ForestTune: Resource-Aware Hyperparameter Schedule Discovery as a Tree Search

Jacob Garby
Philippas Tsigas
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden
{garby,tsigas}@chalmers.se

Abstract

Hyperparameter tuning is a critical component of modern machine learning pipelines, as the performance and efficiency of many data processing workloads depend heavily on the choice of hyperparameters. While most traditional tuning methods in the literature assume static configurations, many hyperparameters benefit from being adapted during execution.

While existing population-based tuning methods such as *PBT* and *PB2* implicitly exploit scheduling effects, they consider a flat population of independent trials. As a result, they lack principled mechanisms for branching executions, consideration of system-level parameters for scheduling, and re-evaluation of configurations under stochasticity.

We propose ForestTune, which addresses these gaps by modelling the tuning process as a search over an ever-growing forest of executions. In ForestTune, schedules correspond to paths through trees and are extended by sampling new configurations from per-node Gaussian process surrogates. This structure enables systematic exploration, adaptive scheduling, and principled reuse of information across related executions.

We evaluate ForestTune on two popular deep learning workloads, spanning image classification (CIFAR-100) and graph neural network training (OGB-Arxiv). Our results show that ForestTune outperforms state-of-the-art methods due to its efficient exploration of hyperparameter schedules and its resource-aware trial scheduling, reducing tuning time to comparable accuracy by up to 71%, while providing an increase in final accuracy of up to 90%.

CCS Concepts

• **Computing methodologies** → **Machine learning**; **Parallel algorithms**.

Keywords

Hyperparameter Tuning, Machine Learning, Asynchrony, Parallel Programming

1 Introduction

Modern deep learning workloads place extreme demands on training infrastructure, requiring systems that can efficiently utilise parallel hardware while preserving optimisation quality. To meet these demands, training is routinely executed on either multi-core CPUs, GPUs, or distributed clusters, often employing asynchronous parallelism to maximise throughput. While such techniques improve hardware utilisation, they introduce complex trade-offs between system performance and algorithmic convergence, governed by a growing set of interacting configuration parameters.

Many of these parameters — such as the degree of parallelism and synchronisation strategy — directly affect optimisation dynamics and final model accuracy. At the same time, learning algorithms themselves expose additional parameters, including learning rate, batch size, and momentum. Collectively, these hyperparameters span both the algorithmic and system layers, and their values critically determine training time and solution quality [4, 18, 28–30]. Choosing appropriate hyperparameters is therefore a central challenge for efficient deep learning.

Traditional hyperparameter tuning methods work by attempting to optimise a black-box function representing the performance of a given hyperparameter configuration on the target workload. Fully evaluating one configuration takes a significant amount of time. It is in most cases intractable to consider all, or even most, configurations. Multi-fidelity approaches mitigate this cost by allocating more resources to promising configurations [10, 18], but still assume that hyperparameters remain fixed throughout execution.

Certain hyperparameters are conducive to being adjusted dynamically throughout an execution. This kind of mutability can be beneficial to training performance [5, 31]. Unfortunately, extending tuning from static configurations to time-varying schedules greatly increases search complexity. Prior work has addressed this challenge only in limited forms, typically focusing on individual hyperparameters such as *learning rate* [20, 32] and *temperature* (for simulated annealing) [17]. These approaches are highly specialised to specific hyperparameters and difficult to generalise.

More general strategies, such as *population-based tuning* [16, 24], implicitly explore hyperparameter schedules through evolutionary mechanisms. However, these methods tend to assume a stationary optimisation landscape and offer limited support for system-level hyperparameters, whose effects depend strongly on runtime conditions such as contention and staleness.

Besides hyperparameter tuning, the problem of producing deep learning infrastructure and systems which are capable of handling enormous quantities of training data in a timely fashion has been attacked from all sides. It has long been so that the amount of computation power of any single processor is insufficient to deal with the scale of modern training tasks, therefore alternate compute paradigms are considered a necessity. Whether the execution is performed on a GPU [22] (or a cluster of several [23]), on a multi-core CPU [2, 5, 11], distributed over racks of connected servers [2], among geographically distributed compute-nodes [9],

or even within a dedicated ASIC [6], the general principle is that training can be made faster by spreading the load across separate compute devices; the work can hence be carried out in parallel.

In this work, we focus on the shared-memory multi-core CPU setting, where fine-grained asynchronous parallelism is commonly used to improve throughput (e.g. Hogwild! [27]). While GPUs are indispensable for large dense models, a growing body of work has shown that modern multi-core CPUs remain a competitive and often preferable platform for many training workloads, particularly when using asynchronous optimisation methods and moderate model sizes. In particular, Ma et al. [21] demonstrate that asynchronous SGD on CPUs can outperform GPU-based approaches in time-to-convergence across several models, even when GPUs achieve higher raw throughput. This behaviour is consistent with earlier findings on lock-free and asynchronous optimisation on shared-memory CPUs [27], and continues to motivate CPU-centric training in settings where flexibility, resource sharing, and cost efficiency are primary concerns. Prior studies have shown that blindly increasing parallelism in such systems can be counterproductive: excessive concurrency may increase training time and degrade model accuracy (as illustrated in Figure 1, discussed in Section 2.1). Moreover, the optimal degree of parallelism can change significantly during execution [5], reflecting a non-stationary optimisation landscape shaped by both algorithmic progress and system dynamics.

We address this challenge by introducing ForestTune, a hyperparameter tuning framework designed for deep learning systems with mutable, system-level, and algorithmic hyperparameters. ForestTune explicitly optimises time-varying hyperparameter schedules while accounting for changes in the optimisation landscape throughout execution. It organises tuning trials as a dynamically growing forest of execution trees, allowing promising executions to branch and explore alternative schedules without restarting training from scratch. System resource parameters are treated as first-class hyperparameters, enabling efficient trial scheduling and reuse of intermediate execution state.

To the best of our knowledge, ForestTune is the first framework that (i) explicitly tunes general hyperparameter schedules under a non-stationary optimisation landscape, and (ii) integrates system-level resource parameters into the tuning process in a way that supports efficient execution and comparison of trials.

The remainder of this paper is organised as follows. Section 2 motivates the need for mutable hyperparameters and discusses limitations of existing tuning frameworks. Section 2.4 formalises the tuning problem and introduces the notation used throughout. Section 3 presents ForestTune and its design rationale, followed by implementation considerations in Section 4. We evaluate ForestTune against state-of-the-art baselines in Section 5. Section 6 discusses related work, followed by a conclusion in Section 7.

2 Problem and Challenges

2.1 Mutable Hyperparameters

We distinguish between mutable hyperparameters, whose values may change during a single execution, and immutable hyperparameters, which are fixed at the outset. Mutable hyperparameters include not only algorithmic parameters such as learning rate, batch size, and momentum, but also system-level parameters such as the

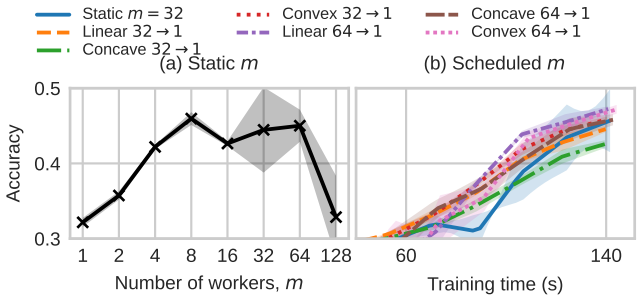


Figure 1: The effect of static and scheduled m on `Hogwild!` training performance for the *GNN* workload.

number of parallel workers and synchronisation semantics [5, 31]. Structural choices like the number of hidden layers, kernel sizes, dataset augmentation, etc., tend to be immutable, since adjusting them mid-run is impossible. Prior work has explored runtime adaptation of individual system parameters (*Pollux* [26] adaptively scales worker count), but these adjustments are built into a specific training scheme, rather than treated as black-box parameters which could be optimised by a general tuner.

It's already well-established in literature that adjusting certain hyperparameters during training can improve performance: learning rate schedules are ubiquitous [20], and at-runtime mutation of algorithmic parameters comes as a side-effect of population-based methods such as *PBT* [16] and PB2 [24]. Such tuners tend to leave *system* hyperparameters fixed, though, especially when those parameters pertain to resource demand. We illustrate this with a canonical example: the number of asynchronous workers used for `Hogwild!` SGD [27].

Number of Workers. `Hogwild!` parallelises SGD across CPU cores by allowing m workers to update shared model parameters without synchronisation or locking, which removes a considerable amount of overhead. However, overlapping updates introduce noisy gradients which, beyond a certain worker count, outweigh the gain in throughput and slow down convergence. The optimal worker count therefore is not the maximum available on the system, and it varies during training. In general, a high m is beneficial early when stability is less important than throughput, while a lower m works better later on for convergence.

In Figure 1, we highlight the importance of correctly scheduling the number of asynchronous workers when running SGD with `Hogwild!` to train a *graph neural network* (GNN) on the popular `OGB-Arxiv` dataset [15]. Each of these executions runs for 140 seconds. In the left-hand pane we test the performance when holding the number of worker threads, m, at different constant values $1 \leq m \leq 128$, and in the right-hand pane we plot the learning trajectory of various *schedules* of m, including the static $m = 32$ for comparison. The "Linear" schedules decrease m between two bounds linearly, whereas the "Concave" schedules decrease faster at the beginning and slower near the end, proportional to $-\sqrt{\text{time}}$.

