Hippo: Sharing Computations in Hyper-Parameter Optimization

Ahnjae Shin
Seoul National University
aj.shin@snu.ac.kr

Joo Seong Jeong
Seoul National University
joosjeong@snu.ac.kr

Do Yoon Kim[*]
University of Michigan
kdoyoon@umich.edu

Soyoung Jung
Seoul National University
sy.jung@snu.ac.kr

Byung-Gon Chun[†]
Seoul National University
FriendliAI
bgchun@snu.ac.kr

ABSTRACT

Hyper-parameter optimization is crucial for pushing the accuracy of a deep learning model to its limits. However, 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 Hippo, 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, Hippo breaks down the hyper-parameter sequences into stages and merges common stages to form a tree of stages (a stage tree). Hippo 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. Hippo applies to not only single studies but multi-study scenarios as well. Evaluations show that Hippo'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.

PVLDB Reference Format:
Ahnjae Shin, Joo Seong Jeong, Do Yoon Kim, Soyoung Jung, and Byung-Gon Chun. Hippo: Sharing Computations in Hyper-Parameter Optimization. PVLDB, 15(5): 1038-1052, 2022.
doi:10.14778/3510397.3510402

1 INTRODUCTION

Deep learning (DL) models have made great leaps in various areas including image classification [18, 34, 51], object detection [73], and speech recognition [3, 32]. However, such benefits come at a cost; training DL models require heavy datasets and long computations, which may take up to a week [87] even on hundreds of GPUs [87].

[*]Work done at Seoul National University
[†]Corresponding author

Proceedings of the VLDB Endowment, Vol. 15, No. 5 ISSN 2150-8097.
doi:10.14778/3510397.3510402

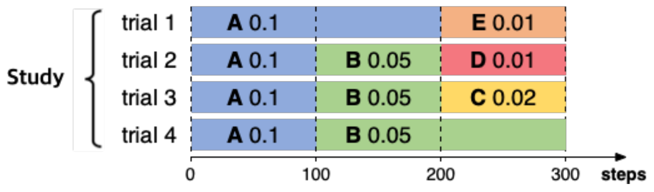


Figure 1: A hyper-parameter optimization study of trials that share common computations. A single hyper-parameter, learning rate, is explored within the search space $\{0.1, 0.05, 0.02, 0.01\}$**. Each trial is split into several stages. Each stage is labeled with an id (*A*-*E*) and its parameter value.**

This cost becomes even more significant when we take hyper-parameter optimization into account. Since hyper-parameters impact the trained models' quality, investigating the hyper-parameter search space often requires hundreds to thousands of training with different hyper-parameter settings [56]. Consequently, naively running hyper-parameter optimization requires an exceedingly large number of GPUs, and it is crucial to explore the 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 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 [20, 36, 46, 78, 87, 90]. However, existing hyper-parameter optimization systems [16, 27, 48, 59] do not consider hyper-parameters as *sequences* of values.

Tuning hyper-parameters as sequences creates an optimization opportunity of sharing common computations. Figure 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 Hippo, a hyper-parameter optimization system that finds and reuses redundant computations in hyper-parameter optimization jobs. Hippo 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. Hippo 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 Hippo 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 Hippo 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, Hippo 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.

The rest of the paper is organized as follows. Section 2 introduces hyper-parameter optimization and highlights related challenges. Section 3 proposes core representations, stage tree and search plan, for identifying and reusing redundant computations. Section 4 describes the Hippo design, and Section 5 elucidates implementation details. Section 6 presents evaluation results, Section 7 explains related work, and Section 8 concludes.

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 [7, 56, 61]. Therefore, hyper-parameter optimization is crucial in training DL 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 [76].

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 [5, 29, 34, 35, 40, 49, 75, 77, 89], which are supported natively by most DL frameworks [1, 10, 11, 43, 68, 74, 83, 84].

We sampled and analyzed 265 papers from recent machine learning conferences (CVPR'20 and ACL'20). We only include papers that train deep neural networks. We discover that 60%(159) use learning-rate as a sequence. Among the 159 papers that use learning-rate sequences, 57%(91) of them used a step-decay learning rate sequence. 27%(43) use complex sequences such as linear increase followed by a cosine decay curve. The remaining 16%(25) papers use elementary functions such as an exponential or polynomial decaying curve.

Another exemplary hyper-parameter tuned as a sequence is *input image size*. Scaling the size of input images during training has been a popular notion among both practitioners [6, 38, 39, 45, 66, 70] and academics [4, 13, 33, 46]. The image size hyper-parameter is actually composed of two independent hyper-parameters: width and height [37]. Nevertheless, the two hyper-parameters are tuned jointly with other hyper-parameters as a sequence. Such hyper-parameters include batch size [37], cut-out rate [15], dropout rate [82], per-image augmentation rate [82], and mixup ratio [82]. Compared to tuning scenarios where there is only a single hyper-parameter with a constant value, it is more challenging for developers to manually tune two or three kinds of hyper-parameters with the multi-step property.

Recent works have also applied this sequence heuristic to other hyper-parameters as well, such as batch size [78], drop-out ratio [12], cutout size [21], optimizer [87], momentum [90], image

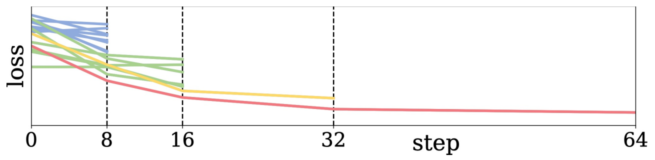


Figure 2: An illustration of Successive Halving (SHA) when reduction factor is 2. The search starts with 16 trials (lines). Only the trials with lower loss values are trained further over the decision boundaries (vertical lines). For every boundary, only half of the trials can proceed.

augmentation parameters [36], input sequence length [20]. In particular, we want to note the recent attention on dynamically scaling batch sizes. With the theoretical [62, 78] and empirical [19] discovery of changing batch sizes for better convergence, batch size scaling is now used in recent DL models, especially large language models [8].

Network architecture parameter is a multi-step hyper-parameter that recently gained attention, in the pursuit of resource-efficient machine learning. Hyper-parameters such as the number of model layers [28, 30], model dimension [30, 86], and input sequence length [30] all belong here. Considering the ever-advancing performance of large-scale pretrained models and their resource-intensive nature, we assume it will become more of a common practice in the future to tune network architecture parameters as a sequence. One remark about network architecture parameters is that they often get tuned in joint with other parameters [28, 30, 46]. For example, Gu et al. [30] scales the number of layers, model dimension, and input sequence length in a joint manner during training. Progressive-Gan [46] also dynamically adjusts batch size and model size simultaneously.

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) [42] is a popular way to allocate more resource to trials that have better accuracy than others. We provide an example run of SHA in Figure 2. 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 [56], an asynchronous variant of SHA, 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 [27].

Redundant computation in hyper-parameter optimization. The sequence characteristic of hyper-parameters create potential reusable computations both within a study and across multiple studies.

Many works [47, 50] emphasize the impact of hyper-parameter sequences on the model accuracy when training large models; two trials with overlapping prefixes may result in totally different accuracies depending on the later values of the hyper-parameter sequence. These sequences are usually manually sampled by researchers [7]. When manually tuning hyper-parameters, a common heuristic to discover an optimal combination of hyper-parameters is local search, slightly modifying a previously attempted hyper-parameter sequence that showed promising results. As a result, promising trials in a study often share common subsequences in their hyper-parameter values.

Multiple studies also potentially share common computation. Hyper-parameter optimization is a feedback-driven exploratory process where the user constantly tries new search spaces and tuning heuristics [48, 85]. Existing hyper-parameter optimization systems [17, 24, 48, 52, 80, 85] like Amazon SageMaker and Google Cloud ML Engine support creating a new study by adjusting a previous study's search space. As a result, identical prefixes exist across multiple studies.

