Towards Efficient Automated Machine Learning

Liam Li

May 2020
CMU-ML-20-104

Machine Learning Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Ameet Talwalkar (Chair)
Maria-Florina Balcan
Jeff Schneider
Kevin Jamieson (University of Washington)

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy.

This work was supported in part by DARPA FA875017C0141, the National Science Foundation grants IIS1705121 and IIS1838017, an Okawa Grant, a Google Faculty Award, an Amazon Web Services Award, a JP Morgan A.I. Research Faculty Award, and a Carnegie Bosch Institute Research Award. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of DARPA, the National Science Foundation, or any other funding agency.

Keywords: AutoML, Hyperparameter Optimization, Neural Architecture Search

To my loving wife and family

Abstract

Machine learning is widely used in a variety of different disciplines to develop predictive models for variables of interest. However, building such solutions is a time consuming and challenging discipline that requires highly trained data scientists and domain experts. In response, the field of automated machine learning (AutoML) aims to reduce human effort and speedup the development cycle through automation.

Due to the ubiquity of hyperparameters in machine learning algorithms and the impact that a well-tuned hyperparameter configuration can have on predictive performance, hyperparameter optimization is a core problem in AutoML. More recently, the rise of deep learning has motivated neural architecture search (NAS), a specialized instance of a hyperparameter optimization problem focused on automating the design of neural networks. Naive approaches to hyperparameter optimization like grid search and random search are computationally intractable for large scale tuning problems. Consequently, this thesis focuses on developing efficient and principled methods for hyperparameter optimization and NAS.

In particular, we make progress towards answering the following questions with the aim of developing algorithms for more efficient and effective automated machine learning:

1. **Hyperparameter Optimization**
   (a) How can we effectively use early-stopping to speed up hyperparameter optimization?
   (b) How can we exploit parallel computing to perform hyperparameter optimization in the same time it takes to train a single model in the sequential setting?
   (c) For multi-stage machine learning pipelines, how can we exploit the structure of the search space to reduce total computational cost?
2. **Neural Architecture Search**
   (a) What is the gap in performance between state-of-the-art weight-sharing NAS methods and random search baselines?
   (b) How can we develop more principled weight-sharing methods with provably faster convergence rates and improved empirical performance?
   (c) Does the weight-sharing paradigm commonly used in NAS have applications to more general hyperparameter optimization problems?

Given these problems, this thesis is organized into two parts. The first part focuses on progress we have made towards efficient hyperparameter optimization by addressing Problems 1a, 1b, and 1c. The second part focuses on progress we have made towards understanding and improving weight-sharing for neural architecture search and beyond by addressing Problems 2a, 2b, and 2c.

Acknowledgments

I am grateful to have had the opportunity to work with so many amazing people throughout my graduate studies. The work presented in this thesis would not have been possible without them and the unwavering support from my loving family and friends.

First and foremost, I would like to thank my advisor Ameet Talwalkar. Ameet has been a constant source of support, encouragement, and guidance throughout graduate school. From our early days in UCLA to the past few years at CMU, Ameet has been my biggest advocate and believed in me even when I doubted myself. His confidence in me has been instrumental in helping me navigate the challenges I faced to get to where I am today. Thank you Ameet.

Special thanks to my close collaborators Professor Kevin Jamieson and Afshin Rostamizadeh for mentoring me when I first started graduate school and being great sources of inspiration and insight. It has also been great to work with Professor Nina Balcan on more recent projects and I have learned so much from her. Other professors I would like to thank include my co-authors Benjamin Recht, Moritz Hardt, and Virginia Smith; Katerina Fragkiadaki and Aarti Singh, course instructors for classes I TAed; and Ryan Tibshirani, for serving on the MLD Wellness Network. Thanks also to Professor Jamieson, Balcan, and Schneider for serving on my thesis committee.

It has been great to have worked with collaborators from multiple institutions. Special thanks to my close collaborators Misha Khodak and Maruan Al-Shedivat, both of whom are tremendously knowledgeable and taught me so much. Thanks also to Tian Li, Evan Sparks, Giulia DeSalvo, Haoping Bai, Nick Roberts, Jonathan Ben-Tzur, and Katerina Gonina.

