps. Conversely, TreeML evaluates the model only when necessary.

Model	Dataset	Algorithm	Policy	# of trials	Merge rate
ResNet56	CIFAR-10	SHA	`reduction`=4, `min`=15, `max`=120	448	2.45
ResNet56	CIFAR-10	ASHA	`reduction`=4, `min`=15, `max`=120	448	2.45
MobileNetV2	CIFAR-10	Grid search	`max`=120	240	3.14
BERT-Base	SQuAD 2.0	Grid search	`max`=27000	40	2.05

Table 6.2: Specification of four studies. Each study is specified a model, dataset, hyper-parameter, tuning algorithm, and a tuning algorithm policy. `min` and `max` are the minimum and maximum training iterations for each trial. Each study is given its own search space represented by number of trials and merge rate.

Merge rate As our evaluation results vary on the configuration of the search



(a) End-to-end time (b) GPU-hour

Figure 6.1: Single-study experiment results for Tune, TreeML-Trial, and TreeML. Compared to Tune, TreeML can reduce end-to-end time by up to 2.76×, and GPU-hours by up to 4.81×.

space, we provide a metric m that summarizes the merging capability of the search space.

$$m = \frac{\text{Total steps}}{\text{Unique steps}}$$

Unique steps is defined as the number of training steps that are needed to train the entire search space, counting identical steps (redundant computation) as one step. *Total steps* is defined as the number of training steps while not considering redundant computations. For example, if there are N identical trials, the merge rate is $m = \frac{N}{1} = N$. Similarly, we define a k-wise merge rate m_k defined on k search spaces.

$$m_k = \frac{\text{Total steps of } k \text{ studies}}{\text{Unique steps across } k \text{ studies}}$$

The merge rate is the theoretical estimate of GPU-hour reduction in TreeML. Actual reduction values differ from this estimate due to three factors: optimization algorithm, checkpoint overhead, and hyper-parameter value. First, hyper-parameter optimization algorithms like SHA and ASHA early-stop trials, thereby pruning outer stages. As a result, the algorithm prunes stages that are less shared. We show experimental results that exhibit this effect in 6.1. Second, checkpoint saving and loading create overhead, decreasing the GPU hour

reduction gain. Lastly, the GPU time for a trial differs by the hyper-parameter value it has. Hyper-parameters like batch size, or optimizer require different computation time. Sharing a stage with hyper-parameters that corresponds to heavy computation is more beneficial than a stage that does not.

Hyper-parameters Table 6.1 summarizes hyper-parameters used for each search space. We use a total of seven hyper-parameters. Five hyper-parameters (learning rate, batch size, momentum, cutout[49] size, input sequence length[11]) are sampled as sequences, and two hyper-parameters (optimizer, weight decay) are sampled as point values. The search space is composed of commonly used functions in research papers, github repositories and Kaggle kernels.

6.1 Single Study

This section compares three different hyper-parameter optimization algorithm systems: Tune, TreeML, and TreeML-Trial. TreeML-Trial is an implementation of TreeML where no computation is reused.

We compare four different studies across three different hyper-parameter optimization systems. The design of each study is described in Table 6.2. Three different models, two different datasets, and three different hyper-parameter optimization algorithms are used for the different studies. We further train the best performing trial for 100 additional steps and the extra training time is accounted to the GPU-hour and the end-to-end time. Aside from system performance, we also compare the final model accuracy of the systems. For ResNet56 and MobileNetV2, we report the top-1 validation accuracy, and for BERT-Base, we report the F1 score.

Figure 6.1 depicts the end-to-end time and the GPU-hour of four studies. Tune and TreeML-Trial show comparable end-to-end time and GPU-hours, except for ASHA. In ASHA, the number of early-stopped trials depends on the

Model	Accuracy / F1 score [%]			
	Reported	Tune	Trial	Stage
ResNet56 (SHA)	93.03	93.08	92.89	93.27
ResNet56 (ASHA)	93.03	93.58	92.89	93.72
MobileNetV2	94.43	95.03	95.04	95.04
BERT-Base	76.28	78.42	78.57	78.18

Table 6.3: Final model metric of all four single-study experiments. Tune, TreeML, and TreeML-Trial reached the reported model accuracy or F1 score, reported from the original paper, popular GitHub repository, or dataset leaderboard.

completion order of trials. Because of its non-deterministic nature, Tune and TreeML-Trial differ in the total number of training steps.