Figure 1 illustrates three key observations. First, the number of asynchronous workers has a substantial impact on both convergence speed and final accuracy, with excessive parallelism amplifying asynchrony-induced noise. Second, appropriately scheduling m over time can outperform any static choice, while also improving

convergence stability. Finally, both the optimal value and schedule for m depend on training phase and workload characteristics, making them difficult to determine a priori.

2.2 Trial Continuation and Robust Re-Evaluation

In many existing hyperparameter tuning frameworks, candidate configurations are evaluated as independent, linear executions that begin from a common initial state and run to completion. They make the implicit assumption that evaluating a given hyperparameter configuration yields a deterministic outcome. In reality, many workloads training and evaluation functions are non-deterministic, and several evaluations of the same trial can lead to substantially different outcomes. This can be for reasons such as the random shuffle of data [4], intentional regularisation [12], or the particular interleavings of asynchronous workers, which depend on runtime scheduling and resource contention [7, 27].

A tuner that relies on a single observation per configuration is therefore vulnerable to noise and may be misled by an unlucky draw. The effect of such stochasticity can be dealt with by re-evaluating certain configurations from the same initial state [3]. However, re-evaluation involves restarting a trial from scratch, which is usually prohibitively expensive. In contrast, if executions could be resumed from intermediate states, re-evaluation could be performed at substantially lower cost, enabling more efficient tuning.

2.3 Unbounded Execution Budget

Many state-of-the-art hyperparameter tuning methods, including Hyperband [18] and BOHB [10], work by orchestrating individual trials, where each trial determines the performance of one certain set of hyperparameters – a configuration. Although the tuning problem is construed as one of finding the best configuration, the user's actual goal is often just to find the best *result* given the best configuration, i.e. the best model parameters. It is therefore important to eventually execute promising configurations all the way until completion, i.e. when the model quality stops increasing due to, for instance, the training algorithm reaching a (local) minimum. However, this is not often trivial. In Hyperband-family algorithms (including BOHB), the structure of the "brackets" which make up the tuning process are predetermined. A single trial can never use a higher budget than the maximum budget (chosen by the user). If the maximum budget is too low, an optimal model will never be reached because the training algorithm simply will not have the chance to run for long enough. On the other hand, if it is too *high*, time spent furthering executions which have already converged will result in wasted time; the excessive budget would ideally have been spent evaluating other configurations instead.

We propose that a more useful and adaptive hyperparameter tuning algorithm would not impose a decision of maximum budget on the user, since it cannot be known beforehand how long is needed to reach convergence. *Some* specification of budget is nevertheless necessary, as in general it refers to an arbitrary resource and the tuning algorithm requires a starting point, but a fixed maximum requires too much foreknowledge about the workload to tune.

2.4 Problem Formulation

Hyperparameter tuning is traditionally posed as the search for a single configuration x minimising a black-box objective $f(x)$ over a search space $\mathcal{X}$ [29]. In this section, we develop existing formulation for hyperparameter tuning, resulting in the mutable-hyperparameter problem with resource-varying trials which ForestTune addresses.

Configuration Space. As in existing literature, we denote the configuration space $\mathcal{X}$ from which candidate hyperparameter configurations $x \in \mathcal{X}$ may be drawn. A configuration parametrises the execution of a given workload. In contrast to most prior formulations, we compose a configuration of a tuple $x = (x_A, x_S)$:

- the *algorithmic* component, $x_A \in \mathcal{X}_A$, represents parameters of the optimisation procedure itself, for example learning rate, momentum, and batch size;
- the *system* component, $x_S \in \mathcal{X}_S$ contains system runtime parameters that affect *how* the execution runs, for example the number of worker threads, the degree of asynchrony, the synchronisation interval.

Additionally, and again in departure from most existing formulations, each configuration requires a certain demand on resources, $x_R \in \mathbb{R}^M_{\geq 0}$, a vector of M resource types (CPU cores, memory, etc.) that tends to be a function of x_S. We then describe the total amount of resources available to concurrent trials as the vector $\mathbf{R}_{\text{sys}} \in \mathbb{R}^M$.

State and Training Chunks. The result of training is summarised by a checkpoint state $\theta \in \Theta$ (model parameters, optimiser state, etc.). One *chunk* of training of wall-clock duration b is captured by a stochastic training operator

$$\theta' = f_{\text{train}}(\theta,\, x,\, b,\, \xi(x)), \tag{1}$$

where $\xi(x)$ is the stochasticity present during the chunk (due to e.g. data shuffling and the non-determinism of concurrent worker interleavings). A separate evaluation function $f_{\text{eval}} : \Theta \to \mathbb{R}$ produces a scalar which we wish to optimise, e.g. validation accuracy. Depending on semantics, optimisation can equally be formulated as minimising or maximising this value, so for consistency we consider the minimising case.

This decomposition of $\mathcal{X}$ exposes two correlations that previous hyperparameter tuning formulations do not address explicitly.

Throughput. The amount of useful work performed in a chunk of fixed wall-clock duration b is a function of x_S. Under `Hogwild!`, for example, the effective rate of gradient updates per second is sublinear in the number of worker threads and plateaus (or even degrades) past a certain number [5, 27].

Noise. The variance of ξ, and therefore of $f_{\text{eval}}(\theta')$, may depend on x. For example, more workers in an asynchronous setting increases the expected *staleness* and raises the variance of stochastic gradients, even at a fixed number of update steps [5, 11].

Traditional and Multi-Fidelity Tuning. The standard hyperparameter tuning problem is, given a fixed budget b, discovering

$$x^\star = \operatorname*{argmin}_{x \in \mathcal{X}} \; \mathbb{E}_\xi[f_{\text{eval}}(f_{\text{train}}(\theta_0, x, b, \xi))]\,. \tag{2}$$

Importantly, the configuration x is constant throughout a given trial. Multi-fidelity methods such as Hyperband [18] and BOHB [10]

use a surrogate function $g(x, b)$ which models the expected performance of evaluating configuration x to a budget b. The budget constrains, epochs, samples, or some other proxy for work, and there is an implicit assumption made by such tuners that budget maps uniformly to wall-clock time. This assumption is relevant for running concurrent trials, as discrepancy in wall-clock time between trials of the same budget would result in idle time for some workers.

Schedule-Based Tuning. As discussed in Section 2.1, allowing for hyperparameters to vary throughout an execution can improve performance. A more general formulation treats x instead as a sequence. Let $\vec{x} = (x^{(1)}, \ldots, x^{(K)}) \in \mathcal{X}^K$ be a schedule of configurations, $\vec{b} = (b^{(1)}, \ldots, b^{(K)})$ the corresponding chunk durations, and let $\theta^{(k)}$ be the state reached after applying chunks $1, \ldots, k$. The objective becomes

$$\vec{x}^\star = \underset{\vec{x},\vec{b}}{\operatorname{argmin}} \ \mathbb{E}\left[f_{\text{eval}}\left(\theta^{(K)}\right)\right], \theta^{(k)} = f_{\text{train}}\left(\theta^{(k-1)}, x^{(k)}, b^{(k)}, \xi^{(k)}\right) \tag{3}$$

Population-Based Tuning (*PBT*) [16] and *PB2* [24] implicitly optimise Equation (3): a population of N agents trains in parallel, and at periodic synchronisation points poorly-performing agents copy the state of stronger ones (*exploitation*) and perturb their configuration (*exploration*). *PBT* and *PB2* retain the implicit assumption inherited from Equation (2): a chunk's wall-clock cost is independent of $x^{(k)}$.

PB2 is able to exploit promising areas of the search space when generating new configurations by maintaining a Gaussian process surrogate function, similar to Hyperband's sampling approach. However, the quality landscape of the configuration space depends on the parent state θ that a given chunk starts from, and is therefore *not stationary*. A single global surrogate fit over all observations averages away this structure, and can lead to producing less suitable configurations.

Concurrent Trials Under a Resource Constraint. Once configurations explicitly include a system component x_S, resource-awareness becomes important to efficiently run trials concurrently. We model this as a schedule of trials, as follows. Let each trial i be started at wall-clock time a_i, run for duration b_i, and demand resources $x_R^{(i)}$. Any valid schedule of trials must satisfy a resource-sharing constraint at every moment:

$$\forall t \in [0, T] : \sum_{i\,:\,a_i \le t < a_i + b_i} x_R^{(i)} \ \le \ \mathbf{R}_{\text{sys}}, \tag{4}$$

where T is the total time the tuning runs for. In order to maximise throughput (budget evaluated per wall-clock time), it is desired to be utilising as much as possible of the system resources, $\mathbf{R}$, at all times. A prudent choice of which trials to run concurrently is necessary to do so. Due to x_S, trials require runtime-varying resources, and therefore these scheduling decisions must be made dynamically.

None of the prior formulations consider this constraint, and instead schedule trials uniformly. They are therefore unable to maximally utilise the underlying resources.

3 Our Method

In this section, we introduce ForestTune, a system for online hyperparameter tuning that constructs and explores hyperparameter schedules during execution, rather than selecting a single static configuration. ForestTune is designed as a runtime mechanism that interleaves tuning decisions with execution, allowing hyperparameters and resource allocations to evolve in response to observed behaviour.