Moreover, multiple tenants may tune the same model and dataset. Similar to the adjustment strategy, a hyper-parameter optimization algorithm [17, 69] may initialize the value of hyper-parameters by incorporating results of previous studies possibly submitted by other tenants. In addition, some crowd systems [25, 81] are designed to enable collaboration and competition among users to tune complex hyper-parameters in a time-efficient way.

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 [53, 57, 79, 88] that target a static set of jobs with configurations known beforehand, as well as systems from the big data domain [9, 23, 31, 44, 54, 67] 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

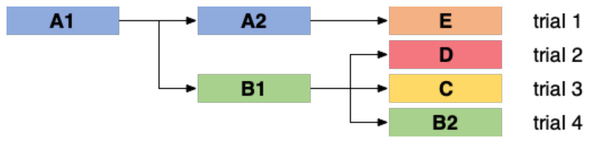


Figure 3: A stage tree formed from the trials of Figure 1. Stage *A*1 can be executed once to serve all four trials, while stage *B*1 can be shared by three trials. A stage can be split into shorter stages to match the length of a stage from another study that shares the same hyper-parameter value.

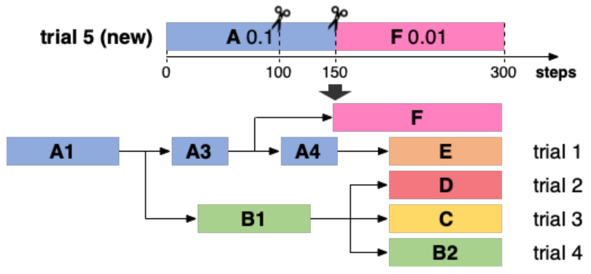


Figure 4: An illustration of a stage tree transformation when a new trial is added to the stage tree in Figure 3. Both the first stage in trial 5 and stage *A*2 in Figure 3 must be split into smaller stages, in order to merge trial 5 into the stage tree. As a result, trial 5 shares stages *A*1 and *A*3 with trial 1.

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 in order 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).

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 Hippo'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 [29] is a technique to increase the learning rate linearly for a few steps and then decay the value using a different function. To express piecewise functions, elementary functions can be concatenated such as trial 1 in Figure 1. Each elementary function corresponds to a stage, which can be further split for merging.

By merging common stages across trials in Figure 1, we get the tree-shaped arrangement of stages in Figure 3. In this form, it is

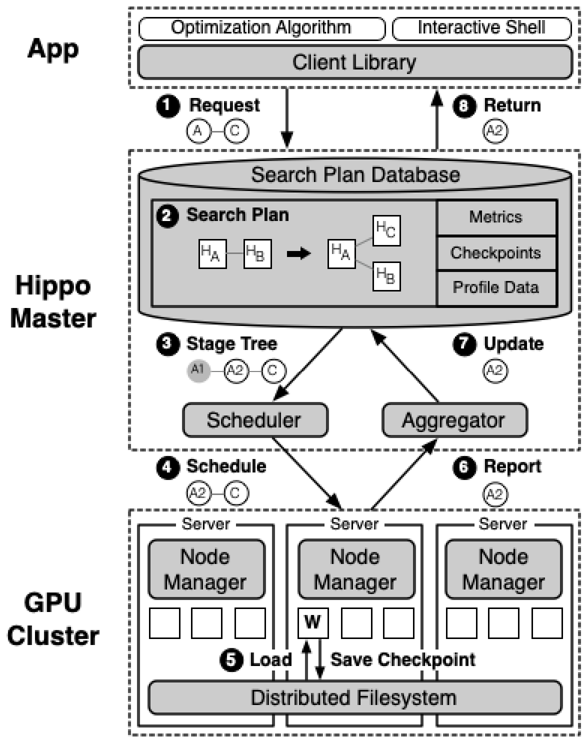


Figure 5: Hippo system architecture. Trial requests are issued by study applications, scheduled by the Hippo Master, and trained on the GPU cluster via workers (shown as W).

evident that stages *A*1 and *B*1 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 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 4 depicts how the stage tree from Figure 3 transforms when a new trial is added. Stage *A* of the new trial (Trial 5) cannot be merged into stage *A*1 or stage *A*2 in Figure 3, because neither of them has a matching step range (steps 0-150). Instead, stage *A*2 needs to be divided into stages *A*3 (steps 100-150) and *A*4 (steps 150-200), and then the new trial's last stage, *F*, is appended to *A*3. All stages that came after *A*2 in the original stage tree are modified to follow *A*4 in the new stage tree.

4 HIPPO SYSTEM DESIGN

In this section, we introduce Hippo, a hyper-parameter optimization system that incorporates stage trees to run studies while automatically reusing computation for sharable stages. Hippo 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). Hippo 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

Hippo 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 Hippo master via a *client library*. Instead of eagerly partitioning a trial into stages, Hippo 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 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 5 shows the overall flow of processing a trial in Hippo. The study application initiates the execution of a trial by submitting the trial to Hippo 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). If metrics that satisfy the trial are already present, Hippo 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 stages to run from 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; Hippo will promptly return the metrics if they are in the database.

4.2 Search Plan

4.2.1 Search Plan Data Structure. When a trial is submitted, Hippo must check if it can reuse an existing model checkpoint that shares hyper-parameter configurations. Hippo 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 6. 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 ($\texttt{EXP}(x; a_0, \gamma) = a_0\gamma^x$) and constant batch size, starting from a model checkpoint that has been trained with H_A for 200 steps (note the directed edge between H_A and H_E).

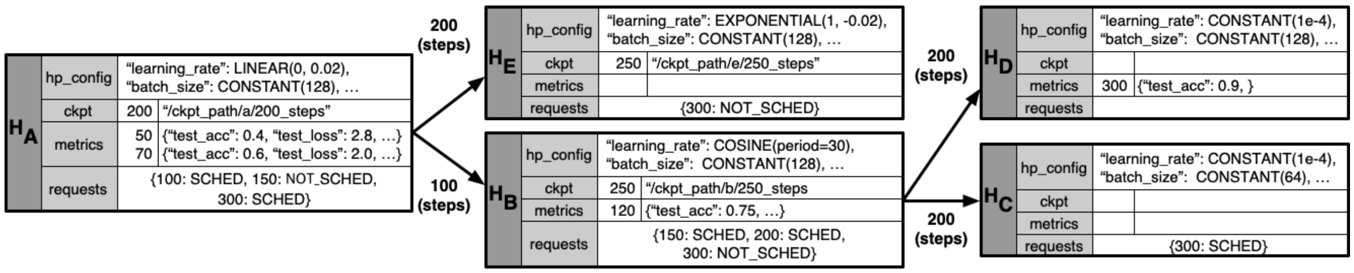
Unlike stage trees, a search plan node is not a scheduling unit. The existence of a node does not necessarily imply that a trial, configured by that node, is currently running in the system. Rather, a node holds statistics gathered by the system regarding the corresponding hyper-parameter configurations. Hippo 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 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 6, 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 4 where a new trial submission requires splitting an existing stage $A2$ and adding a new stage F, Hippo handles this case by adding a search plan node corresponding to F as a child of H_A in Figure 6. H_A itself does not need to be modified. Hippo 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



H_A
hp_config
"learning_rate": LINEAR(0, 0.02), "batch_size": CONSTANT(128), ...
ckpt
200 "/ckpt_path/a/200_steps"
metrics
50 {"test_acc": 0.4, "test_loss": 2.8, ...}
70 {"test_acc": 0.6, "test_loss": 2.0, ...}
requests
{100: SCHED, 150: NOT_SCHED, 300: SCHED}
200 (steps)
100 (steps)
H_E
hp_config
"learning_rate": EXPONENTIAL(1, -0.02), "batch_size": CONSTANT(128), ...
ckpt
250 "/ckpt_path/e/250_steps"
metrics
requests
{300: NOT_SCHED}
H_B
hp_config
"learning_rate": COSINE(period=30), "batch_size": CONSTANT(128), ...
ckpt
250 "/ckpt_path/b/250_steps
metrics
120 {"test_acc": 0.75, ...}
requests
{150: SCHED, 200: SCHED, 300: NOT_SCHED}
200 (steps)
200 (steps)
H_D
hp_config
"learning_rate": CONSTANT(1e-4), "batch_size": CONSTANT(128), ...
ckpt
metrics
300 {"test_acc": 0.9, }
requests
H_C
hp_config
"learning_rate": CONSTANT(1e-4), "batch_size": CONSTANT(64), ...
ckpt
metrics
requests
{300: SCHED}

Figure 6: 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).