Compared to Tune, TreeML can reduce end-to-end time and GPU-hours by up to 2.76× and 4.81×, respectively. As expected, for the two grid search studies, the merge rate (3.14×, 2.05×) matches the GPU-hour saving (3.15×, 2.07×). However, the GPU-hour saving of SHA and ASHA (4.81×, 3.92×) is significantly higher than its merge rate (2.45×). As discussed earlier, the early-stopping mechanism used by these algorithms prune stages that are less shared.

The top-1 accuracies and F1 scores reached in each study is shown in Table 6.3. In all four studies, TreeML successfully achieved top-1 accuracies and F1 scores higher than the reported target values. Moreover, in the experiment some studies reached higher model accuracy on TreeML compared to Tune, demonstrating that TreeML can finish training in a fraction of the time spent by Tune while possibly finding a better model checkpoint.

6.2 Multiple Studies

TreeML is able to merge computation across multiple studies. We compare the GPU-hour and the end-to-end time of TreeML and Tune when running several

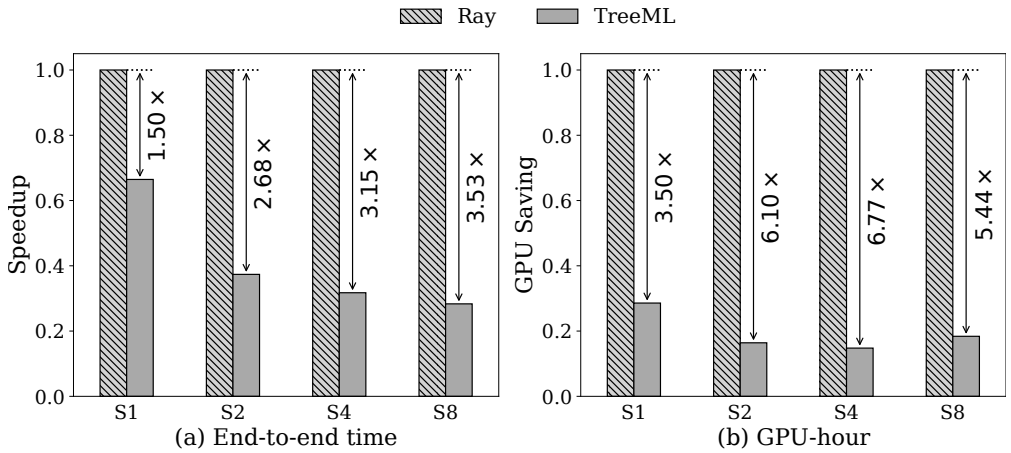


Figure 6.2: Multi-Study results with k-wise merge rates S2: 2.26, S4: 2.77, and S8: 2.47.

studies simultaneously. We vary the number of studies: 1, 2, 4, and 8, and refer to each case as S1, S2, S4, and S8. We create two search space sets where each set contains 8 subspaces. All studies spawn 144 trials where each trial train the ResNet20 model on the CIFAR-10 dataset, and tune learning rate and batch size.

The merge rate for the first search space set ranges from 1.5× to 2.73×. The k-wise merge rate for S2, S4, and S8 is 2.26, 2.77, and 2.47, respectively. Figure 6.2 depicts the results from this search space. We can see that with a relatively large merge rate between the studies, the GPU-hour and the end-to-end time shrinks by up to 6.77× and 3.53×.

The merge rate for the second search space set ranges from 1.2× to 2.1×. The k-wise merge rate for S2, S4, and S8 are 1.40, 1.19, and 1.66, respectively. Figure 6.3 depicts the results from this search space. Though the gains are smaller than in the previously defined search space due to lower merge rates, TreeML still reduces the GPU-hour and end-to-end time by up to 2.32× and 1.99×.