ForestTune is motivated by goals common to prior work, including population-based methods such as *PBT* [16] and *PB2* [24], and multi-fidelity approaches like *Hyperband* [18] and *BOHB* [10]. However, it departs from these methods in four fundamental ways, each addressing a limitation identified in Section 2.

(1) Rather than maintaining a flat population of trials, ForestTune represents the tuning process as a *forest*, where each node corresponds to an execution that resumes from a parent checkpoint. This branching structure enables efficient exploration of multiple related hyperparameter schedules, as well as a single promising trajectory to be explored under multiple stochastic realisations, instead of being prematurely discarded due to unlucky randomness.
(2) In contrast to *PB2*'s single global surrogate model — which, particularly late in a run, is dominated by samples collected at earlier and less relevant stages — each node in ForestTune maintains its own Gaussian process model. This local surrogate is used for Bayesian optimisation when spawning new branches from that node, allowing the method to capture optimisation landscapes that vary across different stages of an execution.
(3) ForestTune treats system resources, such as thread count, as *first-class tunable hyperparameters*. By explicitly scheduling trials according to their resource requirements, the method avoids the over-subscription that arises when the cost of evaluating configurations is assumed to be constant.
(4) Unlike *Hyperband*-style methods [10, 18], ForestTune imposes no fixed upper limit on the trial budget. This reduces the sensitivity of the method to configuration choices made before the tuning run begins and allows progressively longer trials to be explored indefinitely.

Together, these design choices allow ForestTune to adapt both hyperparameters and resource usage dynamically over the course of training, while retaining the ability to revisit and refine promising trajectories as new information becomes available.

We next formalise the underlying concepts and introduce the notation used throughout the remainder of the paper.

3.1 Preliminaries

Where prior hyperparameter tuning methods model trials as independent, linear executions, ForestTune represents tuning work as a dynamically constructed forest of execution fragments. Each node in this forest corresponds to a *bounded chunk* of execution (see Section 2.4) of the target workload and captures both execution state and tuning metadata required for further exploration. A *node* is conceptually a tuple with the following fields:

Forest Structure and Notation. ForestTune organises nodes into a directed forest F. For a node $m \in F$, we use the following notation (summarised in Table 1):

- m_D: the depth of node m, defined as the length of the path from a root node to m;

structure TuneNode
- $\mathbf{x} : \mathcal{X}$ *Hyperparameter configuration.*
- $\theta : \Theta$ *Checkpoint state; type Θ is workload-dependent.*
- $y : \mathbb{R} \mid \text{null}$ *Node performance (i.e. result of f_{eval}); null if not evaluated.*
- $\tilde{f} : \mathcal{X} \to \mathbb{R}$ *Local Bayesian optimisation model.*
- $b : \mathbb{R}$ *Incremental budget for this node.*

- m_{children}: the set of child nodes of m;
- m_θ: the checkpoint state stored at node m.

Execution Semantics. A node conceptually represents the (potentially pending, if not yet processed) end result of applying the workload's training process to the checkpoint state reached by its parent node (or from a freshly initialised model in the case of root nodes). Once processed, each node furthers the execution by a specific amount of budget, dependent on its depth, m_D: nodes deeper in the forest represent a longer training budget.

A path from a root node to any other node represents a single linear execution under a certain hyperparameter schedule; the hyperparameter configuration changes only from node-to-node. Where a node m branches into multiple children, each of the children resume execution from the same checkpoint state m_θ but with different hyperparameter configurations. This branching mechanism enables ForestTune to explore multiple alternative continuations of the same partially executed workload, and is the key mechanism by which different hyperparameter *schedules* are evaluated.

Any subtree of the forest therefore represents a *family* of executions: multiple executions of the same workload that share an identical prefix but diverge in their subsequent hyperparameter schedules. By reusing checkpointed states, ForestTune avoids repeatedly executing identical prefixes from scratch, enabling efficient exploration of similar schedules with minimal redundant computation.

Overall, ForestTune considers a hyperparameter tuning problem as the problem of building up such a forest over time, involving both choosing *which* nodes to create (and hence which hyperparameter schedules to trial), as well as scheduling *when* each node should be evaluated. This design allows it to balance exploration and exploitation while efficiently utilising available parallelism.

In the remainder of this section, we describe the ForestTune algorithm in increasing level of detail.

3.2 Algorithm Overview

Our design separates two concerns: the *policy* of deciding which configurations to evaluate next, and the *scheduling mechanism* of efficiently evaluating them in parallel across workers within a fixed set of system resources. The target workload is treated as a black box, exposed only through a training function f_{train} and an evaluation function f_{eval} (Algorithm 1). These are the same operators introduced in Section 2.4: we simply abbreviate the chunk operator of Equation (1) as $f_{\text{train}}(m, b)$, since a node m already carries both the starting state θ and configuration x (as m_θ and m_x), and the per-chunk stochasticity ξ is drawn implicitly each time a node is trained. ForestTune is driven by a scheduler with two entry points, UpdateSchedule and FinishSchedule, invoked repeatedly by the main loop (Algorithm 2). `UpdateSchedule` constructs

Table 1: Summary of notation

Notation	Description
Spaces and workload functions	
$\mathcal{X}$	Hyperparameter configuration space
Θ	Space of checkpoint states (workload-dependent)
$f_{\text{train}}(m, b)$	Trains node m for budget b, returning a new state $\in \Theta$ (see Equation (1))
$f_{\text{eval}}(\theta)$	Evaluates a state, e.g. validation accuracy
Forest and TuneNode notation	
F	The tune-forest
F_D	All nodes at depth D
m_x	Node's hyperparameter configuration ($\in \mathcal{X}$)
m_θ	Node's checkpoint state ($\in \Theta$)
m_y	Node performance, or null if unevaluated
$m_{\tilde{f}}$	Node's local Bayesian-optimisation surrogate
m_R	Node's system-resource requirement
m_D	Depth of node m
m_{children}	Children of node m
$\tilde{f}_{\text{root}}$	Root Bayesian-optimisation surrogate

the next *schedule* — a set of `TuneNodes` to be processed concurrently. `FinishSchedule` synchronises workers and incorporates their individual training results into the surrogate models that guide subsequent configuration decisions.

In the context of resource-aware trial scheduling, a *schedule* refers to a set of TuneNodes which ForestTune has decided should be evaluated concurrently. As will become apparent when looking at the scheduler, an entire schedule is always created such that the system has sufficient resources to process it entirely in parallel; the workload itself defines the specific operation associated with processing the schedule.

A ForestTune run is divided into iterations called *growth seasons*. At the start of each season, ForestTune resets the depth counter $D \leftarrow 0$ and creates T new root nodes. The season then performs a depth-wise pass through the forest. At each depth D, the system (i) creates new nodes at that depth by branching from selected nodes at depth $D-1$, and (ii) evaluates all nodes at depth D via one or more resource-feasible schedules. The per-node training budget increases with depth, so deeper nodes correspond to longer cumulative executions.

The number of nodes created at each iteration of the upward pass through the forest is controlled by successive division of the number of new TuneNodes to produce: fewer are made as D increases. Conversely, the budget for deeper nodes increases deeper into the forest. A season ends when successive division results in there being no more nodes to create at the current depth, and since a number of new TuneNodes are created at the beginning of each season, subsequent seasons increase in depth monotonically. There is therefore no upper limit to the depth that can be reached (and therefore no limit to how long ForestTune can run while still trialling longer executions); this is one of the motivating factors behind ForestTune's design in contrast to methods such as *BOHB* [10] which impose an important decision on selecting a suitable maximum budget.

In Figure 2 we illustrate the behaviour of ForestTune. The diagram represents the resultant search forest after two *growth seasons*

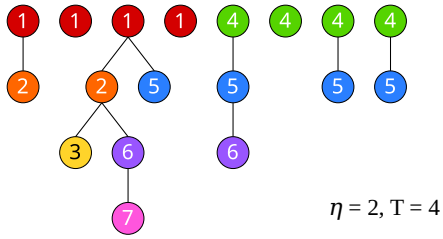


Figure 2: The result of two seasons of ForestTune's tree search based on successive division, with $\eta = 2, T = 4$. Each circle depicts a TuneNode, and the numbers show the order in which they are created.

of an example execution with $\eta = 2, T = 4$; these parameters are described in more detail in Section 3.4. Each circle represents one TuneNode, labelled with a number representing the growth step at which it was created; they are also shaded with a colour representative of the same. First, $B := T = 4$ root nodes are created (A, see Algorithm 1). Once they have all been scheduled and evaluated, the approximately best performing $B := B \div \eta = 2$ nodes in F_D are branched from due to `GrowTrees` (B). This process is repeated once, after which point $B = 0$ causing the season to end. The second season, beginning from iteration 4, plays out in much the same way, except now the number of branches made after the first iteration is $|F_D| \div \eta = 4$. Notice that during the second season, nodes which were created in the first season are branched from for a second time. This happens when an older node's performance is still strong relative to its cousin nodes. Fundamentally, it is this re-branching that allows ForestTune to consider different sequences of configurations.