I am also grateful for the friends that I have made throughout my graduate school including Chris Wu, Nolan Donoghue, Beer Changpinyo, and Ray Zhang from my time at UCLA; Misha, Greg Plumb, Sebastian Caldas, Jeffrey Li, Jeremy Cohen, Joon Kim, and Lucio Dery from the recently renamed Sage Lab; Maruan, Mariya Toneva, Anthony Platanios, and Simon Du, who were instrumental in launching the MLD Blog; Ifigeneia Apostolopoulou, Ritesh Noothigattu, Chieh Lin, Lisa Lee, and Leqi Liu from CMU; and Xavier Bouthillier, Jean-François Kagy, David Rosenberg, and Ningshan Zhang. Thanks also to Eric Forister, Maria Salgado, and Robert Sherman for helping me get into graduate school in the first place.

The administrative staff at CMU has also been a great resource: special thanks to Diane Stidle for always being there when I had questions and helping me feel at home at CMU and to Christy Melucci for helping our lab run so smoothly.

Finally, my family has been there for me from the very beginning: my mom Yuhong Chen, sister Grace Cantarella, Anthony and Grazia Cantarella, Jun Li, Du Chun, Louisa and Hansen Li; their love keeps me going. I dedicate this thesis to them and to my wife Seon Hee, who encouraged me to pursue my dreams, drove cross country with me through multiple blizzards to help me achieve them, and continues to be my greatest source of strength.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Machine learning has made significant strides with the rise of deep learning and large-scale model development evidenced by successes in robotics [Kober et al., 2013], healthcare [Greenspan et al., 2016], and autonomous driving [Chen et al., 2015]. However, developing these solutions is a resource intensive endeavour both in terms of computation and human expertise. Research in automated machine learning (AutoML) aims to alleviate both the computational cost and human effort associated with developing machine learning solutions through automation with efficient algorithms.

Applying machine learning to real world problems is a multi-stage process requiring significant human effort from data collection to model deployment. Although research in AutoML aim to automate all aspects of a typical machine learning pipeline, much of the research has focused on addressing the model search phase, where there is a multitude of tuning parameters that need to be set. These "hyperparameters" critically impact the performance of a given model and require careful tuning to achieve optimal performance. Hence, research in hyperparameter optimization aims to develop more efficient algorithms to reduce both the time and computational cost of identifying a good hyperparameter setting.

Recent trends towards larger models and search spaces have drastically increased the computational cost of hyperparameter optimization. For example, training a single state-of-the-art neural machine translation architectures can require days to weeks [Britz et al., 2017] and searching for quality image classification architectures can require evaluating over 20k architectures from a search space for neural architecture search [Real et al., 2018]. For these types of large-scale problems, naive methods like grid search and random search that allocate a uniform training resource to every configuration are prohibitively slow and expensive. The long wait before receiving a feedback on the performance of a well-tuned model limits user productivity, while the high cost limits broad accessibility. Consequently, faster and more cost efficient hyperparameter optimization methods are necessary for modern machine learning methods.

This thesis addresses this need with novel algorithms that exploit cost efficient methods of evaluating configurations, enabling the exploration of orders-of-magnitude more hyperparameter configurations than naive baselines. Before outlining the contributions presented in subsequent