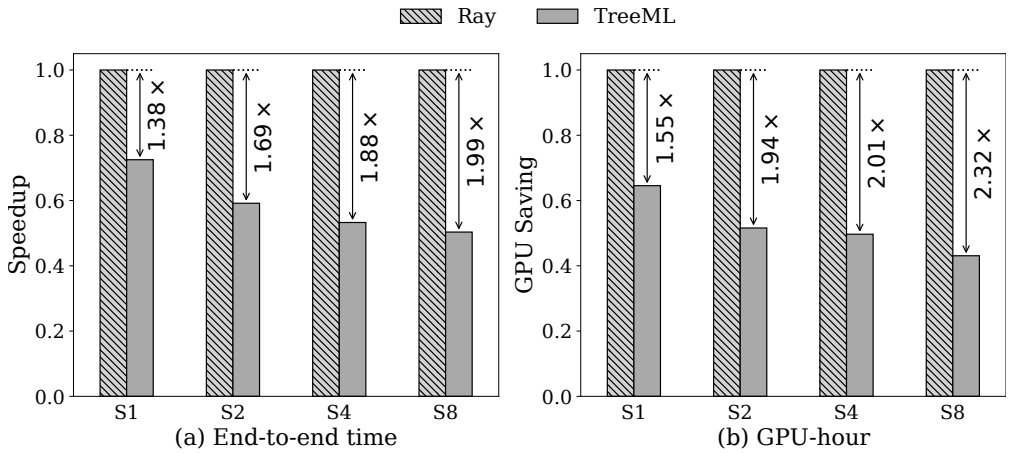


Figure 6.3: Multi-Study results with k-wise merge rates S2: 1.40, S4: 1.19, and S8: 1.66.

6.3 Scheduler Comparison

In this section, we compare TreeML's critical path scheduler with other possible scheduling policies. The three policies we compare are as follows.

- *BFS*: schedule stages in breadth-first search order
- *THR*: maximize throughput by scheduling stages with the largest number of child stages first
- *Critical*: the scheduler policy used in TreeML

As a fair comparison, the *Critical* scheduler does not batch stages; a checkpoint is always loaded from the disk. Therefore all policies have similar checkpoint overheads.

We compare the policies with two hyper-parameter optimization algorithms: grid search and ASHA. We train ResNet20 with CIFAR10, and tune the learning rate and batch size as hyper-parameters. The search space's merge rate is set to 3.55.

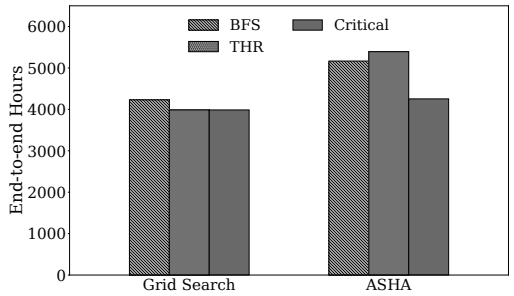


Figure 6.4: End to end time of three scheduling policies.

As shown in Figure 6.4, TreeML's critical scheduling policies has lower makespan compared to the two baselines. In grid search, *THR* and *Critical* have similar makespan, but in ASHA, *Critical* is 1.27 times faster. This is because when asha dynamically adds new trials, the critical path before trial submission tends to be also an critical path in the new stage tree; However the number of descendants the stage has changes as new trials are added.

Chapter 7

Related Work

Trial-based systems There have been recent systems [14, 15, 50, 51, 52, 17, 53, 54] for hyper-parameter optimization, helping users to manage their hyper-parameter optimization jobs on distributed environments. However, the trial-based systems miss the opportunities to reduce resource usage by reusing the common computation results.

Tune [14], for example, is a hyper-parameter optimization system built on top of Ray [47]. Since Tune does not understand the internals of a trial, a single trial cannot be further split into multiple stages to merge the common computation between trials, achieving sub-optimal performance compared to TreeML. Other popular trial-based hyper-parameter optimization systems such as Google Vizier [15], NNI [50], Optuna [51], Kubeflow [52], CHOPT [17], HyperDrive [53], and SageMaker [54] provide similar trial-level user APIs and schedule hyper-parameter optimization jobs on a trial basis, failing to share common computation as they cannot identify stages.