Algorithm 1 Build Stage Tree

```
 1: function BuildStageTree(requests $R$)
 2:     Initialize empty lookup table, $L$
 3:     Initialize empty set of stages, $S$
 4:
 5:     for $r \in R$ do
 6:         FindLatestCheckpoint($r$, $L$)
 7:     for $end$, $start$ in $L$ do
 8:         $S$.put(Stage($start$, $end$))
 9:     return BuildTree($S$)
10:
11: function FindLatestCheckpoint($r$, $L$)
12:     if $r$.node == null || $r \in L$ then
13:         return
14:
15:     for $s \in \{r$.step - 1, $r$.step - 2, ..., $r$.node.init_step$\}$ do
16:         if checkpoint_exists($r$.node, $s$) then
17:             $L[r]$ = ($r$.node, $s$)
18:             return
19:
20:     $r_p$ = ($r$.node.parent, $r$.node.init_step)
21:     $L[r] = r_p$
22:     FindLatestCheckpoint($r_p$, $L$)
```

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 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 path starts from an existing checkpoint. For example, to serve the request 300 of H_E in Figure 6, 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 6). 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 6) 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 8) 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 9) 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 22). Also, the parent node is added to the lookup table (line 21). It is worth noting that the lookup table is also used as a memoization mechanism (line 12).

Figure 7 illustrates the stage tree generated from the search plan in Figure 6 via Algorithm 1. A stage will be executed by resuming from the nearest available checkpoint, where available checkpoints are marked as shaded areas. 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).

Algorithm Analysis. Let us define $|R|$ as the number of requests, N as the number of nodes in the search plan, and d as the height of the search plan. The worst-case time complexity of

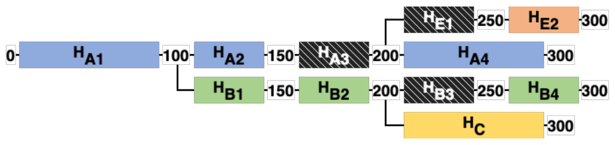


Figure 7: A stage tree generated from the search plan in Figure 6. The numbers below each stage indicate the step to start and stop training. Shaded stages indicate stages with checkpoints where training can be resumed from.

`FindLatestCheckpoint` is $O(d)$. The worst-case occurs when the lookup table is empty and the given request points to a step in a leaf node; the function should recursively call itself from the leaf node to the root node. However, as the output of this function is memoized, a node is only visited once for every child it has. Therefore, the function visits non-leaf nodes as many times as the number of edges ($N - 1$) and visits leaf nodes as many times as the number of requests ($|R|$). Therefore, the worst-case time required to run `FindLatestCheckpoint` for all $|R|$ requests is $O(N + |R|)$. Since `BuildTree` can be done in $O(N)$, worst-case time of `BuildStageTree` is $O(N + |R|)$.

In practice, the scheduler calls the algorithm consecutively. Similar to the dynamic programming method used in the algorithm, checkpoints created from processing a previously created stage tree acts as memoization that warm-starts the following algorithm invocation. Therefore, the worst-case time complexity for creating all stage trees throughout a study is $O(N' + |R'|)$. R' is the total number of requests submitted through the study, and N' is the final number of nodes in search plan.

4.2.2 Search Plan Database. Hippo stores all search plans that are currently being served in the search plan database. When a new trial is added, Hippo 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. 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. 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.

Trial additions and removals can cause a central checkpoint to become a peripheral one, or vice versa. Therefore, Hippo 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 [31].

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, Hippo 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 Hippo 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, Hippo does not require any kind of synchronization mechanism between search plans.

4.3 Scheduler

Hippo 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, Hippo 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.

5 IMPLEMENTATION

We have implemented Hippo in 5K lines of Python code. Communication between the Hippo 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 [63] containers. Additionally, we use GlusterFS [72] as the distributed file system for saving and sharing checkpoints between nodes. Our current implementation of Hippo utilizes the DL framework PyTorch 1.5.0 to train DNN models, though Hippo's design is not tied to any specific framework.

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.

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 double the number of GPUs of the previous attempt.

Client Library. We implement a client library that serves three purposes. First, the client library is the entry point to Hippo. 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 [27, 41, 42, 55, 56]. 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.

Table 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.

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			✓

6 EVALUATION

In this section, we first compare Hippo with Tune [59], a black-box hyper-parameter optimization framework built on top of Ray [65]. We conducted four single study experiments comparing Tune and Hippo (Section 6.1), and two multi-study experiments, each with a varying number of studies that run in parallel (Section 6.2). Then we show simulation results on how final accuracy changes according to different hyper-parameter search spaces (Section 6.3). We also compare how Hippo and Tune behave under different dynamic scenarios (Section 6.4). Finally, we demonstrate the effect of our scheduling policy via comparison with other policies (Section 6.5).

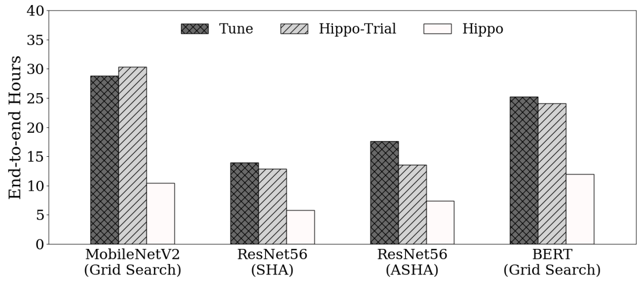
Environment. Each experiment uses a homogeneous GPU cluster of five Amazon EC2 p2.8x instances, each with 8 NVIDIA Tesla K80 GPUs. A distributed file system using GlusterFS is set up on Amazon EBS volumes. All experiment scripts are implemented in PyTorch 1.5.0 [68]. In all of our experiments, we measure 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-implement the ASHA [56] 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 Hippo 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, Hippo evaluates the model only when necessary.

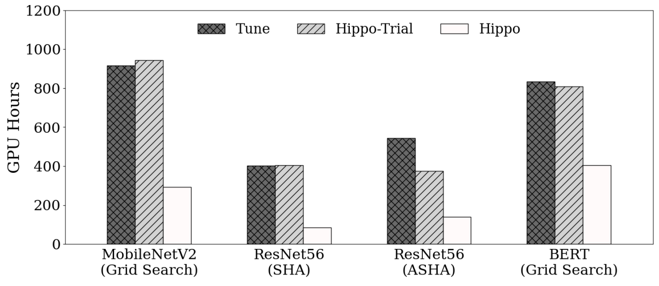
Merge rate. As our evaluation results vary on the configuration of the search space, we provide a metric m that summarizes the

Table 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.

Model	Dataset	Tune Algorithm	Algorithm Policy	# of trials	Merge rate (m)
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



(a) End-to-end time



(b) GPU-hours

Figure 8: Single-study experiment results for Tune, Hippo-Trial, and Hippo. Compared to Tune, Hippo can reduce end-to-end time by up to 2.76×, and GPU-hours by up to 4.81×.

merging capability of the search space.

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

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.

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

The merge rate is the theoretical estimate of GPU-hour reduction in Hippo. 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 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[21] size, input sequence length[20]) 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, Hippo, and Hippo-Trial. Hippo-Trial is an implementation of Hippo 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 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. We train ResNet56 and MobileNetV2 from scratch, but we fine-tune BERT-Base.