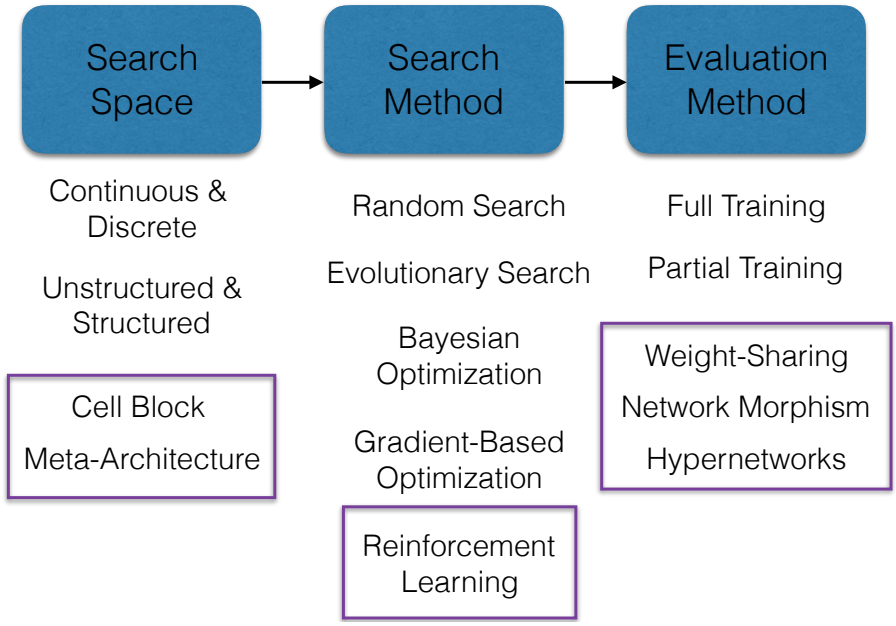


Figure 1.1: **Components of hyperparameter optimization.** Primarily NAS-specific methods are outlined in purple.

chapters towards efficient hyperparameter optimization and neural architecture search, we first provide relevant background on these two research subjects to contexualize our work.

1.2 Hyperparameter Optimization

We start with an overview of the components of hyperparameter optimization to provide a taxonomy with which to characterize different approaches. As shown in Figure 1.1, a general hyperparameter optimization problem has three components: (1) a search space, (2) a search method, and (3) an evaluation method. We provide a high level overview of each component in turn below and defer more detailed discussions as needed to Chapters 2, 3, and 4.

1.2.1 Search Space

Hyperparameter optimization involves identifying a good hyperparameter configuration from a set of possible configurations. The search space defines this set of configurations, and can include continuous or discrete hyperparameters in a structured or unstructured fashion [Bergstra and Bengio, 2012, Feurer et al., 2015a, Olson and Moore, 2016, Snoek et al., 2012]. Examples of continuous hyperparameters include learning rate and momentum for stochastic gradient descent, degree of regularization to apply to the training objective, and scale of a kernel similarity function for kernel classification. Examples of discrete hyperparameters include choices of activation function, number of layers in a neural network, and number of trees in a random forest. Finally, structured search spaces include those with conditional hyperparameters (e.g., the relevant

hyperparameters can depend on the choice of supervised learning method) and other tree type search spaces [Olson and Moore, 2016].

1.2.2 Search Method

Given a search space, there are various search methods to select putative configurations to evaluate. Random search over a predefined probability distribution over the search space is the most basic approach, yet it is quite effective in practice [Bergstra and Bengio, 2012]. Various adaptive methods have also been introduced, e.g., evolutionary search, gradient-based optimization, and Bayesian optimization. Although these adaptive approaches differ in how they determine which models to evaluate, they all attempt to bias the search in some way towards configurations that are more likely to perform well.

Due to the underlying assumptions made by different search methods, the choice of an appropriate search method can depends on the search space. Bayesian approaches based on Gaussian processes [Kandasamy et al., 2016, Klein et al., 2017a, Snoek et al., 2012, Swersky et al., 2013] and gradient-based approaches [Bengio, 2000, Maclaurin et al., 2015] are generally only applicable to continuous search spaces. In contrast, tree-based Bayesian [Bergstra et al., 2011, Hutter et al., 2011], evolutionary strategies [Olson and Moore, 2016], and random search are more flexible and can be applied to any search space. The application of reinforcement learning to general hyperparameter optimization problems is limited due to the difficulty of learning a policy over large continuous action spaces.