Computation sharing systems Reusing intermediate outputs across multiple jobs is a commonly used technique for multi-job systems. The workloads covered by such systems can largely be categorized into two groups: (i) a static setting in which all jobs are available at once so that the system can analyze sharable computation from the start, and (ii) a dynamic setting where jobs are continuously submitted to the system.

Several recent machine learning systems [35, 36, 37, 38, 55] fall into the former, static setting. [36] attempts to reduce the resource usage of hyper-parameter optimization jobs by caching intermediate data from data preprocessing steps and feature extraction steps. Pretzel [35] performs offline analysis on a given set of machine learning jobs and compiles a model plan that is able to reuse computation across the jobs, while Clipper [55] employs a prediction cache that stores the whole result of executing a job. Both Pretzel and Clipper target inference workloads, and thus are inapplicable to model training settings. Helix [38] selectively caches intermediate results for a job, depending on the storage cost and materialization cost, and reuses them in subsequent iterations. None of these systems particularly consider dynamically arriving jobs. On the other hand, TreeML's search plan data structure allows us to dynamically accommodate new trials and identify reusable stage results without running an offline analysis of all trials.

Various big data systems [39, 40, 41, 42, 43, 44] assume the latter, dynamic setting. Nectar [40] enables reusing common computation in DryadLINQ programs within a datacenter. Tachyon [39] implements an algorithm that bounds the recovery cost of any file in the whole job lineage by checkpointing certain key files. Although these systems take dynamically added jobs into account, they were not designed to handle hyper-parameter optimization workloads.

Systems focusing on a specific algorithm As hyper-parameter optimiza-

tion algorithms such as ASHA [8] and PBT [46] have been devised to optimize the resource usage on distributed environments, systems to efficiently run those algorithms have been introduced alongside with the algorithms themselves. However, the systems are not generic since each of these systems is specifically designed for executing only a specific algorithm. HyperSched [56] extends ASHA [8] and supports algorithms similar to ASHA. On the other hand, TreeML aims to support various hyper-parameter optimization algorithms including ASHA [8], SHA [33], PBT [46], and the median-stopping rule [15].

Chapter 8

Conclusion

TreeML is a hyper-parameter optimization system that removes redundant computation in the training process by breaking down the hyper-parameter sequences into stages, merging common stages to form a tree of stages, and executing a stage once per tree. TreeML is applicable to not only single-study scenarios but also multi-study scenarios. Our evaluations show that TreeML saves GPU-hours and reduces end-to-end training time significantly compared to Ray Tune on multiple models and hyperparameter optimization algorithms.

Appendix A

Appendix

A.1 Search Space

In this section, we explain how each hyper-parameter sequences are parameterized as functions.

- `Constant(init)`

  A constant function of value `init`

- `Exponential(init, gamma)`

  An exponentially decaying function that starts at `init` and multiplies the value with `gamma` every iteration.

- `MultiStep(init, milestones, gamma)`

  A piece-wise constant function that starts as `init` and decays by `gamma` every time the iteration number reaches a milestone. `milestones` is an array of desired milestone.

- `CyclicLR(init, max_y, period1, period2)`[22]

An oscillating function. All periods linearly increase from `init` to `max_y` for `period1` iterations, and linearly decrease back to `init` for `period2` iterations.

- `Cosine(init, min_y, period, gamma)`[57]
  A periodic cosine-annealing function. Each period starts as `init` and ends as `min_y`. The period length starts with `period`, and is multiplied by `gamma` every period.

- `Warmup(init, period, func)`[24]
  A linear function that starts as `init` and increases linearly for `period` iterations. After the linear increase, continue the sequence specified with `func`.

Bibliography

[1] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," in *CVPR*, 2016.

[2] A. Krizhevsky, "Learning Multiple Layers of Features from Tiny Images," *Tech report*, 2009.

[3] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A Large-Scale Hierarchical Image Database," in *CVPR*, 2009.

[4] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You Only Look Once: Unified, Real-Time Object Detection," in *CVPR*, 2016.

[5] A. Hannun, C. Case, J. Casper, B. Catanzaro, G. Diamos, E. Elsen, R. Prenger, S. Satheesh, S. Sengupta, A. Coates, and A. Y. Ng, "Deep Speech: Scaling up end-to-end speech recognition," *arXiv:1412.5567*, 2014.