Figure 8 depicts the end-to-end time and the GPU-hour of four studies. Tune and Hippo-Trial show comparable end-to-end time and GPU-hours, except for ASHA. In ASHA, the number of early-stopped trials depends on the completion order of trials. Because of its non-deterministic nature, Tune and Hippo-Trial differ in the total number of training steps.

Compared to Tune, Hippo 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

Table 3: Final model metric of all four single-study experiments. Tune, Hippo, and Hippo-Trial reached the reported model accuracy or F1 score, reported from the original paper, popular GitHub repository, or dataset leaderboard.

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

rate (2.45×). As discussed earlier, the early-stopping mechanism used by these algorithms prune stages that are less shared. This reduces the search space size, increasing the effective merge rate.

The top-1 accuracies and F1 scores reached in each study is shown in Table 3. In all four studies, Hippo 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 Hippo compared to Tune, demonstrating that Hippo can finish training in a fraction of the time spent by Tune while possibly finding a better model checkpoint.

In our experiment results in section 6.1, the best performing trial had dynamically changing hyper-parameters other than learning rate. The ResNet56 model had both learning rate and batch size as dynamic values. MobileNetV2 had learning rate, batch size, and cutout. BERT-Base had learning rate and sequence length.

6.2 Multiple Studies

Hippo is able to merge computation across multiple studies. We compare the GPU-hour and the end-to-end time of Hippo and Tune when running multiple 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 9-(a) 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 9-(b) depicts the results from this search space. Though the gains are smaller than in the previously defined search space due to lower merge rates, Hippo still reduces the GPU-hour and end-to-end time by up to 2.32× and 1.99×.

Note that whether a hyper-parameter is parameterized with a continuous function (e.g., `Exp(init, gamma)`) or a discrete function (e.g., piecewise linear) does not affect the fact that trials can share hyper-parameter prefixes. For instance, two trials that share the same `init` and `gamma` values for a exponential learning-rate will have completely identical hyper-parameter values for all training steps, resulting in a merge rate of 2.

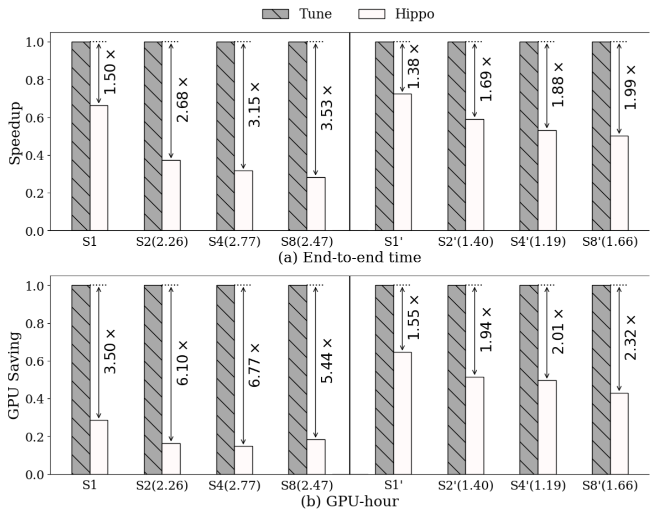


Figure 9: End-to-end time speedups and GPU-hour savings of two multi-study experiments. For each experiment, we define a search space and vary the number of studies from one to eight. The k-wise merge rate for each run is shown in parenthesis. For example, S2(2.26) means there were two studies submitted and the k-wise merge rate was 2.26. All values are normalized with respect to Tune.

6.3 Simulated Experiments

In this section, we show that merging computations does not affect model performance negatively. We collected traces of the ResNet56 model (training time, checkpointing time, evaluation accuracy) in the previous experiments (Section 6.1,6.2). With the collected trace, we created a search space superset. From the superset, we create several random search space subsets. Each search space subset has different number of trials, and merge rate. Using the evaluation accuracy values collected, each subset is undergone a simulation to calculate the final accuracy, and end-to-end job completion time. The simulation assumes the same setting as in Section 6.1. We also randomize the hyper-parameter optimization algorithm used (one of *gridsearch*, *sha*, *asha*). The simulator is implemented by emulating the *scheduler* and *aggregator* in Figure 5. The simulator yields the same final accuracy as in Table 3.

We sampled search spaces with six different numbers of trials and ran simulations on each search space. Figure 10 shows the final accuracy for each of the search spaces. As the number of trials in a search space increases, the amount of reusable computation and merge rate also increases. We can observe that search spaces with high merge rates also result in high final accuracy.

6.4 Dynamic Scenarios

In this section, we evaluate Hippo under different dynamic scenarios. To understand the merging ability of Hippo with varying levels

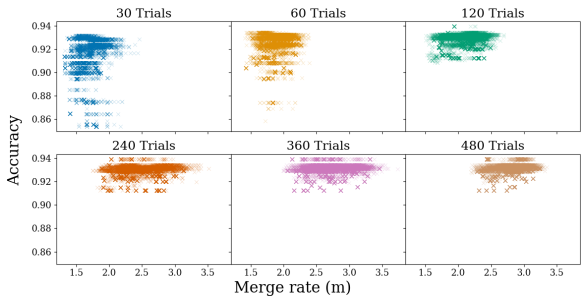


Figure 10: Simulation results on various search space subsets composed of different number of trials. The x-axis shows the merge rate of individual subsets, and the y-axis shows the final accuracy of a study optimized upon the subset.

of dynamism, we define a hyper-parameter tuner that submits requests to Hippo. We model the request arrival pattern from the tuner as a poisson process. The rate parameter γ is varied, simulating different levels of dynamism. For higher γ values, the average arrival rate of requests would be lower.

Figure 11 shows Hippo's merging ability with respect to a range of dynamic settings. The different colors in the figure correspond to a different random generator seed value which controls the permutation of the requests to be submitted. We measure the actual merge rate of the study. Unlike in the static setting where all trials are submitted simultaneously, new reusable computations occur dynamically for hyper-parameter optimization algorithms. This is because Hippo needs to compute them, although it would not be necessary if the information were available beforehand. Therefore, the more dynamic the tuner is, the actual merge rate decreases. Figure 11 shows such characteristic of Hippo. Tuners that submitted trials with average interval times more than 20 seconds had a drop in merge rate. In practice, Hippo mitigates this problem by introducing peripheral checkpoints (Section 4.2.1). Checkpoints are created periodically in the middle of stages to be reused in later submitted requests.

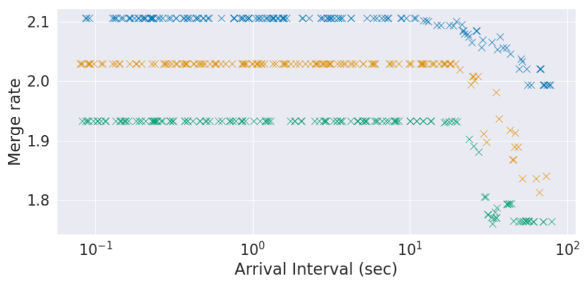


Figure 11: Hippo's merging ability in dynamic settings. The x-axis represents the average arrival interval of a request. The y-axis represents the actual merge rate of the study. The three colors represent different seed values. The different seed values change the permutation of the submitted requests. Hippo can merge almost every computation when trials are submitted frequently (with arrival interval 0.1s 20s).

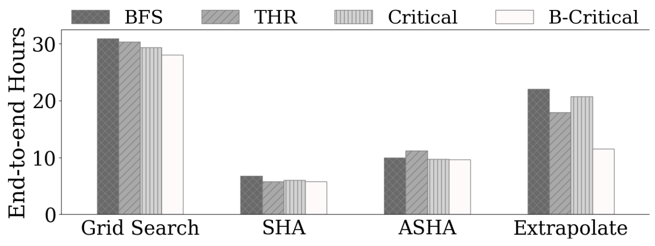


Figure 12: End-to-end time of four scheduling policies on four hyper-parameter optimization algorithms.