1.2.3 Evaluation Method

For each hyperparameter configuration considered by a search method, we must evaluate its quality. The default approach to perform such an evaluation involves fully training a model with the given hyperparameters, and subsequently measuring its quality, e.g., its predictive accuracy on a validation set. In contrast, partial training methods exploit early-stopping to speed up the evaluation process at the cost of noisy estimates of configuration quality. These methods use Bayesian optimization [Falkner et al., 2018, Kandasamy et al., 2016, Klein et al., 2017a], performance prediction [Domhan et al., 2015, Golovin et al., 2017], or, as we show in Chapter 2, multi-armed bandits to adaptively allocate resources to different configurations.

As we will will discuss in Section 1.4, our contributions to efficient hyperparameter optimization largely focus on developing principled early-stopping methods to speed up configuration evaluation.

1.3 Neural Architecture Search

The success of deep learning in various challenging real-world applications has generated significant interest in designing even better neural network architectures. Initial work in neural architecture search (NAS) by Zoph and Le [2017] used reinforcement learning to identify state-of-the-art architectures for image classification and language modeling demonstrating the promising potential of NAS to automatically identify architectures that outperform human designed ones.

Although NAS may appear to be a subfield of AutoML independent of hyperparameter optimization, in reality, it is a specialized hyperparameter optimization problem distinguished by the search spaces considered. Hence, we use the same taxonomy as that in the previous section to provide an overview for NAS methods. We will discuss concepts in more detail as needed in Chapters 5, 6, and 7.

1.3.1 Search Space

NAS-specific search spaces usually involve discrete architectural hyperparameters that control at a more granular level which operations to use and how to connect the outputs of these operations to form an architecture. Architecture instances within these search spaces can be represented as directed acyclic graphs (DAG) [Liu et al., 2019, Pham et al., 2018], where nodes represent local computation to form intermediate features and edges indicate the flow of information from one node to another. Additionally, since a search space for designing an entire architecture would have too many nodes and edges, search spaces are usually defined over some smaller building block, i.e., cell blocks, that are repeated in some way via a preset or learned meta-architecture to form a larger architecture [Elsken et al., 2018b].

1.3.2 Search Method

NAS-specific search methods can be categorized into the same broad categories as those for hyperparameter optimization but are specially tailored for structured NAS search spaces. Reinforcement learning and evolutionary search approaches dominated early methods due to their suitability for complex discrete search spaces [Real et al., 2017, 2018, Zoph and Le, 2017]. More recently, gradient-based optimization of a continuous relaxation of the discrete search space has also seen success [Cai et al., 2019, Liu et al., 2019]. Bayesian optimization approaches with specialized architecture similarity kernels have also been developed for NAS [Jin et al., 2018, Kandasamy et al., 2018].

1.3.3 Evaluation Method

The first generation of NAS methods relied on full training evaluation, and thus required thousands of GPU days to achieve a desired result [Real et al., 2017, 2018, Zoph and Le, 2017, Zoph et al., 2018]. NAS-specific evaluation methods —such as network morphism, weight-sharing, and hypernetworks— exploit the structure of neural networks to provide even cheaper, heuristic estimates of quality. Many of these methods center around sharing and reuse: network morphisms build upon previously trained architectures [Cai et al., 2018, Elsken et al., 2018a, Jin et al., 2018]; hypernetworks and performance prediction encode information from previously seen architectures [Brock et al., 2018, Liu et al., 2018a, Zhang et al., 2019]; and weight-sharing methods [Bender et al., 2018, Cai et al., 2019, Liu et al., 2019, Pham et al., 2018, Xie et al., 2019] use a single set of weights for all possible architectures. These more efficient NAS evaluation methods are 2-3 orders-of-magnitude cheaper than full training.

As we will will discuss in Section 1.4, our contributions to more efficient NAS largely focus on developing a better understanding of the weight-sharing evaluation scheme and designing better architecture search algorithms using our insights.

1.4 Thesis Outline and Contributions

With the requisite background from the previous section, we now present our contributions towards more efficient automated machine learning. This thesis is organized into two parts, the first part presents our contributions towards more efficient hyperparameter optimization while the second part focuses on our contributions in neural architecture search. We provide an outline of our contributions in each chapter below with a motivating problem and our associated solution.