[6] D. Amodei, S. Ananthanarayanan, R. Anubhai, J. Bai, E. Battenberg, C. Case, J. Casper, B. Catanzaro, Q. Cheng, G. Chen, *et al.*, "Deep speech 2: End-to-end speech recognition in english and mandarin," in *ICML*, 2016.

[7] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, J. Klingner, A. Shah, M. Johnson,

X. Liu, L. Kaiser, S. Gouws, Y. Kato, T. Kudo, H. Kazawa, K. Stevens, G. Kurian, N. Patil, W. Wang, C. Young, J. Smith, J. Riesa, A. Rudnick, O. Vinyals, G. Corrado, M. Hughes, and J. Dean, "Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation," *arXiv:1609.08144*, 2016.

[8] L. Li, K. Jamieson, A. Rostamizadeh, E. Gonina, J. Ben-tzur, M. Hardt, B. Recht, and A. Talwalkar, "A System for Massively Parallel Hyperparameter Tuning," in *MLSys*, 2020.

[9] D. Ho, E. Liang, X. Chen, I. Stoica, and P. Abbeel, "Population Based Augmentation: Efficient Learning of Augmentation Policy Schedules," in *ICML*, 2019.

[10] T. Karras, T. Aila, S. Laine, and J. Lehtinen, "Progressive Growing of GANs for Improved Quality, Stability, and Variation," in *ICLR*, 2018.

[11] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," *arXiv:1810.04805*, 2018.

[12] S. L. Smith, P.-J. Kindermans, and Q. V. Le, "Don't Decay the Learning Rate, Increase the Batch Size," in *ICLR*, 2018.

[13] J. Zhang and I. Mitliagkas, "YellowFin and the Art of Momentum Tuning," in *MLSys*, 2019.

[14] R. Liaw, E. Liang, R. Nishihara, P. Moritz, J. E. Gonzalez, and I. Stoica, "Tune: A Research Platform for Distributed Model Selection and Training," in *ICML AutoML Workshop*, 2018.

[15] D. Golovin, B. Solnik, S. Moitra, G. Kochanski, J. E. Karro, and D. Sculley, "Google Vizier: A Service for Black-Box Optimization," in *ACM SIGKDD*, 2017.

[16] H. Cui, G. R. Ganger, and P. B. Gibbons, "MLtuner: System Support for Automatic Machine Learning Tuning," *arXiv:1803.07445*, 2018.

[17] J. Kim, M. Kim, H. Park, E. Kusdavletov, D. Lee, A. Kim, J.-H. Kim, J.-W. Ha, and N. Sung, "CHOPT : Automated Hyperparameter Optimization Framework for Cloud-Based Machine Learning Platforms," *arXiv:1810.03527*, 2018.

[18] K. Mahajan, A. Balasubramanian, A. Singhvi, S. Venkataraman, A. Akella, A. Phanishayee, and S. Chawla, "Themis: Fair and Efficient GPU Cluster Scheduling," in *NSDI*, 2020.

[19] X. Bouthillier and G. Varoquaux, "Survey of achine-learning experimental methods at NeurIPS2019 and ICLR2020," *Tech report*, 2020.

[20] L. N. Smith, "A disciplined approach to neural network hyper-parameters: Part 1 – learning rate, batch size, momentum, and weight decay," *arxiv:1803.09820*, 2018.

[21] S. Ioffe and C. Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift," in *ICML*, 2015.

[22] L. N. Smith, "Cyclical Learning Rates for Training Neural Networks," in *WACV*, 2017.

[23] L. N. Smith and N. Topin, "Super-Convergence: Very Fast Training of Neural Networks Using Large Learning Rates," *arxiv:1708.07120*, 2017.

[24] P. Goyal, P. Dollár, R. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He, "Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour," *arXiv:1706.02677*, 2017.

[25] D. P. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," in *ICLR*, 2015.

[26] M. D. Zeiler, "ADADELTA: An Adaptive Learning Rate Method," *arXiv:1212.5701*, 2012.

[27] G. Hinton, N. Srivastava, and K. Swersky, "RMSProp." `http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf`.