6.5 Scheduler Comparison

In this section, we compare Hippo's critical path scheduler with other possible scheduling policies on the simulator used in Section 6.3. The four 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*: Hippo's scheduler policy
- *B-Critical*: Hippo's scheduler policy with stage batching

We compare the policies with four hyper-parameter optimization algorithms: grid search, SHA, ASHA, and learning curve extrapolation [22]. The extrapolation optimization algorithm uses curve-fitting to predict if the given trial would reach the desired accuracy. For our experiments, the algorithm estimates the probability of the trial surpassing accuracy 93.03% before epoch 120. We follow the original paper's choice of configurations; at every 30 epochs, the trial is early-stopped if the probability of reaching the desired accuracy is below 5%.

Figure 12 depicts the end-to-end time of four hyper-parameter optimization algorithm with four different scheduling policies. The *B-Critical* policy was the fastest across all four optimization algorithms. The end-to-end time speedup of *B-Critical* relative to BFS is 1.10x (Grid Search), 1.17x (SHA), 1.03x (ASHA), 1.92x (Extrapolate). The speedup of *Critical* relative to *BFS*/*THR* is 1.05x/1.03x (Grid Search), 1.16x/0.97x (SHA), 1.03x/1.15x (ASHA), 1.06x/0.86x (Extrapolate).

For SHA, *Critical* is slightly slower than *THR*. This is because the resulting search plan of the SHA study contained 28 paths with a similar cost to the critical one. Similarly, if the stages require similar training costs, multiple nearly-critical paths can exist. Assuming there are enough GPUs to train the stages in parallel, *THR* may be more efficient than *Critical*. Nevertheless, if the number of stages increase, the batching optimization *B-Critical* would yield similar performance to *THR* like in Figure 12.

For ASHA, *THR* is the slowest algorithm; *B-Critical* is 1.16x faster. When ASHA dynamically adds new trials, the critical path before trial submission tends to be also a critical path in the new stage tree; however, the number of descendants the stage has changes as new trials are added.

For *Extrapolate*, the algorithm evaluates trials every epoch, creating many small stages. Therefore, the stage batching in *B-Critical* is more effective in *Extrapolate* than other hyper-parameter optimization algorithms.

7 RELATED WORK

Systems for Hyper-parameter tuning. In recent years, there have been hyper-parameter optimization systems [2, 26, 27, 48, 59, 60, 64, 69, 71] which help users to manage their hyper-parameter optimization jobs in distributed environments. Tune [59], for example, is a hyper-parameter optimization system built on top of Ray [65]. 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 Hippo. Other popular trial-based hyper-parameter optimization systems such as Google Vizier [27], NNI [64], Optuna [2], Kubeflow [26], CHOPT [48], HyperDrive [71], and SageMaker [69] 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.

Hippo is distinguished from the existing systems mentioned above in that it is purposefully designed to support tuning hyper-parameters, of which values dynamically change during a trial. Other systems model hyper-parameter optimization workloads as a function of a single configuration input. Alternatively, Hippo captures the multi-step characteristic in some hyper-parameters and extends the traditional view of hyper-parameter from a single input to a sequence of configurations. This approach creates potentially overlapping operations within a study and across multiple studies.

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 [14, 53, 57, 79, 88] fall into the former, the static setting. Liam et al. [57] proposes an algorithm to reduce the resource usage of static data preprocessing pipelines by caching intermediate data. On the contrary, this paper proposes both a data management system and algorithm to reduce the overall amount of computation in dynamic hyper-parameter optimization jobs by merging common stages. Moreover, the technique performs model training sequentially, running individual hyper-parameter optimization trials one by one on a single GPU, whereas Hippo exploits a multi-GPU cluster to run multiple stages in parallel via a system scheduler. Pretzel [53] 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 [14] 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 [88] 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, Hippo's search plan data structure allows the system to dynamically accommodate new trials and identify reusable stage results without running an offline analysis of all trials.

Various big data systems [9, 23, 31, 44, 54, 67] assume the latter, dynamic setting. Nectar [31] enables reusing common computation in DryadLINQ programs within a datacenter. Tachyon [54] 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 optimization algorithms such as ASHA [56] and PBT [41] 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 [58] extends ASHA [56] and supports algorithms similar to ASHA. On the other hand, Hippo aims to support various hyper-parameter optimization algorithms including ASHA [56], SHA [42], PBT [41], and the median-stopping rule [27].

Model merging optimization. Liu et al. [60] propose a *pack* mechanism that trains multiple models as a single batch, which allows a worker to run multiple stages at once. Packing models does not necessarily eliminate redundant computation, though; the help of a stage-based hyper-parameter optimization system like Hippo is still needed to identify common computations and schedule them as a single stage. Hippo's goal of training multiple different models and studies simultaneously is well aligned with that of the pack mechanism.

Integrating model packing into Hippo's workers is straightforward – instead of scheduling a single path of stages to a worker, Hippo's scheduler fetches multiple paths per worker. The workers then pack stages from each path and run the packed stages as a single batch. Such an extension implies an interesting challenge of designing a scheduling policy that considers both resource packing and stage trees. Resource packing is beneficial when there are more trials to execute than the number of GPUs. Unfortunately, the number of queued stages dynamically changes, and thus, the system constantly switches from an overloaded state to an under-utilized state and vice versa. Therefore, to effectively integrate resource sharing with Hippo, the system should pack trials when overloaded and unpack trials when there are idle resources. We leave designing such a policy as future work.

8 CONCLUSION

Hippo is a hyper-parameter optimization system that removes redundant computation for model training 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. Hippo is also applicable to multi-study scenarios. Our evaluations show that Hippo significantly saves GPU-hours and reduces end-to-end training time compared to Ray Tune on multiple models and hyperparameter optimization algorithms.

ACKNOWLEDGMENTS

We thank the reviewers for their valuable comments. This work was supported by FriendliAI Inc.

REFERENCES

[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 G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Savannah, GA, USA) *(OSDI'16)*. USENIX Association, USA, 265–283.

[2] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. 2019. Optuna: A Next-Generation Hyperparameter Optimization Framework. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Anchorage, AK, USA) *(KDD '19)*. Association for Computing Machinery, New York, NY, USA, 2623–2631. https://doi.org/10.1145/3292500.3330701

[3] Dario Amodei, Sundaram Ananthanarayanan, Rishita Anubhai, Jingliang Bai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Qiang Cheng, Guoliang Chen, Jie Chen, Jingdong Chen, Zhijie Chen, Mike Chrzanowski, Adam Coates, Greg Diamos, Ke Ding, Niandong Du, Erich Elsen, Jesse Engel, Weiwei Fang, Linxi Fan, Christopher Fougner, Liang Gao, Caixia Gong, Awni Hannun, Tony Han, Lappi Johannes, Bing Jiang, Cai Ju, Billy Jun, Patrick LeGresley, Libby Lin, Junjie Liu, Yang Liu, Weigao Li, Xiangang Li, Dongpeng Ma, Sharan Narang, Andrew Ng, Sherjil Ozair, Yiping Peng, Ryan Prenger, Sheng Qian, Zongfeng Quan, Jonathan Raiman, Vinay Rao, Sanjeev Satheesh, David Seetapun, Shubho Sengupta, Kavya Srinet, Anuroop Sriram, Haiyuan Tang, Liliang Tang, Chong Wang, Jidong Wang, Kaifu Wang, Yi Wang, Zhijian Wang, Zhiqian Wang, Shuang Wu, Likai Wei, Bo Xiao, Wen Xie, Yan Xie, Dani Yogatama, Bin Yuan, Jun Zhan, and Zhenyao Zhu. 2016. Deep Speech 2 : End-to-End Speech Recognition in English and Mandarin. In *Proceedings of The 33rd International Conference on Machine Learning (Proceedings of Machine Learning Research)*, Maria Florina Balcan and Kilian Q. Weinberger (Eds.), Vol. 48. PMLR, New York, New York, USA, 173–182. https://proceedings.mlr.press/v48/amodei16.html

[4] Elahe Arani, Shabbir Marzban, Andrei Pata, and Bahram Zonooz. 2021. RGPNet: A Real-Time General Purpose Semantic Segmentation. In *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision (WACV)*. 3009–3018.