Part 1: Hyperparameter Optimization

Motivated by the trend towards ever more expensive models and larger search spaces, we endeavour to develop faster methods for hyperparameter optimization that can evaluate orders-of-magnitude more configurations than when using full training with the same computational resource. Our contributions below exploit early-stopping and reuse of computation to drastically decrease the average cost of evaluating a configuration, thereby increasing the total number of configurations we can evaluate with the same resource.

This part is based on work presented in Li et al. [2017], Li et al. [2018a], Li et al. [2020a], and Li et al. [2018b].

Chapter 2: Principled Early-Stopping with Hyperband

Problem Early-stopping is a commonly used during manual hyperparameter tuning and is a natural approach to speed up the search for a quality hyperparameter setting. Early-stopping rules like median early-stopping rule [Golovin et al., 2017] and improvement-based stopping criterion are also available. However, these approaches are heuristic in nature and, therefore, lack correctness and complexity guarantees. As we will show in Chapter 2, learning curves for different hyperparameter configurations are not well behaved and can be non-monotonic, non-smooth, and even diverge. Hence, a more principled approach is needed to guarantee robustness and efficiency.

Solution [Li et al., 2017, 2018a] In contrast to prior work that rely on heuristics to perform early-stopping, we develop Hyperband, a principled approach to early-stopping for hyperparameter optimization based on the successive halving bandit algorithm [Karnin et al., 2013]. Our bandit-based algorithm poses hyperparameter optimization as a pure-exploration infinitely armed bandit problem. Then, under weak assumptions, we show Hyperband only requires a small factor more resource than that of optimal resource allocation to identify a sufficiently good configuration. Our experiments on a suite of hyperparameter tuning tasks demonstrate Hyperband to be over an order-of-magnitude faster than methods that use full evaluation.

Chapter 3: Hyperparameter Optimization in the Large-Scale Regime

Problem Hyperparameter tuning workloads for modern machine learning models require evaluating thousands of expensive to train models motivating the *large-scale regime* for hyperparameter optimization. In the large-scale regime, we have to exploit parallelism afforded to us through the widespread availability of cloud computing to evaluate orders-of-magnitude more models than available workers in order to return a good configuration in a small multiple of the wallclock time required to train a single model. How can we design an algorithm suitable for the large-scale regime of hyperparameter optimization?

Solution [Li et al., 2020a] We develop the **A**synchronous **S**uccessive **H**alving **A**lgorithm (ASHA) for embarrassingly parallel asynchronous hyperparameter optimization. ASHA can identify a quality configurations in the time it takes to train a single model if given enough distributed workers. Our algorithm addresses drawbacks of original synchronous successive halving algorithm (Chapter 2) when running in distributed computing system. In particular, the asynchronous algorithm is robust to stragglers and dropped jobs and scales linearly with the number of workers. We also discuss systems design considerations that we faced when deploying ASHA in a live production system. Our empirical studies for both sequential and distributed settings show that ASHA can achieve state-of-the-art performance across a broad array of hyperparameter tuning tasks.

Impact The theoretically principled algorithms we develop in Chapter 2 and Chapter 3 have gained wide adoption, as evidenced by the availability of HYPERBAND and ASHA in many AutoML frameworks like Optuna [Akiba et al.], Katib [Zhou et al., 2019], and Ray Tune [Liaw et al., 2018].

Chapter 4: Reuse in Pipeline-Aware Hyperparameter Optimization

Problem Machine learning solutions often involve multiple stages forming a pipeline for data processing, featurization, and model training. Each stage of the pipeline involves methods with associated hyperparameters that need to be tuned to maximize performance on the downstream task. For these structured search spaces, is there a way to reuse computation across pipeline configurations to reduce total computational cost?