[28] A. G. Baydin, R. Cornish, D. M. Rubio, M. Schmidt, and F. Wood, "Online Learning Rate Adaptation with Hypergradient Descent," in *ICLR*, 2018.

[29] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "PyTorch: An Imperative Style, High-Performance Deep Learning Library," in *NeurIPS*, 2019.

[30] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: A System for Large-Scale Machine Learning," in *OSDI*, 2016.

[31] 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:1512.01274*, 2015.

[32] D.-A. Clevert, T. Unterthiner, and S. Hochreiter, "Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)," in *ICLR*, 2016.

[33] K. Jamieson and A. Talwalkar, "Non-stochastic Best Arm Identification and Hyperparameter Optimization," in *AISTATS*, 2016.

[34] Q. Wang, Y. Ming, Z. Jin, Q. Shen, D. Liu, M. J. Smith, K. Veeramachaneni, and H. Qu, "ATMSeer: Increasing Transparency and Controllability in Automated Machine Learning," in *CHI*, 2019.

[35] Y. Lee, A. Scolari, B.-G. Chun, M. D. Santambrogio, M. Weimer, and M. Interlandi, "PRETZEL: Opening the Black Box of Machine Learning Prediction Serving Systems," in *OSDI*, 2018.

[36] L. Li, E. Sparks, K. Jamieson, and A. Talwalkar, "Reuse in Pipeline-Aware Hyperparameter Tuning," in *Systems for ML Workshop at NeurIPS*, 2018.

[37] E. R. Sparks, S. Venkataraman, T. Kaftan, M. J. Franklin, and B. Recht, "KeystoneML: Optimizing Pipelines for Large-Scale Advanced Analytics," in *ICDE*, 2017.

[38] D. Xin, S. Macke, L. Ma, J. Liu, S. Song, and A. Parameswaran, "Helix: Holistic Optimization for Accelerating Iterative Machine Learning," in *VLDB*, 2019.

[39] H. Li, A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica, "Tachyon: Reliable, Memory Speed Storage for Cluster Computing Frameworks," in *SOCC*, 2014.

[40] P. K. Gunda, L. Ravindranath, C. Thekkath, Y. Yu, and L. Zhuang, "Nectar: Automatic Management of Data and Computation in Datacenters," in *OSDI*, 2010.

[41] A. Jindal, S. Qiao, H. Patel, Z. Yin, J. Di, M. Bag, M. Friedman, Y. Lin, K. Karanasos, and S. Rao, "Computation Reuse in Analytics Job Service at Microsoft," in *ACM SIGMOD*, 2018.

[42] J. Camacho-Rodríguez, D. Colazzo, M. Herschel, I. Manolescu, and S. Roy Chowdhury, "Reuse-based Optimization for Pig Latin," in *ACM CIKM*, 2016.

[43] I. Elghandour and A. Aboulnaga, "ReStore: Reusing Results of MapReduce Jobs in Pig," in *ACM SIGMOD*, 2012.

[44] T. Nykiel, M. Potamias, C. Mishra, G. Kollios, and N. Koudas, "MRShare: Sharing Across Multiple Queries in MapReduce," in *VLDB*, 2010.

[45] L. Li, K. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar, "Hyperband: A Novel Bandit-based Approach to Hyperparameter Optimization," *Journal of Machine Learning Research*, vol. 18, no. 1, pp. 6765–6816, 2017.

[46] M. Jaderberg, V. Dalibard, S. Osindero, W. M. Czarnecki, J. Donahue, A. Razavi, O. Vinyals, T. Green, I. Dunning, K. Simonyan, C. Fernando, and K. Kavukcuoglu, "Population Based Training of Neural Networks," *arXiv:1711.09846*, 2017.

[47] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan, and I. Stoica, "Ray: A Distributed Framework for Emerging AI Applications," in *OSDI*, 2018.

[48] "GlusterFS," 2019. `https://www.gluster.org/`.

[49] T. Devries and G. W. Taylor, "Improved Regularization of Convolutional Neural Networks with Cutout," *arXiv:1708.04552*, 2017.