Scheduling is performed on a depth-by-depth basis. It is necessary that all nodes at depth D are evaluated before branching and beginning to evaluate those at depth $D + 1$ because the information required to decide from which nodes at depth D to branch is dependent on knowledge of the performance of all nodes at that depth. It is for this reason that seasons advance strictly depth-wise.

`UpdateSchedule` considers every node in F_D, i.e. every node at depth D, and if every one of them has already been evaluated, we are then ready to move on to the next iteration. This is the point at which the forest growth procedure, which is the heart of our tuning mechanism, is invoked (B). `UpdateSchedule` then, in either case, performs a system resource-aware scheduling of nodes $\in F_D$ that are yet to be evaluated. As can be seen in Algorithm 2, `UpdateSchedule` is invoked just before calling the workers to perform their workload-specific task, as this way each will see their assigned work once they begin.

The function `PackSchedule` (described in more detail in Section 3.3) performs an efficient bin-packing, determining which nodes should be evaluated next by the computation medium (e.g. CPU threads). After returning, `schedule` contains a list of TuneNodes which can all be executed in parallel in accordance with $\mathbf{R}_{\text{sys}}$. The schedule is read and each node within is processed in parallel (D). `FinishSchedule`. The result of processing all nodes in `schedule` is that each m_θ is now updated to represent the new model state. We then compute the performance, m_y for each; the evaluation function $f_{\text{eval}}(\theta)$ tends, for deep learning problems, to

Algorithm 1 ForestTune's interface

Constants:
Require: $f_{\text{eval}}(\theta) : \Theta \to \mathbb{R}$ *Evaluate a node state, e.g. Accuracy*
Require: $f_{\text{train}}(m, b) : \text{TuneNode} \times \mathbb{R} \to \Theta$ *Processes a node, e.g. by SGD, for budget b*
Require: $\mathbf{R}_{\text{sys}} : \mathbb{R}^M$ *System resource limit vector*
Require: $\eta : \mathbb{R}$ *Successive division factor*
Require: $T : \mathbb{N}^+$ *Number of new roots per season*

Require: `F` : Graph(TuneNode) *The tune-forest itself*
Require: $D : \mathbb{N}$ *Depth currently being considered*
Require: $B : \mathbb{N}_0$ *The number of branches to make next*
Require: `schedule` : [TuneNode] *TuneNodes to evaluate next*
Require: $b : \mathbb{R}$ *Incremental budget for current schedule*
Require: $\tilde{f}_{\text{root}} : \mathcal{X} \to \mathbb{R}$ *Root B.O. surrogate*

function Initialise()
  `F` ← null
  $D \leftarrow 0$
  `schedule` ← []
  $b \leftarrow b_{\min}$
  for $i = 1, \ldots, T$ **do**
    NewRootNode() (A)

function UpdateSchedule()
  if $\forall m \in F_D \mid m_y \neq \text{null}$ **then**
    $B \leftarrow \text{GrowTrees}(B)$ (B)
  PackSchedule($\mathbf{R}_{\text{sys}}$)

function FinishSchedule()
  for all $m \in$ `schedule` **in parallel do** (C)
    $m_y \leftarrow f_{\text{eval}}(m_\theta)$ *Evaluate the newly trained node state.*
    if m is a root **then**
      $\tilde{f}_{\text{root}}$.report(m_x, m_y) *Update root surrogate.*
    else
      $(m_{\text{parent}})_{\tilde{f}}$.report$(m_x, m_y)$ *Update parent's surrogate.*

Algorithm 2 Invocation

This shows how ForestTune is invoked to perform a tuning task. f_{train} and f_{eval} together define the objective.
Initialise() *Create first roots*
UpdateSchedule() *Generate initial schedule*
while ¬ Terminate **do** *Stop on wall-clock / budget / season cap*
  for all $m \in$ `schedule` **in parallel do** (D)
    $m_\theta \leftarrow f_{\text{train}}(m, b)$
    $m_y \leftarrow f_{\text{eval}}(m_\theta)$
  FinishSchedule() *Report results into $\tilde{f}_{root}$ / parents*
  UpdateSchedule()
return $\arg\max_{m \in F} m_y$ *Best evaluated node*

compute the validation accuracy given model θ. So that this result may influence the selection of new configurations in subsequent iterations, each node's parent's Bayesian optimisation surrogate model is updated with the newly evaluated configuration-performance pair as a sample. If the node is a root, the root-level surrogate is updated instead.

Throughout a season, `UpdateSchedule` and `FinishSchedule` are called repeatedly to supply the workers with new tasks (nodes) until D is completely processed, at which point `GrowTrees` (Algorithm 4) is invoked to progress to the next depth. As discussed further in Section 3.4, `GrowTrees` (i) selects a subset of the best-performing nodes at depth D to branch from, and (ii) reduces the number of branches produced at each successive iteration within a season. When this number reaches zero, the season terminates and a new season begins. Since there is no limit on the number of seasons, ForestTune may be executed indefinitely, gradually exploring longer-running executions.

3.3 Efficient Scheduling

At each depth within a season, a certain – potentially large – number of configurations must be evaluated. Each configuration associated with a node m requires a certain allocation of system resources in order to run. In the simplest case, this is just because a portion of system memory is needed to fit a model which is to be trained. The requisite resources can also vary for different configurations, for example if one of the hyperparameters dictates how many CPU cores should be used. Multiple configurations can be evaluated in parallel as long as the sum of all of their resource requirements fits within the total system resources.

Since all nodes at a given depth must be processed before progressing, and processing a node can take a significant amount of time as the budget increases deeper into the forest, it is vital to produce an efficient schedule. A suboptimal scheduling order can significantly increase the total wall-clock time required to evaluate all nodes at a depth.

Algorithm 3 Node evaluation scheduling

function PackSchedule($\mathbf{R}_{\max}$)
- `schedule` $\leftarrow$ []
- $\mathbf{R}_{\text{remaining}} \leftarrow \mathbf{R}_{\max}$
- candidates $\leftarrow \mathsf{F}_D$
- SortDescending(candidates, $|m_R|$) (E)
- **for all** $m \in \mathsf{F}_D$ **do**
  - **if** $m_R \leq \mathbf{R}_{\text{remaining}} \wedge m_y = \text{null}$ **then**
    - *If node fits resources and is unevaluated…*
    - $\mathbf{R}_{\text{remaining}} \leftarrow \mathbf{R}_{\text{remaining}} - m_R$. (F)
    - `schedule` $\leftarrow$ `schedule` ++ $[m]$ *Append node to current schedule*

This scheduling problem is equivalent to efficient bin packing with a bin size of $\mathbf{R}_{\max}$, since we want to arrange the nodes at a given depth such that as few bins (i.e. *schedules*) as possible are required to fit them all. Since bin packing is NP-hard, ForestTune employs a heuristic based on the *first-fit decreasing* to achieve good scheduling efficiency in practice. Our approach is illustrated in Algorithm 3; nodes in F_D are sorted in descending order ((E)) of their required resources, and a schedule is filled until no more can fit ((F)). This process is repeated until all nodes at depth D have been assigned to a schedule.

This heuristic is simple to implement, incurs low overhead, and has proven effective in practice for balancing parallelism against resource utilisation. While it does not guarantee an optimal packing, it substantially reduces idle resources compared to naive scheduling strategies and integrates naturally with ForestTune's depth-wise execution model.

3.4 Seasons and Growth

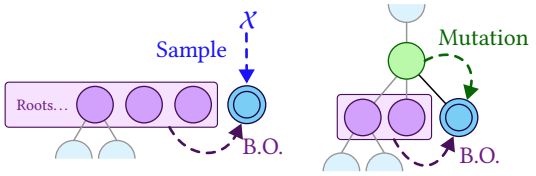
The structure and progression of seasons in ForestTune is governed by two parameters: η and T. η describes the *successive division factor*, and controls how rapidly the number of nodes selected for branching decreases across successive iterations within a season. T determines how many new execution trees are introduced at the beginning of each season. Considerations for choosing these parameters are discussed in Section 5. We now describe the behaviour of the tree growth procedure (Algorithm 4):

Algorithm 4 Tree growth algorithm

function GrowTrees($B : \mathbb{N}_0$)
- *First, produce new branches in the forest, from the currently considered depth*
- candidates $\leftarrow \mathsf{F}_D$
- $\varepsilon \sim \text{Gumbel}(0, \sigma(\text{candidates}_y))$ (G)
- SortDescending(candidates, $m_y + \varepsilon$) (H)
- **for** $i = 1, \ldots, B$ **do**
  - parent $\leftarrow$ candidates$_i$
  - child $\leftarrow$ CreateChild(parent) (I)
  - F.AddEdge(parent, child)
- $B \leftarrow \frac{B}{\eta}$ *Successive division* (J)
- $b \leftarrow b \times \eta$ *Exponential budget increase*
- **if** $B = 0$ **then** *Begin a new growth season*
  - $D \leftarrow 0$
  - $b \leftarrow b_{\min}$
  - **for** $i = 1, \ldots, T$ **do**
    - NewRootNode() (K)
  - $B \leftarrow |\mathsf{F}_0|$ *Reset number of branches*
- **else**
  - *Continue the current growth season*
  - $D \leftarrow D + 1$
- **return** B

`GrowTrees` takes as an input a single parameter, $B \in \mathbb{N}_0$, which is the number of branches that it should create. Branches may be created from any node in F_D, including nodes that already have children. Allowing re-branching from previously expanded nodes is essential to ForestTune's ability to explore multiple hyperparameter schedules that share common execution prefixes.