Solution [Li et al., 2018b] We present a novel paradigm for hyperparameter optimization in structured search spaces where shared computation between pipelines are reused instead of recomputed for each individual pipeline. Under this paradigm, we present an approach that optimizes both the design and execution of pipelines to maximize speedups from reuse. We design pipelines amenable for reuse by (1) introducing a novel hybrid hyperparameter tuning method called gridded random search, and (2) reducing the average training time in pipelines by adapting early-stopping hyperparameter tuning approaches. We conduct experiments on simulated and real-world machine learning pipelines to show that a pipeline-aware approach to hyperparameter tuning can offer over an order-of-magnitude speedup over independently evaluating pipeline configurations.

Part 2: Neural Architecture Search

Due to the astronomical cost associated with neural architecture search methods that relied on full training [Real et al., 2017, 2018, Zoph and Le, 2017], weight-sharing has emerged as a cost-effective evaluation strategy for NAS. Weight-sharing methods reduce the cost of architecture search to that of training a single super network encompassing all possible architectures within the search space [Liu et al., 2019, Pham et al., 2018]. Although weight-sharing NAS methods have seen empirical success [Cai et al., 2019, Liu et al., 2019, Pham et al., 2018], we address the need for a better understanding of weight-sharing paradigm to inform more principled and robust methods.

This part is based on work presented in Li and Talwalkar [2019], Li et al. [2020b], and Khodak et al. [2020].

Chapter 5: Random Search Baselines for NAS

Problem While neural architecture search methods have identified state-of-the art architectures for image recognition and language modeling, it is unclear how much of a gap exists between traditional hyperparameter optimization methods and specialized NAS approaches. How do traditional hyperparameter optimization methods perform on NAS search spaces and what are the right baselines for NAS methods? In particular, are the speedups from weight-sharing methods attributable to the search method used or to the faster architecture evaluation scheme?

Solution [Li and Talwalkar, 2019] Motivated by our work in hyperparameter optimization, we investigate the effectiveness of random search in combination with efficient evaluation strategies on two standard NAS benchmarks. Towards this effort, we evaluate ASHA on these two benchmarks to quantify the gap between NAS methods and traditional hyperparameter optimization approaches. Additionally, we develop a novel algorithm for NAS that combines random search with weight-sharing (RSWS) to drastically reduce evaluation cost. Our results establish RSWS as a strong baseline for NAS that has since proven to be a difficult baseline to beat [Carlucci et al., 2019, Cho et al., 2019, Dong and Yang, 2020, Yu et al., 2020]. Finally, we discuss the state of reproducibility in NAS and present suggestions to improve results reporting to ground future empirical studies.

Impact At the time of publication, RSWS attained state-of-the-art performance on designing RNN cells for language modeling on the Penn Treebank dataset [Marcus et al., 1993] and remains the method with the top result on that benchmark. RSWS has also gained traction as a strong baseline for NAS, warranting comparisons from recent NAS work [Cho et al., 2019, Dong and Yang, 2020, Zela et al., 2020b].

Chapter 6: Geometry-Aware Optimization for Gradient-Based NAS

Problem State-of-the-art NAS methods that outperform RSWS by larger margins combine gradient-based optimization techniques with weight-sharing to speed up the search for a good architecture. However, this training process remains poorly understood leading to a multitude of

methods based on heuristics and intuition. How do these methods behave and how can we design more principled approaches with faster convergence rates and improved empirical performance?

Solution [Li et al., 2020b] We present a geometry-aware framework for designing and analyzing gradient-based NAS methods by drawing a connection to the theory of mirror descent. Our framework allows us to reason about the convergence rate of existing gradient-based weight-sharing approaches to NAS. Furthermore, our convergence analysis allows us to design novel methods that exploit the underlying problem structure to quickly find highly-performant architectures. Our methods (1) enjoy faster convergence guarantees whose iteration complexity has a weaker dependence on the size of the search space and (2) achieve higher accuracy on the latest NAS benchmarks in computer vision.

Chapter 7: Weight-Sharing Beyond NAS

Problem Although NAS is a specialized instance of a hyperparameter optimization problem, it is natural to wonder whether NAS specific methods have applications more broadly. Weight-sharing in particular holds particular appeal due to its computational efficiency, reducing the cost of exploring all architectures in the NAS search space to that of training a single network. Is it possible to apply weight-sharing to problems beyond NAS and what insights can we derive from this exploration?