[5] Atilim Gunes Baydin, Robert Cornish, David Martínez-Rubio, Mark Schmidt, and Frank Wood. 2018. Online Learning Rate Adaptation with Hypergradient Descent. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net. https://openreview.net/forum?id=BkrsAzWAb

[6] Kurian Benoy. 2020. Classifying Flowers with Fastai2. https://www.kaggle.com/kurianbenoy/classifying-flowers-with-fastai2/notebook

[7] Xavier Bouthillier and Gaël Varoquaux. 2020. *Survey of machine-learning experimental methods at NeurIPS2019 and ICLR2020*. Research Report. Inria Saclay Ile de France. https://hal.archives-ouvertes.fr/hal-02447823

[8] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. arXiv:2005.14165 [cs.CL]

[9] Jesús Camacho-Rodríguez, Dario Colazzo, Melanie Herschel, Ioana Manolescu, and Soudip Roy Chowdhury. 2016. Reuse-Based Optimization for Pig Latin. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management* (Indianapolis, Indiana, USA) *(CIKM '16)*. Association for Computing Machinery, New York, NY, USA, 2215–2220. https://doi.org/10.1145/2983323.2983669

[10] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. arXiv:1512.01274 [cs.DC]

[11] François Chollet et al. 2015. Keras. https://github.com/fchollet/keras.

[12] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. 2016. Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs). In *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, Yoshua Bengio and Yann LeCun (Eds.). http://arxiv.org/abs/1511.07289

[13] Federico Colangelo, Federica Battisti, and Alessandro Neri. 2020. Progressive Training Of Convolutional Neural Networks For Acoustic Events Classification. In *28th European Signal Processing Conference, EUSIPCO 2020, Amsterdam, Netherlands, January 18-21, 2021*. IEEE, 26–30. https://doi.org/10.23919/Eusipco47968.2020.9287362

[14] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. 2017. Clipper: A Low-Latency Online Prediction Serving System. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 613–627. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/crankshaw

[15] Ekin Dogus Cubuk, Barret Zoph, Jon Shlens, and Quoc Le. 2020. RandAugment: Practical Automated Data Augmentation with a Reduced Search Space. In *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin (Eds.), Vol. 33. Curran Associates, Inc., 18613–18624. https://proceedings.neurips.cc/paper/2020/file/d85b63ef0ccb114d0a3bb7b7d808028f-Paper.pdf

[16] Henggang Cui, Gregory R. Ganger, and Phillip B. Gibbons. 2018. MLtuner: System Support for Automatic Machine Learning Tuning. arXiv:1803.07445 [cs.LG]

[17] Piali Das, Nikita Ivkin, Tanya Bansal, Laurence Rouesnel, Philip Gautier, Zohar Karnin, Leo Dirac, Lakshmi Ramakrishnan, Andre Perunicic, Iaroslav Shcherbatyi, Wilton Wu, Aida Zolic, Huibin Shen, Amr Ahmed, Fela Winkelmolen, Miroslav Miladinovic, Cedric Archembeau, Alex Tang, Bhaskar Dutt, Patricia Grao, and Kumar Venkateswar. 2020. Amazon SageMaker Autopilot: A White Box AutoML Solution at Scale. In *Proceedings of the Fourth International Workshop on Data Management for End-to-End Machine Learning* (Portland, OR, USA) *(DEEM'20)*. Association for Computing Machinery, New York, NY, USA, Article 2, 7 pages. https://doi.org/10.1145/3399579.3399870

[18] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. ImageNet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*. 248–255. https://doi.org/10.1109/CVPR.2009.5206848

[19] Aditya Devarakonda, Maxim Naumov, and Michael Garland. 2018. AdaBatch: Adaptive Batch Sizes for Training Deep Neural Networks. arXiv:1712.02029 [cs.LG]

[20] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, Jill Burstein, Christy Doran, and Thamar Solorio (Eds.). Association for Computational Linguistics, 4171–4186. https://doi.org/10.18653/v1/n19-1423

[21] Terrance Devries and Graham W. Taylor. 2017. Improved Regularization of Convolutional Neural Networks with Cutout. *arXiv:1708.04552* (2017).

[22] Tobias Domhan, Jost Tobias Springenberg, and Frank Hutter. 2015. Speeding up Automatic Hyperparameter Optimization of Deep Neural Networks by Extrapolation of Learning Curves. In *Proceedings of the 24th International Conference on Artificial Intelligence* (Buenos Aires, Argentina) *(IJCAI'15)*. AAAI Press, 3460–3468.

[23] Iman Elghandour and Ashraf Aboulnaga. 2012. ReStore: Reusing Results of MapReduce Jobs. *Proc. VLDB Endow.* 5, 6 (feb 2012), 586–597. https://doi.org/10.14778/2168651.2168659

[24] Canva Engineering. 2021. Machine learning hyperparameter optimization with Argo. https://canvatechblog.com/machine-learning-hyperparameter-optimization-with-argo-a60d70b1fc8c.

[25] Lex Fridman, Jack Terwilliger, and Benedikt Jenik. 2019. DeepTraffic: Crowdsourced Hyperparameter Tuning of Deep Reinforcement Learning Systems for Multi-Agent Dense Traffic Navigation. arXiv:1801.02805 [cs.NE]

[26] Johnu George, Ce Gao, Richard Liu, Hou Gang Liu, Yuan Tang, Ramdoot Pydipaty, and Amit Kumar Saha. 2020. A Scalable and Cloud-Native Hyperparameter Tuning System. arXiv:2006.02085 [cs.DC]

[27] Daniel Golovin, Benjamin Solnik, Subhodeep Moitra, Greg Kochanski, John Elliot Karro, and D. Sculley (Eds.). 2017. *Google Vizier: A Service for Black-Box Optimization*. http://www.kdd.org/kdd2017/papers/view/google-vizier-a-service-for-black-box-optimization

[28] Linyuan Gong, Di He, Zhuohan Li, Tao Qin, Liwei Wang, and Tieyan Liu. 2019. Efficient Training of BERT by Progressively Stacking. In *Proceedings of the 36th International Conference on Machine Learning (Proceedings of Machine Learning Research)*, Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.), Vol. 97. PMLR, 2337–2346. https://proceedings.mlr.press/v97/gong19a.html

[29] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. 2017. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677* (2017).

[30] Xiaotao Gu, Liyuan Liu, Hongkun Yu, Jing Li, Chen Chen, and Jiawei Han. 2021. On the Transformer Growth for Progressive BERT Training. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Association for Computational Linguistics, Online, 5174–5180. https://doi.org/10.18653/v1/2021.naacl-main.406

[31] Pradeep Kumar Gunda, Lenin Ravindranath, Chandramohan A. Thekkath, Yuan Yu, and Li Zhuang. 2010. Nectar: Automatic Management of Data and Computation in Datacenters. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*. USENIX Association, Vancouver, BC. https://www.usenix.org/conference/osdi10/nectar-automatic-management-data-and-computation-datacenters

[32] Awni Hannun, Carl Case, Jared Casper, Bryan Catanzaro, Greg Diamos, Erich Elsen, Ryan Prenger, Sanjeev Satheesh, Shubho Sengupta, Adam Coates, and

Andrew Y. Ng. 2014. Deep Speech: Scaling up end-to-end speech recognition.