Branching is biased towards nodes with higher observed performance, and we therefore consider a performance-sorted view of F_D ((H)). To avoid repeatedly focusing on only the highest-performing nodes — thereby prematurely narrowing the explored configuration space and potentially discarding executions whose performance improves later — ForestTune does not deterministically select



(a) New root node **(b) Branch from node**

Figure 3: Origins of θ for the new node, ◎

the top-B nodes. Instead, selection is performed using a stochastic ranking mechanism.

Concretely, when sorting nodes, a random variable ε is added to each node's performance, such that higher-performing nodes remain more likely to be selected while lower-performing nodes retain a proportional probability. The perturbation ε is drawn from a *Gumbel* distribution scaled by the standard deviation of the set of all $m_y, m \in F_D$ (G). We use a Gumbel distribution rather than a Gaussian, as it provides stable and proportional jittered top-B selection even when $\sigma(\text{candidates}_y)$ is large; in such cases, Gaussian perturbations exhibit excessive variance and may disproportionately favour substantially inferior nodes.

From each of the B selected parent nodes, a new child node is created (I). The function `CreateChild` initiates a new node with a new hyperparameter configuration x drawn according to the candidate selection and mutation procedure, described below. A new node inherits the parent checkpoint state $\theta_{\text{child}} \leftarrow \theta_{\text{parent}}$, and starts unevaluated, $y = \text{null}$. Crucially, rather than a blank surrogate $\tilde{f}$, the child node receives a copy of its parent's: it adopts all of $\tilde{f}_{\text{parent}}$'s samples (so that it's usable for Bayesian optimisation right away if its parent had enough samples), but with posterior variance scaled by a fixed factor > 1 which allows it to benefit from existing samples, while encouraging broader exploration since the optimisation landscape varies throughout an execution.

B is then divided by η (J) so that subsequent growth stages produce fewer and fewer new nodes, concentrating the search on the best ones. If this update yields $B = 0$, the current season terminates, as no further branches can be created. In such a case, the next season is initialised by resetting D and B and creating T new root nodes (K). The procedure for selecting configurations for new root nodes is described in Section 3.5.

3.5 Candidate Selection and Mutation

There are two points in ForestTune where new TuneNodes are created: at the beginning of a season when new roots are selected, and during an iteration when a new branch is made from an existing node. Since the entire reason behind creating nodes and constructing a forest is to explore the space of configuration sequences, the selection of new nodes' configurations is naturally important. Depending on the context, ForestTune chooses between one of three different sources from which to select a configuration for a new TuneNode. Outlined in Figure 3, these are: uniform random sampling, incremental mutation, and Bayesian optimisation. In each diagram, it is the bold blue node's (◎) configuration that is being decided. We describe these mechanisms below, ordered from most to least informed.

Bayesian Optimisation. Both new root nodes (Figure 3a) and new nodes created during a season (Figure 3b) can be configured using Bayesian optimisation (*B.O.*). Bayesian optimisation can provide better guesses of what effective configurations will be by considering all previous performance examples and building a model to approximate the relationship $\mathcal{X} \to \mathbb{R}$. This model is referred to as a surrogate function, and it has the property of being cheap to sample (unlike the target function f, Section 2.4). Given a surrogate function, *B.O.* allows us to draw a new configuration $x \in \mathcal{X}$ that is the most *useful* to evaluate next. This decision balances exploitation of known promising configurations and exploration of unexplored configuration space.

ForestTune maintains a number of independent *B.O.* instances. A root-level surrogate model, $\tilde{f}_{\text{root}}$, is initialised at startup and is used to propose configurations for new root nodes. In addition, each TuneNode m maintains its own local surrogate model, $m_{\tilde{f}}$, which is derived from that its parent's, and further trained on observations obtained from its evaluated children.

In Figure 3, nodes whose configurations contribute towards the surrogate function relevant to the illustrated operation are encapsulated in boxes and painted purple: ◯. $\tilde{f}_{\text{root}}$ learns from all $m \in F_0, m_y \neq \text{null}$, while a parent node p's (Figure 3b, ◯) *B.O.* model considers all evaluated children. *B.O.* is only effective once a sufficient number of samples have been collected, with the required number depending on the dimensionality of $\mathcal{X}$. Consequently, it cannot be relied upon to select configurations for new root nodes early in execution. However, since newly created nodes inherit from their parent's surrogate, they are usually able to sample using *B.O.* right away.

It is instructive to contrast this design with approaches such as BOHB [10], which employ a single multi-fidelity surrogate that models configuration performance across different budgets. Though this is a great way to learn from both broad sweeps of low-budget executions as well as focused executions of promising configurations, it is not directly applicable to ForestTune because it assumes immutable hyperparameters. By instead using node-local surrogate models, ForestTune learns execution-stage-specific behaviour and thereby captures the performance of configuration *sequences*.

Node Mutation. When a parent node ◯ needs to branch and create a new child ◎ (Figure 3b), *B.O.* is attempted first to draw a new configuration. Since the parent's surrogate is inherited from *its* parent (unless it's a root), it's usually already suitable for sampling from. Mutation serves as a fallback for certain situations in which a surrogate lacks sufficient samples to be statistically useful (primarily early in a run). In such a case, $\bigcirc_\theta$ is perturbed slightly to produce the new configuration. A small perturbation is more appropriate than a uniform draw $\in \mathcal{X}$ because of the temporal correlation between adjacent stages of an execution.

The exact implementation of mutation is dependent on $\mathcal{X}$ and the semantics of the workload. We discuss suitable mutation strategies for the workloads evaluated in this paper, as well as general considerations, in Section 4.

Random Sampling. When creating new root nodes at the beginning of a season (Figure 3a), if the root surrogate $\tilde{f}_{\text{root}}$ has insufficient samples, configurations are drawn at random from $\mathcal{X}$. The sampling distribution for each hyperparameter may vary depending

on its semantics. Uniform distributions are appropriate when all values in a range are assumed equally likely to perform well, while alternative distributions such as log-normal may be more suitable for parameters like learning rate, where sensitivity varies across the domain.

In practice, the root *B.O.* model becomes useable after just a few evaluations, and by the second season Random Sampling is usually no longer necessary.

4 Implementation Considerations

ForestTune is written in Python, structured around a generic workload interface which makes it convenient to use and extend. Our machine learning workloads are implemented using *PyTorch* [25]. The source code is available upon request.

Constraining Memory Usage. As defined in Section 3, ForestTune conceptually maintains a model state for every node in the tuning forest, since execution may resume from any node at a later point. A naïve implementation would therefore incur unbounded memory growth as the forest expands. To address this, we store all model checkpoints on disk and retain models in memory only for nodes that are currently scheduled for execution. Models are loaded on demand when a node is scheduled and evicted once execution completes. In practice, the overhead of swapping models between memory and disk is negligible, as the dominant cost remains training time rather than checkpoint I/O.

Mutation. Standard unbiased mutation is performed by adding a normally distributed *r.v.* to each hyperparameter, with variance proportional to the size of that parameter's configuration space.

When mutating with a heuristic, the normal distributions take non-zero biased means: for both workloads it nudges learning rate and worker count downward over the schedule (CIFAR also nudges momentum up; GNN anneals the two neighbourhood fanouts k_1, k_2 and dropout down). These heuristics reflect workload-specific domain knowledge and are not intrinsic to the ForestTune framework. To account for this, in Section 5 we also evaluate two primary variants of ForestTune that do not rely on such heuristic biases, allowing us to assess the impact of mutation design choices independently of the core algorithm.

5 Evaluation

We assess ForestTune against two state-of-the-art hyperparameter tuning baselines: *PB2* [24], which, like ForestTune, also adjusts hyperparameters *during* training; and *BOHB* [10], a widely-used combination of Hyperband with Bayesian optimisation that holds each configuration fixed per trial. Our evaluation aims to address the following questions:

(1) Does ForestTune's support for mutable hyperparameters yield better-tuned models, and reach a given quality faster, than the baselines? (Section 5.2)
(2) How efficiently does ForestTune convert wall-clock time and evaluation budget into tuning progress? (Section 5.3)
(3) Which components of ForestTune contribute most to its performance gains? (Section 5.4)
(4) Is ForestTune practical in terms of the memory and storage overhead? (Section 5.5)

5.1 Experimantal Setup

We evaluate ForestTune on two parallel machine-learning workloads. The first trains a ResNet-20 [14] to classify CIFAR-100 images. The second trains a two-layer GraphSAGE [13] for node classification on the OGB-Arxiv citation network [15]. In both cases training is performed with asynchronous parallel SGD (`Hogwild!` [27]), which makes the number of parallel workers itself a tunable *system* hyperparameter. All experiments are performed on a single machine using an AMD *EPYC* 9754 processor, utilising 128 physical cores.

Budget unit. We measure budget in wall-clock seconds of training rather than in epochs. Epoch-based budgets interact poorly with the worker-count hyperparameter: a TuneNode training with a single worker takes far longer in wall-clock than one using many workers, so ForestTune's depth barrier would be dominated by low-parallelism stragglers. Defining a unit of budget as one second of training makes every node TuneNode at a given depth complete at the same time.