Solution [Khodak et al., 2020] We identify weight-sharing as a widely applicable technique for accelerating traditional hyperparameter optimization. We study weight-sharing for feature map selection and show that it yields highly competitive performance. We also pose this setting as a theoretically-tractable type of NAS and provide a sample-complexity-based explanation for why NAS practitioners frequently obtain better performance using bilevel optimization instead of joint empirical risk minimization over architectures and shared weights. Finally, we argue that weight-sharing is a natural approach for hyperparameter optimization in the federated learning setting and introduce an exponentiated gradient algorithm called FedEx for this settings

Part I

Hyperparameter Optimization

Chapter 2

Principled Early-Stopping with Hyperband

With the requisite background on hyperparameter optimization in Chapter 1.2, we present HYPERBAND, our algorithm for theoretically grounded early-stopping, in this chapter.

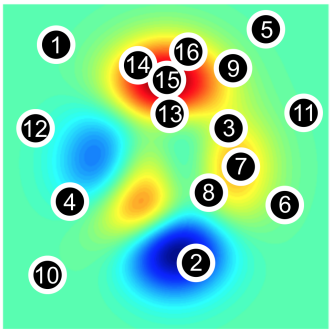
2.1 Introduction

Early efforts towards more efficient hyperparameter optimization were dominated by *Bayesian optimization* methods [Bergstra et al., 2011, Hutter et al., 2011, Snoek et al., 2012] that focus on optimizing hyperparameter *configuration selection*. These methods aim to identify good configurations more quickly than standard baselines like random search by selecting configurations in an adaptive manner; see Figure 2.1a. Although empirical evidence suggests that these methods outperform random search [Eggensperger et al., 2013, Snoek et al., 2015, Thornton et al., 2013], these approaches tackle the fundamentally challenging problem of simultaneously fitting and optimizing a high-dimensional, non-convex function with unknown smoothness, and possibly noisy evaluations.

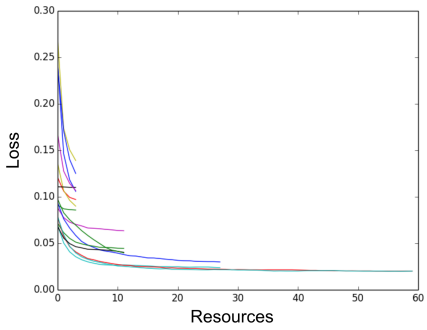
An orthogonal approach to hyperparameter optimization focuses on speeding up *configuration evaluation*; see Figure 2.1b. These approaches are adaptive in computation, allocating more resources to promising hyperparameter configurations while quickly eliminating poor ones. Resources can take various forms, including size of training set, number of features, or number of iterations for iterative algorithms. By adaptively allocating resources, these approaches aim to examine orders-of-magnitude more hyperparameter configurations than approaches that uniformly train all configurations to completion, thereby quickly identifying good hyperparameters.

More recently, hybrid approaches that combine adaptive configuration selection with adaptive evaluation have been introduced [Domhan et al., 2015, Falkner et al., 2018, Jaderberg et al., 2017, Klein et al., 2017a]. In this chapter, we focus on speeding up random search as it offers a simple and theoretically principled launching point [Bergstra and Bengio, 2012].[1] We will compare early-stopping with simple random search to leading hybrid approaches in Chapter 3.

[1]Random search will asymptotically converge to the optimal configuration, regardless of the smoothness or structure of the function being optimized, by a simple covering argument. While the rate of convergence for random search depends on the smoothness and is exponential in the number of dimensions in the search space, the same is true for Bayesian optimization methods without additional structural assumptions [Kandasamy et al., 2015].



(a) Configuration Selection



(b) Configuration Evaluation

Figure 2.1: (a) The heatmap shows the validation error over a two-dimensional search space with red corresponding to areas with lower validation error. Configuration selection methods adaptively choose new configurations to train, proceeding in a sequential manner as indicated by the numbers. (b) The plot shows the validation error as a function of the resources allocated to each configuration (i.e. each line in the plot). Configuration evaluation methods allocate more resources to promising configurations.