[33] Md Kamrul Hasan, Md Tasnim Jawad, Kazi Nasim Imtiaz Hasan, Sajal Basak Partha, Md Masum Al Masba, Shumit Saha, and Mohammad Ali Moni. 2021. COVID-19 identification from volumetric chest CT scans using a progressively resized 3D-CNN incorporating segmentation, augmentation, and class-rebalancing. *Informatics in medicine unlocked* 26 (2021), 100709. https://doi.org/10.1016/j.imu.2021.100709

[34] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[35] Geoffrey Hinton, Nitish Srivastava, and Kevin Swersky. 2012. Neural networks for machine learning lecture 6a overview of mini-batch gradient descent. *Cited on* 14, 8 (2012), 2. http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf.

[36] Daniel Ho, Eric Liang, Xi Chen, Ion Stoica, and Pieter Abbeel. 2019. Population Based Augmentation: Efficient Learning of Augmentation Policy Schedules. In *Proceedings of the 36th International Conference on Machine Learning (Proceedings of Machine Learning Research)*, Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.), Vol. 97. PMLR, 2731–2741. https://proceedings.mlr.press/v97/ho19b.html

[37] Elad Hoffer, Berry Weinstein, Itay Hubara, Tal Ben-Nun, Torsten Hoefler, and Daniel Soudry. 2019. Mix & Match: training convnets with mixed image sizes for improved accuracy, speed and scale resiliency. arXiv:1908.08986 [cs.CV]

[38] Jeremy Howard. 2018. Training Imagenet in 3 hours for USD 25; and CIFAR10 for USD 0.26. https://www.fast.ai/2018/04/30/dawnbench-fastai/

[39] IAFOSS. 2018. Similarity DenseNet121 [0.805LB] kernel time limit. https://www.kaggle.com/iafoss/similarity-densenet121-0-805lb-kernel-time-limit/notebook

[40] Sergey Ioffe and Christian Szegedy. 2015. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. In *Proceedings of the 32nd International Conference on Machine Learning (Proceedings of Machine Learning Research)*, Francis Bach and David Blei (Eds.), Vol. 37. PMLR, Lille, France, 448–456. https://proceedings.mlr.press/v37/ioffe15.html

[41] Max Jaderberg, Valentin Dalibard, Simon Osindero, Wojciech M. Czarnecki, Jeff Donahue, Ali Razavi, Oriol Vinyals, Tim Green, Iain Dunning, Karen Simonyan, Chrisantha Fernando, and Koray Kavukcuoglu. 2017. Population Based Training of Neural Networks. arXiv:1711.09846 [cs.LG]

[42] Kevin Jamieson and Ameet Talwalkar. 2016. Non-stochastic Best Arm Identification and Hyperparameter Optimization. In *Proceedings of the 19th International Conference on Artificial Intelligence and Statistics (Proceedings of Machine Learning Research)*, Arthur Gretton and Christian C. Robert (Eds.), Vol. 51. PMLR, Cadiz, Spain, 240–248. https://proceedings.mlr.press/v51/jamieson16.html

[43] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional Architecture for Fast Feature Embedding. In *Proceedings of the 22nd ACM International Conference on Multimedia* (Orlando, Florida, USA) *(MM '14)*. Association for Computing Machinery, New York, NY, USA, 675–678. https://doi.org/10.1145/2647868.2654889

[44] Alekh Jindal, Shi Qiao, Hiren Patel, Zhicheng Yin, Jieming Di, Malay Bag, Marc Friedman, Yifung Lin, Konstantinos Karanasos, and Sriram Rao. 2018. Computation Reuse in Analytics Job Service at Microsoft. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 191–203. https://doi.org/10.1145/3183713.3190656

[45] Kiran U Kamath. 2020. fastai MultiLabel Classification using Kfold CV. https://www.kaggle.com/kirankamat/fastai-multilabel-classification-using-kfold-cv/notebook

[46] Tero Karras, Timo Aila, Samuli Laine, and Jaakko Lehtinen. 2018. Progressive Growing of GANs for Improved Quality, Stability, and Variation. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net. https://openreview.net/forum?id=Hk99zCeAb

[47] Chiheon Kim, Saehoon Kim, Jongmin Kim, Donghoon Lee, and Sungwoong Kim. 2021. Automated Learning Rate Scheduler for Large-batch Training. In *8th ICML Workshop on Automated Machine Learning (AutoML)*.

[48] Jinwoong Kim, Minkyu Kim, Heungseok Park, Ernar Kusdavletov, Dongjun Lee, Adrian Kim, Ji-Hoon Kim, Jung-Woo Ha, and Nako Sung. 2018. CHOPT : Automated Hyperparameter Optimization Framework for Cloud-Based Machine Learning Platforms. arXiv:1810.03527 [cs.LG]

[49] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Yoshua Bengio and Yann LeCun (Eds.). http://arxiv.org/abs/1412.6980

[50] Alexander Kolesnikov, Lucas Beyer, Xiaohua Zhai, Joan Puigcerver, Jessica Yung, Sylvain Gelly, and Neil Houlsby. 2020. Big transfer (bit): General visual representation learning. In *Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part V 16*. Springer, 491–507.

[51] Alex Krizhevsky. 2009. Learning Multiple Layers of Features from Tiny Images. *Tech report* (2009).

[52] Lak Lakshmanan and Wenzhe Li. 2018. Hyperparameter tuning on Google Cloud Platform is now faster and smarter. https://cloud.google.com/blog/products/gcp/hyperparameter-tuning-on-google-cloud-platform-is-now-faster-and-smarter.

[53] Yunseong Lee, Alberto Scolari, Byung-Gon Chun, Marco Domenico Santambrogio, Markus Weimer, and Matteo Interlandi. 2018. PRETZEL: Opening the Black Box of Machine Learning Prediction Serving Systems. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 611–626. https://www.usenix.org/conference/osdi18/presentation/lee

[54] Haoyuan Li, Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. 2014. Tachyon: Reliable, Memory Speed Storage for Cluster Computing Frameworks. In *Proceedings of the ACM Symposium on Cloud Computing* (Seattle, WA, USA) *(SOCC '14)*. Association for Computing Machinery, New York, NY, USA, 1–15. https://doi.org/10.1145/2670979.2670985

[55] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. 2017. Hyperband: A novel bandit-based approach to hyperparameter optimization. *The Journal of Machine Learning Research* 18, 1 (2017), 6765–6816.

[56] Liam Li, Kevin Jamieson, Afshin Rostamizadeh, Ekaterina Gonina, Jonathan Ben-tzur, Moritz Hardt, Benjamin Recht, and Ameet Talwalkar. 2020. A System for Massively Parallel Hyperparameter Tuning. In *Proceedings of Machine Learning and Systems*, I. Dhillon, D. Papailiopoulos, and V. Sze (Eds.), Vol. 2. 230–246. https://proceedings.mlsys.org/paper/2020/file/f4b9ec30ad9f68f89b29639786cb62ef-Paper.pdf

[57] Liam Li, Evan Sparks, Kevin Jamieson, and Ameet Talwalkar. 2018. Exploiting Reuse in Pipeline-Aware Hyperparameter Tuning. In *Systems for ML Workshop at NeurIPS*.

[58] Richard Liaw, Romil Bhardwaj, Lisa Dunlap, Yitian Zou, Joseph E. Gonzalez, Ion Stoica, and Alexey Tumanov. 2019. HyperSched: Dynamic Resource Reallocation for Model Development on a Deadline. In *Proceedings of the ACM Symposium on Cloud Computing* (Santa Cruz, CA, USA) *(SoCC '19)*. Association for Computing Machinery, New York, NY, USA, 61–73. https://doi.org/10.1145/3357223.3362719

[59] Richard Liaw, Eric Liang, Robert Nishihara, Philipp Moritz, Joseph E. Gonzalez, and Ion Stoica. 2018. Tune: A Research Platform for Distributed Model Selection and Training. In *ICML AutoML Workshop*.