Configuration spaces. Both workloads use five-dimensional configuration spaces. For CIFAR-100, we tune the learning rate $r \in [10^{-4}, 5 \times 10^{-1}]$, batch size $B \in \{4, \ldots, 512\}$, momentum $\mu \in [0.2, 0.95]$, weight decay $\lambda \in [10^{-5}, 5 \times 10^{-3}]$, and the number of asynchronous workers $m \in \{1, \ldots, 128\}$. r, λ, and m are all sampled logarithmically. For OGB-Arxiv we tune the learning rate $r \in [10^{-4}, 10^{-1}]$, the two GraphSAGE neighbourhood sample sizes $k_1, k_2 \in \{5, \ldots, 25\}$, the dropout probability $p \in [0, 0.7]$, and the worker count $m \in \{1, \ldots, 128\}$. Again, r and m are sampled in log-space.

Baselines. We compare ForestTune (notated as "ForestTune (Hier)" to differentiate it from the two ablation variants) against two state-of-the-art tuners. *PB2* [24] is the closest point of comparison, as it also mutates hyperparameters dynamically during training across an evolving population. *BOHB* [10] combines Hyperband's successive halving with a TPE surrogate, but holds each configuration fixed for the lifetime of a trial. Both baselines are executed using the Ray Tune framework [19]. Because Ray Tune accounts each trial as requiring a fixed number of CPUs regardless of how many worker threads it spawns, we throttle all three tuners through a shared, cross-process resource pool that caps the total number of in-use cores at 128. This guarantees that every method is given identical hardware and that none can oversubscribe the machine.

Execution Parameters. Unless stated otherwise, ForestTune is run with $T = 16$ new roots per season, a successive-division factor of $\eta = 2$, and a minimum (depth-0) trial budget of 30 s on CIFAR-100 and 15 s on OGB-Arxiv. *PB2* and *BOHB* require setting certain hyperparameters, too, and to ensure the most fair comparison we performed a grid search for each to determine the best settings for each baseline-workload combination. Each (workload, tuner) configuration is evaluated using three random seeds. We report the mean best-so-far validation accuracy, with shaded regions indicating the best- and worst-seen performance. Each run is assigned a fixed wall-clock budget — 4000 s for CIFAR-100 and 1500 s for OGB-Arxiv — after which the best validation accuracy observed during the run is taken as the final result.

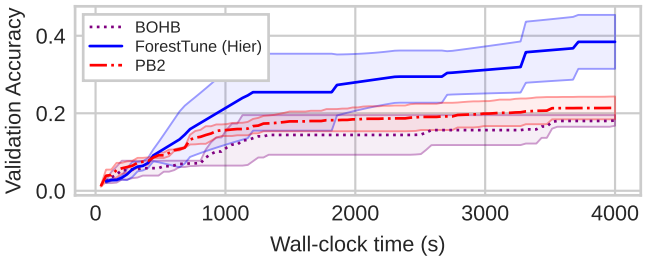


Figure 4: Best accuracy over wall-clock time for CIFAR-100

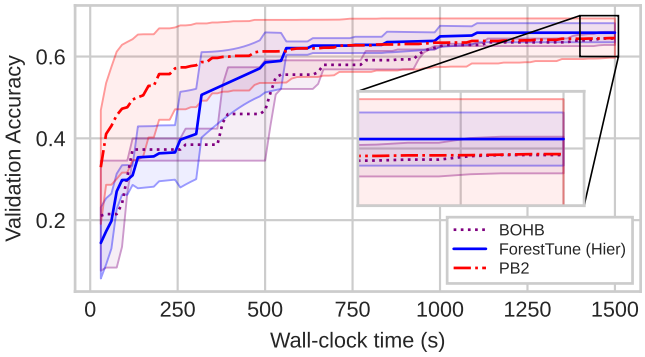


Figure 5: Best accuracy over wall-clock time for OGB-Arxiv

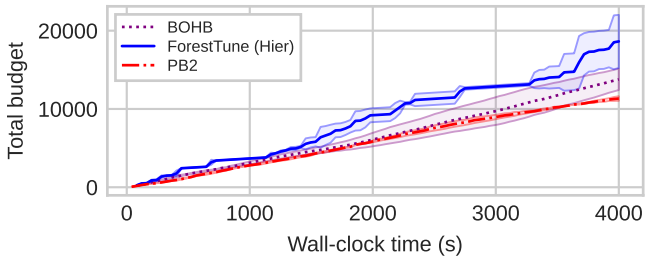


Figure 6: Cumulative budget usage for CIFAR-100

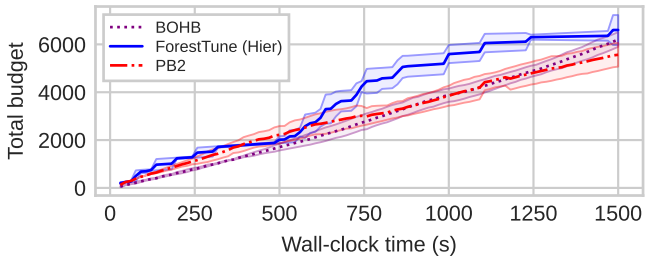


Figure 7: Cumulative budget usage for OGB-Arxiv

5.2 Tuning Quality

In order to evaluate the effectiveness of ForestTune as a hyperparameter tuner, we consider both the quality of the best model found and the wall-clock time required to reach it. The results thereof are presented in Figures 4 and 5, which plot the best achieved performance (i.e. validation accuracy; higher is better) against time for ForestTune, *PB2*, and *BOHB*. Unless otherwise stated, we report results for the *hierarchical* variant of ForestTune, which we treat as the default configuration. The impact of alternative variants is examined in Section 5.4.

Among the two workloads, CIFAR-100 is the more challenging, both in terms of training time and attainable accuracy. Consequently, it is also the workload on which differences between tuners are most pronounced. After 4000 s, ForestTune achieves an average validation accuracy of 0.384, compared to 0.202 for *PB2* and 0.182 for *BOHB*. This corresponds to a 90% relative improvement over *PB2*. Considering tuning speed, ForestTune reaches the final best accuracy achieved by *PB2* (0.202 at approximately $t = 3500$ s) after only 1000 s, representing a 71% reduction in time to reach this performance level. These results indicate that ForestTune is both more effective at identifying high-quality configurations and more efficient at converting wall-clock time into tuning progress on this workload.

On the OGB-Arxiv workload, all three tuners converge to similar final accuracies within comparable wall-clock times. While ForestTune attains a marginally higher average accuracy by the end of the run, the difference is small. In this case, *PB2* exhibits slightly better *any-time* performance, achieving higher accuracy earlier in the run, whereas ForestTune improves more gradually and overtakes after approximately 600 s. his behaviour is likely attributable to the reduced sensitivity of the OGB-Arxiv workload to precise hyperparameter settings. In such cases, the conservative incremental budgeting employed by both ForestTune and *BOHB* may be unnecessary, delaying early progress without providing substantial long-term benefit.

5.3 Budget Utilisation

We now examine *how* ForestTune achieves the performance gains observed in Section 5.2. In particular, we focus on two complementary aspects: (i) the efficiency with which training budget is consumed through resource-aware scheduling, and (ii) the effectiveness of configuration selection when controlling for total budget expended. Figures 8 and 9 illustrate how effectively ForestTune is able to find strong-performing configuration schedules; Figures 6 and 7 highlights the benefit of ForestTune's resource demand-aware trial scheduling, and Figure 10 shows ForestTune's choice of trial budgets compared to the baselines.

Trial Scheduling Efficiency. A central motivation behind ForestTune's design is explicit awareness of per-configuration resource demands. Because resource usage is treated as a first-class tunable parameter, ForestTune schedules trials concurrently only when their combined resource requirements fit within the system capacity (Section 3.3). In contrast, the baselines we consider assume uniform per-trial resource usage and therefore schedule trials without accounting for differences in parallelism, leading to periods of resource over-subscription. This effect is visible in Figures 6 and 7, which plot cumulative training budget consumed as a function of wall-clock time. For both workloads, ForestTune consistently consumes more training budget per unit of real time than *PB2* and *BOHB*, indicating more efficient utilisation of available system resources. As a result, ForestTune is able to evaluate more training work within the same wall-clock budget.

Configuration Selection Quality. Improved scheduling efficiency alone does not explain the tuning gains observed in Section 5.2. To isolate the effect of configuration selection, Figures 8 and 9 compare the best validation accuracy achieved by each tuner after consuming the same total amount of training budget, aggregated across concurrent trials. On CIFAR-100, after an initial exploration

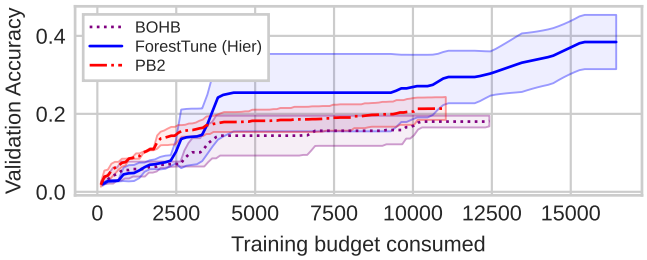


Figure 8: Best performance per budget used for CIFAR-100.