[50] Microsoft, "Neural Network Intelligence (NNI)," 2019. `https://github.com/Microsoft/nni`.

[51] T. Akiba, S. Sano, T. Yanase, T. Ohta, and M. Koyama, "Optuna: A Next-generation Hyperparameter Optimization Framework," in *ACM SIGKDD*, 2019.

[52] J. George, C. Gao, R. Liu, H. G. Liu, Y. Tang, R. Pydipaty, and A. K. Saha, "A Scalable and Cloud-Native Hyperparameter Tuning System," *arXiv:2006.02085*, 2020.

[53] J. Rasley, Y. He, F. Yan, O. Ruwase, and R. Fonseca, "HyperDrive: Exploring Hyperparameters with POP Scheduling," in *Middleware*, 2017.

[54] V. Perrone, H. Shen, A. Zolic, I. Shcherbatyi, A. Ahmed, T. Bansal, M. Donini, F. Winkelmolen, R. Jenatton, J. B. Faddoul, B. Pogorzelska, M. Miladinovic, K. Kenthapadi, M. Seeger, and C. Archambeau, "Amazon SageMaker Automatic Model Tuning: Scalable Black-box Optimization," *arxiv:2012.08489*, 2020.

[55] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica, "Clipper: A Low-Latency Online Prediction Serving System," in *NSDI*, 2017.

[56] R. Liaw, R. Bhardwaj, L. Dunlap, Y. Zou, J. E. Gonzalez, I. Stoica, and A. Tumanov, "HyperSched: Dynamic Resource Reallocation for Model Development on a Deadline," in *SOCC*, 2019.

[57] I. Loshchilov and F. Hutter:, "SGDR: Stochastic Gradient Descent with Warm Restarts," 2017.

Acknowledgements

This work would not have been possible without the members of the Software Platform Lab. I am especially grateful to my advisor, Byung-Gon, for his unlimited support of time and fund. I would especially like to thank the members of the Crane team (a.k.a. AutoML team) for working together to build such good systems.

I thank each individuals of my family, especially my wife, Yurim, who have told me pursuit for treasure and knowledge is joyful even in the darkest hours when we have each other.

초록

초 매개변수 최적화는 딥러닝 모델의 성능을 한계까지 끌어올리기 위해서는 필수 불가결한 과정이다. Study, 혹은 초 매개변수 최적화 작업은 각각 다른 초 매개변수 값을 가진 무수히 많은 딥러닝 학습 작업으로 이루어져 있으며, 각 학습 작업은 trial이라 불린다. 매우 많은 학습을 해야 하기에 연산이 많고, 짧게는 몇 시간에서 몇 주일씩 걸리기도 한다. 본 연구에서는 한 초 매개변수 최적화 작업으로부터 파생된 여러 trial 들의 초 매개변수 순열이 공통된 앞부분을 가짐을 밝힌다. 이러한 발견으로부터, Hippo라는 새 시스템을 제안한다. Hippo는 trial들에서 공통된 순열 앞부분을 찾아 연산 결과를 재활용하여 전체 연산량을 크게 줄인다. 기존 초 매개변수 최적화 시스템은 trial마다 매번 새로 학습하는 반면, Hippo는 주어진 초 매개변수 순열을 stage라는 작은 단위로 쪼개어 동일한 stage끼리 합쳐 stage tree의 형태로 만든다. Hippo는 Search Plan이라는 내부 자료구조를 통해 현 초 매개변수 최적화 study의 모든 상태를 기록하며, 임계 경로 기반 스케줄러를 통해 전체 작업 수행 시간을 최적화한다. Hippo는 한 번에 한 개의 study뿐만 아니라, 복수의 study도 동시에 수행할 수 있다. Hippo는 여러 모델과 여러 초 매개변수 최적화 알고리즘에서 기존의 초 매개변수 최적화 시스템보다 전체 수행 시간을 최대 2.76배, GPU hour를 최대 4.81배 최적화한다. 복수의 study를 동시에 수행할 경우 수행 시간은 최대 4.81배, GPU hour는 최대 6.77배 최적화 할 수 있다.

주요어: 딥러닝 시스템, 초매개변수, 초매개변수 최적화

학번: 2019-24157