[60] Rui Liu, Sanjay Krishnan, Aaron J. Elmore, and Michael J. Franklin. 2021. Understanding and Optimizing Packed Neural Network Training for Hyper-Parameter Tuning. In *Proceedings of the Fifth Workshop on Data Management for End-To-End Machine Learning* (Virtual Event, China) *(DEEM '21)*. Association for Computing Machinery, New York, NY, USA, Article 3, 11 pages. https://doi.org/10.1145/3462462.3468880

[61] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. 2020. Themis: Fair and Efficient GPU Cluster Scheduling. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 289–304. https://www.usenix.org/conference/nsdi20/presentation/mahajan

[62] Sam McCandlish, Jared Kaplan, Dario Amodei, and OpenAI Dota Team. 2018. An Empirical Model of Large-Batch Training. arXiv:1812.06162 [cs.LG]

[63] Dirk Merkel. 2014. Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux J.* 2014, 239, Article 2 (March 2014). http://dl.acm.org/citation.cfm?id=2600239.2600241

[64] Microsoft. 2017. Neural Network Intelligence (NNI). https://github.com/Microsoft/nni

[65] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. 2018. Ray: A Distributed Framework for Emerging AI Applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 561–577. https://www.usenix.org/conference/osdi18/presentation/moritz

[66] Anna Novikova. 2019. fast.ai starter with ResNet 50. https://www.kaggle.com/demonplus/fast-ai-starter-with-resnet-50/notebook

[67] Tomasz Nykiel, Michalis Potamias, Chaitanya Mishra, George Kollios, and Nick Koudas. 2010. MRShare: Sharing across Multiple Queries in MapReduce. *Proc. VLDB Endow.* 3, 1–2 (sep 2010), 494–505. https://doi.org/10.14778/1920841.1920906

[68] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett (Eds.). 8024–8035. https://proceedings.neurips.cc/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html

[69] Valerio Perrone, Huibin Shen, Aida Zolic, Iaroslav Shcherbatyi, Amr Ahmed, Tanya Bansal, Michele Donini, Fela Winkelmolen, Rodolphe Jenatton, Jean Baptiste Faddoul, Barbara Pogorzelska, Miroslav Miladinovic, Krishnaram Kenthapadi, Matthias Seeger, and Cédric Archambeau. 2021. *Amazon SageMaker Automatic Model Tuning: Scalable Gradient-Free Optimization.* Association for Computing Machinery, New York, NY, USA, 3463–3471. https://doi.org/10.1145/3447548.3467098

[70] Miguel Pinto. 2019. pneumothorax fastai U-Net. https://www.kaggle.com/mnpinto/pneumothorax-fastai-u-net/notebook

[71] Jeff Rasley, Yuxiong He, Feng Yan, Olatunji Ruwase, and Rodrigo Fonseca. 2017. HyperDrive: Exploring Hyperparameters with POP Scheduling. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference* (Las Vegas, Nevada) *(Middleware '17)*. Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/3135974.3135994

[72] Inc Red Hat. 2020. GlusterFS. https://www.gluster.org/

[73] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. 2016. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition.* 779–788.

[74] Frank Seide and Amit Agarwal. 2016. CNTK: Microsoft's Open-Source Deep-Learning Toolkit. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (San Francisco, California, USA) *(KDD '16)*. Association for Computing Machinery, New York, NY, USA, 2135. https://doi.org/10.1145/2939672.2945397

[75] Leslie N. Smith. 2017. Cyclical Learning Rates for Training Neural Networks. In *2017 IEEE Winter Conference on Applications of Computer Vision (WACV)*. 464–472. https://doi.org/10.1109/WACV.2017.58

[76] Leslie N. Smith. 2018. A disciplined approach to neural network hyper-parameters: Part 1 – learning rate, batch size, momentum, and weight decay. arXiv:1803.09820 [cs.LG]

[77] Leslie N. Smith and Nicholay Topin. 2018. Super-Convergence: Very Fast Training of Neural Networks Using Large Learning Rates. arXiv:1708.07120 [cs.LG]

[78] Samuel L. Smith, Pieter-Jan Kindermans, Chris Ying, and Quoc V. Le. 2018. Don't Decay the Learning Rate, Increase the Batch Size. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net. https://openreview.net/forum?id=B1Yy1BxCZ

[79] Evan R. Sparks, Shivaram Venkataraman, Tomer Kaftan, Michael J. Franklin, and Benjamin Recht. 2017. KeystoneML: Optimizing Pipelines for Large-Scale Advanced Analytics. In *33rd IEEE International Conference on Data Engineering, ICDE 2017, San Diego, CA, USA, April 19-22, 2017*. IEEE Computer Society, 535–546. https://doi.org/10.1109/ICDE.2017.109

[80] Danny Stoll, Jörg K. H. Franke, Diane Wagner, Simon Selg, and Frank Hutter. 2020. Hyperparameter Transfer Across Developer Adjustments. arXiv:2010.13117 [cs.LG]

[81] Nako Sung, Minkyu Kim, Hyunwoo Jo, Youngil Yang, Jinwoong Kim, Leonard Lausen, Youngkwan Kim, Gayoung Lee, Donghyun Kwak, Jung-Woo Ha, and Sunghun Kim. 2017. NSML: A Machine Learning Platform That Enables You to Focus on Your Models. In *NIPS Workshop on Machine Learning Systems (LearningSys)*.

[82] Mingxing Tan and Quoc Le. 2021. EfficientNetV2: Smaller Models and Faster Training. In *Proceedings of the 38th International Conference on Machine Learning (Proceedings of Machine Learning Research)*, Marina Meila and Tong Zhang (Eds.), Vol. 139. PMLR, 10096–10106. https://proceedings.mlr.press/v139/tan21a.html

[83] Eclipse Deeplearning4j Development Team. 2016. ND4J: Fast, Scientific and Numerical Computing for the JVM. https://github.com/eclipse/deeplearning4j

[84] Seiya Tokui, Ryosuke Okuta, Takuya Akiba, Yusuke Niitani, Toru Ogawa, Shunta Saito, Shuji Suzuki, Kota Uenishi, Brian Vogel, and Hiroyuki Yamazaki Vincent. 2019. Chainer: A Deep Learning Framework for Accelerating the Research Cycle. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD 2019, Anchorage, AK, USA, August 4-8, 2019*, Ankur Teredesai, Vipin Kumar, Ying Li, Rómer Rosales, Evimaria Terzi, and George Karypis (Eds.). ACM, 2002–2011. https://doi.org/10.1145/3292500.3330756

[85] Qianwen Wang, Yao Ming, Zhihua Jin, Qiaomu Shen, Dongyu Liu, Micah J. Smith, Kalyan Veeramachaneni, and Huamin Qu. 2019. *ATMSeer: Increasing Transparency and Controllability in Automated Machine Learning.* Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/3290605.3300911

[86] Tao Wei, Changhu Wang, Yong Rui, and Chang Wen Chen. 2016. Network Morphism. In *Proceedings of The 33rd International Conference on Machine Learning (Proceedings of Machine Learning Research)*, Maria Florina Balcan and Kilian Q. Weinberger (Eds.), Vol. 48. PMLR, New York, New York, USA, 564–572. https://proceedings.mlr.press/v48/wei16.html

[87] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. 2016. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144* (2016).

[88] Doris Xin, Stephen Macke, Litian Ma, Jialin Liu, Shuchen Song, and Aditya G. Parameswaran. 2018. Helix: Holistic Optimization for Accelerating Iterative Machine Learning. *Proc. VLDB Endow.* 12, 4 (2018), 446–460. https://doi.org/10.14778/3297753.3297763

[89] Matthew D Zeiler. 2012. Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701* (2012).

[90] Jian Zhang and Ioannis Mitliagkas. 2019. YellowFin and the Art of Momentum Tuning. In *Proceedings of Machine Learning and Systems*, A. Talwalkar, V. Smith, and M. Zaharia (Eds.), Vol. 1. 289–308. https://proceedings.mlsys.org/paper/2019/file/b3e3e393c77e35a4a3f3cbd1e429b5dc-Paper.pdf