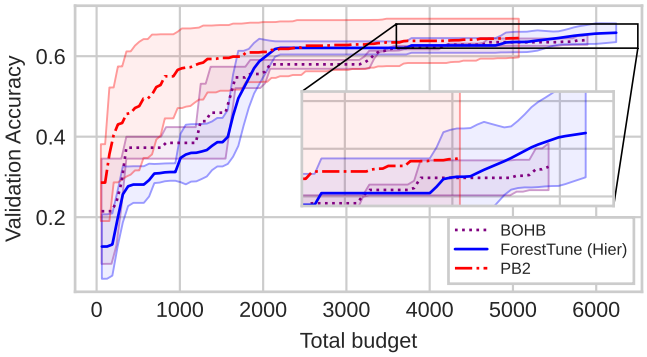


Figure 9: Best performance per budget used for OGB-Arxiv.

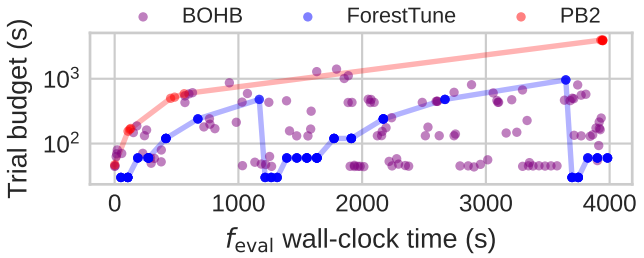


Figure 10: Trial budgets throughout execution (CIFAR-100)

phase, ForestTune maintains a clear advantage over both *PB2* and *BOHB* even when controlling for budget consumed. This indicates that ForestTune is not only more efficient at executing trials, but also more effective at identifying high-quality configuration schedules. On OGB-Arxiv, the differences between tuners remain modest, consistent with the results in Section 5.2.

Trial Budget Distribution. Unlike methods that impose a fixed maximum budget per trial, ForestTune places no explicit upper bound on the budgets it explores. While *PB2* similarly increases trial budgets linearly by continuously extending a fixed population, *BOHB* relies on a statically defined tournament structure with a predetermined maximum budget.

Figure 10 illustrates these differences. *PB2*'s structure means that its population of trials all monotonically increase in budget, saving time by never restarting executions. After a certain point, however, further budget yields diminishing returns. For this reason, *BOHB* shapes its trial brackets around a maximum budget; after this maximum is reached, new trials are started from scratch. The downside of this approach is the requirement of a good maximum budget, as a wrong estimate could either reduce the best achievable

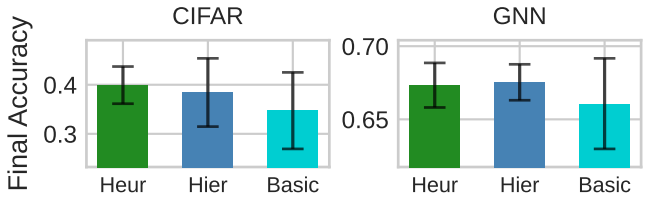


Figure 11: Summary of comparison between ForestTune variants.

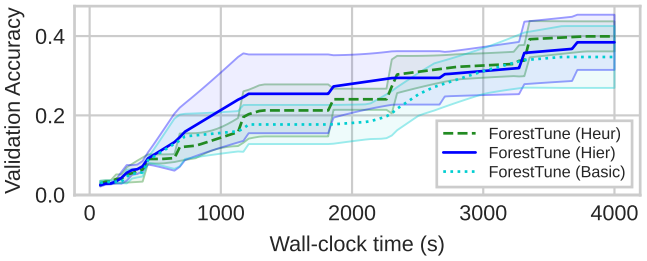


Figure 12: Tuning curve for each ForestTune variant on CIFAR-100.

performance (too low), or result in wasted time (too high). ForestTune avoids this trade-off by increasing the maximum explored budget gradually across seasons, as the achievable depth grows with the number of root nodes. As shown in Figure 10, this allows ForestTune to explore increasingly long-running trials without requiring an a priori decision on an appropriate maximum budget.

5.4 Ablation

To identify which components contribute most to ForestTune's performance we compare three variants. The hierarchical (*Hier*) variant initialises each node-local surrogate by inheriting its parent's GP, as described in Section 3. The *Basic* variant instead initialises each node-local surrogate with a uniform prior mean, without inheritance. Finally, we evaluate a heuristic-guided variant (*Heur*) that augments random mutation with a handwritten, workload-specific bias (described in Section 4).

A summary of each variant's final accuracy is presented in Figure 11. On CIFAR-100, the variant with hierarchical inheritance (*Hier*) has the best upper-bound on its performance, at the cost of greater variance compared to *Heur*. This can be explained by the hand-tuned heuristic providing more specific, directed search. Still, the non-heuristic *Hier* variant effectively finds a model with good performance. Comparing *Hier* and *Basic* provides justification for the hierarchical method, as accuracy drops from 0.39 to 0.35 without it. This happens because when a new node is *not* seeded with their parent's surrogate, the "fresh" GP does not get used until it has sufficient samples (instead falling back to random mutation). New nodes created under *Hier* can sample from their inherited GP right away.

On the near-saturated OGB-Arxiv workload, all three variants achieve similar final accuracies, within 0.66–0.68 on average, and differences are statistically negligible. However, *Basic* exhibits substantially higher variance across seeds than *Hier* and *Heur*. This is also observed on CIFAR-100. We attribute this to *Basic* having less

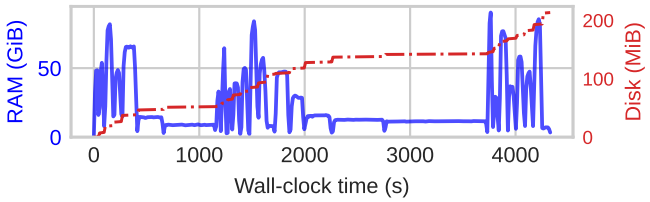


Figure 13: Memory and disk usage on CIFAR-100.

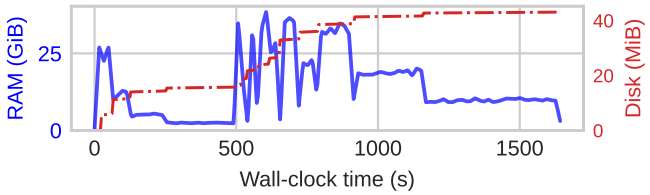


Figure 14: Memory and disk usage on OGB-Arxiv.

direction in its configuration search: it does not rely on prior knowledge from ancestor surrogates, or on expert-knowledge heuristics, and so its search is far more stochastic.

5.5 Memory Usage

ForestTune only holds the models and data relevant to active trials (i.e. those currently being evaluated) in RAM. Therefore, the physical memory usage cannot grow indefinitely, which is necessary for ForestTune to be practical. This behaviour is illustrated in Figures 13 and 14, which show memory consumption throughout a run for both workloads. Memory usage fluctuates with the number of concurrently active workers. The highest memory demand coincides with short periods in which many low-budget trials are evaluated in parallel, corresponding to the high-branching phases visible in Figure 10.

When TuneNodes are not actively being evaluated, their models are stored on disk so that they can be resumed in the future. Disk usage therefore grows monotonically, albeit to a degree an order of magnitude less than RAM usage for our use cases. Most of the RAM usage is not taken up by the model checkpoints, but rather by per-worker data such as training datasets and the Pytorch runtime.

6 Related Work

Existing hyperparameter tuning techniques which support discovery of hyperparameter schedules (as opposed to static configurations) tend to fall in the category of *population-based tuning*. The first such method, *PBT* [16], maintains a set of ongoing trials. During an iteration, each trial runs for some additional budget. The best performing configurations go on to seed the next round, whose trials continue from the best from the previous round. Configurations are perturbed in order to explore the search space. Instead of random perturbation, *Population-Based Bandits (PB2)* [24] draws subsequent configurations from a probabilistic model.

While these population-based methods implicitly explore hyperparameter schedules, there are drawbacks: Firstly, they assume that the system resources required to evaluate a certain configuration are outside the space of tunable hyperparameters. If this assumption does not hold (for example if the number of compute devices or amount of memory varies per configuration), they may attempt to concurrently evaluate more trials than there exist resources for. Secondly, *PB2* employs a single surrogate model to sample candidate configurations throughout the entire execution, which is unable to learn an optimisation landscape that varies throughout an execution. Therefore, late in a *PB2* execution, the prediction model may be primarily trained on less relevant samples.

Hyperband [18] introduced a tournament-bracket structure which lets more promising configurations use a greater proportion of time. Plain Hyperband samples new configurations from the search space at random, while BOHB [10] augments Hyperband with the use of Bayesian optimisation.

The popular framework *Optuna* [1] provides an interface for defining and running tuning tasks, and supports the use of Bayesian optimisation to make informed decisions of which static candidate configurations to trial.

Other interesting research directions within hyperparameter tuning including inference-aware techniques that takes into account the devices on which the model will be deployed [8, 28] and methods which estimate and consider the amount of power and memory used by neural networks [30].

7 Conclusion

We presented ForestTune, a resource-aware hyperparameter tuning system that supports mutable hyperparameters by modelling tuning as the construction of a forest of executions. By allowing executions to branch from intermediate checkpoints and evolve their hyperparameters over time, ForestTune discovers hyperparameter schedules rather than static configurations. This design enables effective reuse of partial executions, stage-specific optimisation through local surrogate models, and efficient utilisation of system resources.

Across two parallel machine-learning workloads, we showed that ForestTune achieves strong any-time performance and higher final model quality than state-of-the-art baselines. Our evaluation demonstrates that these gains arise both from more effective configuration selection and from resource-aware scheduling that converts wall-clock time into executed training budget more efficiently.

More broadly, ForestTune reframes hyperparameter tuning as a problem of *structured execution*. Rather than treating trials as independent, fixed-lifetime entities, ForestTune exposes execution state, resource usage, and stochasticity as first-class concerns. This enables principled branching, re-evaluation of promising trajectories, and coordinated scheduling decisions that span both algorithmic and system-level parameters. We believe this execution-centric view provides a flexible foundation for tuning long-running, resource-intensive workloads where static configurations are insufficient.

References

[1] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. 2019. Optuna: A Next-Generation Hyperparameter Optimization Framework. In *The 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2623–2631.

[2] Tal Ben-Nun and Torsten Hoefler. 2019. Demystifying Parallel and Distributed Deep Learning: An In-depth Concurrency Analysis. *ACM Comput. Surv.* 52, 4 (Aug. 2019), 65:1–65:43. doi:10.1145/3320060

[3] Mickael Binois, Jiangeng Huang, Robert Gramacy, and Mike Ludkovski. 2017. Replication or Exploration? Sequential Design for Stochastic Simulation Experiments. *Technometrics* 61 (10 2017). doi:10.1080/00401706.2018.1469433

[4] Xavier Bouthillier, Pierre Delaunay, Mirko Bronzi, Assya Trofimov, Brennan Nichyporuk, Justin Szeto, Nazanin Mohammadi Sepahvand, Edward Raff, Kanika Madan, Vikram Voleti, Samira Ebrahimi Kahou, Vincent Michalski, Tal Arbel, Chris Pal, Gael Varoquaux, and Pascal Vincent. 2021. Accounting for Variance in Machine Learning Benchmarks. *Proceedings of Machine Learning and Systems* 3 (March 2021), 747–769. https://proceedings.mlsys.org/paper_files/paper/2021/hash/0184b0cd3cfb185989f858a1d9f5c1eb-Abstract.html

[5] Karl Bäckström. 2023. Adaptiveness, Asynchrony, and Resource Efficiency in Parallel Stochastic Gradient Descent. https://research.chalmers.se/en/publication/535694 ISBN: 9789179058555.

[6] Yu-Hsin Chen, Tushar Krishna, Joel S. Emer, and Vivienne Sze. 2017. Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks. *IEEE Journal of Solid-State Circuits* 52, 1 (Jan. 2017), 127–138. doi:10.1109/JSSC.2016.2616357

[7] Xiaoge Deng, Tao Sun, Shengwei Li, Dongsheng Li, and Xicheng Lu. 2024. Stability and Generalization of Asynchronous SGD: Sharper Bounds Beyond Lipschitz and Smoothness. *Advances in Neural Information Processing Systems* 37 (Dec. 2024), 7675–7713. doi:10.52202/079017-0246

[8] Aditya Dhakal, K. K. Ramakrishnan, Sameer G. Kulkarni, Puneet Sharma, and Junguk Cho. 2022. Slice-Tune: a system for high performance DNN autotuning. In *Proceedings of the 23rd ACM/IFIP International Middleware Conference* (New York, NY, USA, 2022-11-08) *(Middleware '22)*. Association for Computing Machinery, 228–240. doi:10.1145/3528535.3565247

[9] Yasmine Djebrouni, Isabelly Rocha, Sara Bouchenak, Lydia Chen, Pascal Felber, Vania Marangozova, and Valerio Schiavoni. 2023. Characterizing Distributed Machine Learning Workloads on Apache Spark: (Experimentation and Deployment Paper). In *Proceedings of the 24th International Middleware Conference on ZZZ* (Bologna Italy, 2023-11-27). ACM, 151–164. doi:10.1145/3590140.3629112

[10] Stefan Falkner, Aaron Klein, and Frank Hutter. 2018. BOHB: Robust and Efficient Hyperparameter Optimization at Scale. In *Proceedings of the 35th International Conference on Machine Learning*. 1436–1445.

[11] Jacob Garby and Philippas Tsigas. 2026. Interval-Asynchrony: Delimited Intervals of Localised Asynchrony for Fast Parallel SGD. In *Euro-Par 2025: Parallel Processing*, Wolfgang E. Nagel, Diana Goehringer, and Pedro C. Diniz (Eds.). Springer Nature Switzerland, Cham, 236–249.

[12] Xavier Gastaldi. 2017. Shake-Shake regularization. doi:10.48550/arXiv.1705.07485 arXiv:1705.07485 [cs].

[13] William L. Hamilton, Rex Ying, and Jure Leskovec. 2018. Inductive Representation Learning on Large Graphs. arXiv:1706.02216 [cs.SI] https://arxiv.org/abs/1706.02216

[14] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep Residual Learning for Image Recognition. arXiv:1512.03385 [cs.CV] https://arxiv.org/abs/1512.03385

[15] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. 2021. Open Graph Benchmark: Datasets for Machine Learning on Graphs. arXiv:2005.00687 [cs.LG] https://arxiv.org/abs/2005.00687

[16] 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* abs/1711.09846 (2017). https://api.semanticscholar.org/CorpusID:1596043

[17] Scott Kirkpatrick, C. Gelatt, and M. Vecchi. 1983. Optimization by Simulated Annealing. *Science (New York, N.Y.)* 220 (06 1983), 671–80. doi:10.1126/science.220.4598.671

[18] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. 2018. Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization. arXiv:1603.06560 [cs.LG] https://arxiv.org/abs/1603.06560

[19] 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. *arXiv preprint arXiv:1807.05118* (2018).

[20] Ilya Loshchilov and Frank Hutter. 2016. SGDR: Stochastic Gradient Descent with Warm Restarts. doi:10.48550/arXiv.1608.03983

[21] Yujing Ma, Florin Rusu, and Martin Torres. 2019. Stochastic Gradient Descent on Modern Hardware: Multi-core CPU or GPU? Synchronous or Asynchronous?. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*.

[22] Sparsh Mittal and Shraiysh Vaishay. 2019. A survey of techniques for optimizing deep learning on GPUs. *Journal of Systems Architecture* 99 (Oct. 2019), 101635. doi:10.1016/j.sysarc.2019.101635

[23] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. 2021. Efficient large-scale language model training on GPU clusters using megatron-LM. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '21)*. Association for Computing Machinery, New York, NY, USA, 1–15. doi:10.1145/3458817.3476209

[24] Jack Parker-Holder, Vu Nguyen, and Stephen J Roberts. 2020. Provably Efficient Online Hyperparameter Optimization with Population-Based Bandits. 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., 17200–17211. https://proceedings.neurips.cc/paper/2020/file/c7af0926b294e47e52e46cfebe173f20-Paper.pdf

[25] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in PyTorch. (2017).

[26] Aurick Qiao, Sang Keun Choe, Suhas Jayaram Subramanya, Willie Neiswanger, Qirong Ho, Hao Zhang, Gregory R. Ganger, and Eric P. Xing. 2021. Pollux: Co-adaptive Cluster Scheduling for Goodput-Optimized Deep Learning. arXiv:2008.12260 [cs.DC] https://arxiv.org/abs/2008.12260

[27] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. 2011. Hogwild!: A lock-free approach to parallelizing stochastic gradient descent. *Advances in neural information processing systems* 24 (2011).

[28] Isabelly Rocha, Pascal Felber, Valerio Schiavoni, and Lydia Chen. 2022. EdgeTune: Inference-Aware Multi-Parameter Tuning. In *Proceedings of the 23rd ACM/IFIP International Middleware Conference* (New York, NY, USA, 2022-10-24) *(Middleware '22)*. Association for Computing Machinery, 1–14. doi:10.1145/3528535.3533273

[29] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. 2012. Practical Bayesian Optimization of Machine Learning Algorithms. In *Advances in Neural Information Processing Systems*, Vol. 25. Curran Associates, Inc. https://papers.nips.cc/paper/2012/hash/05311655a15b75fab86956663e1819cd-Abstract.html

[30] Dimitrios Stamoulis, Ermao Cai, Da-Cheng Juan, and Diana Marculescu. 2017. HyperPower: Power- and Memory-Constrained Hyper-Parameter Optimization for Neural Networks. arXiv:1712.02446 [cs] doi:10.48550/arXiv.1712.02446

[31] Chenning Xie, Rong Chen, Haibing Guan, Binyu Zang, and Haibo Chen. 2015. SYNC or ASYNC: time to fuse for distributed graph-parallel computation. *SIGPLAN Not.* 50, 8 (jan 2015), 194–204. doi:10.1145/2858788.2688508

[32] Yang You, Jing Li, Sashank Reddi, Jonathan Hseu, Sanjiv Kumar, Srinadh Bhojanapalli, Xiaodan Song, James Demmel, Kurt Keutzer, and Cho-Jui Hsieh. 2020. Large Batch Optimization for Deep Learning: Training BERT in 76 minutes. doi:10.48550/arXiv.1904.00962 arXiv:1904.00962 [cs].