In this chapter, as a counterpoint to prior work focused on adaptive configuration selection, we develop a novel configuration evaluation approach by formulating hyperparameter optimization as a pure-exploration adaptive resource allocation problem addressing how to allocate resources among randomly sampled hyperparameter configurations. Our procedure, HYPERBAND, relies on a principled early-stopping strategy to allocate resources, allowing it to evaluate orders-of-magnitude more configurations than black-box procedures like Bayesian optimization methods. HYPERBAND is a general-purpose technique that makes minimal assumptions unlike prior configuration evaluation approaches [Agarwal et al., 2011, Domhan et al., 2015, György and Kocsis, 2011, Jamieson and Talwalkar, 2015, Sparks et al., 2015, Swersky et al., 2014].

Our theoretical analysis demonstrates the ability of HYPERBAND to adapt to unknown convergence rates and to the behavior of validation losses as a function of the hyperparameters. In addition, HYPERBAND is $5\times$ to $30\times$ faster than popular Bayesian optimization algorithms on a variety of deep-learning and kernel-based learning problems. A theoretical contribution of this chapter is the introduction of the pure-exploration, infinite-armed bandit problem in the non-stochastic setting, for which HYPERBAND is one solution.

The remainder of the chapter is organized as follows. Section 2.2 summarizes related work in two areas: (1) hyperparameter optimization, and (2) pure-exploration bandit problems. Section 2.3 describes HYPERBAND and the successive halving bandit algorithm, which HYPERBAND calls as a subroutine. In Section 2.4, we present a wide range of empirical results comparing HYPERBAND with popular adaptive configuration selection methods. Section 2.5 frames the hyperparameter optimization problem as an infinite-armed bandit problem and summarizes the theoretical results for HYPERBAND.

2.2 Related Work

In this section, we provide a discussion of early work in hyperparameter optimization motivating HYPERBAND and also summarize significant related work on bandit problems. We will discuss more recent methods introduced after HYPERBAND in the next chapter.

2.2.1 Hyperparameter Optimization

Bayesian optimization techniques model the conditional probability $p(y|\lambda)$ of a configuration's performance on an evaluation metric y (i.e., test accuracy), given a set of hyperparameters λ. Sequential Model-based Algorithm Configuration (SMAC), Tree-structure Parzen Estimator (TPE), and Spearmint are three well-established methods [Feurer et al., 2014]. SMAC uses random forests to model $p(y|\lambda)$ as a Gaussian distribution [Hutter et al., 2011]. TPE is a non-standard Bayesian optimization algorithm based on tree-structured Parzen density estimators [Bergstra et al., 2011]. Lastly, Spearmint uses Gaussian processes (GP) to model $p(y|\lambda)$ and performs slice sampling over the GP's hyperparameters [Snoek et al., 2012].

Previous work compared the relative performance of these Bayesian searchers [Bergstra et al., 2011, Eggensperger et al., 2013, Feurer et al., 2014, 2015a, Snoek et al., 2012, Thornton et al., 2013]. An extensive survey of these three methods by Eggensperger et al. [2013] introduced a benchmark library for hyperparameter optimization called HPOlib, which we use for our experiments. Bergstra et al. [2011] and Thornton et al. [2013] showed Bayesian optimization methods empirically outperform random search on a few benchmark tasks. However, for high-dimensional problems, standard Bayesian optimization methods perform similarly to random search [Wang et al., 2013]. Methods specifically designed for high-dimensional problems assume a lower effective dimension for the problem [Wang et al., 2013] or an additive decomposition for the target function [Kandasamy et al., 2015]. However, as can be expected, the performance of these methods is sensitive to required inputs; i.e. the effective dimension [Wang et al., 2013] or the number of additive components [Kandasamy et al., 2015].

Gaussian processes have also been studied in the bandit setting using confidence bound acquisition functions (GP-UCB), with associated sublinear regret bounds [Grünewälder et al., 2010, Srinivas et al., 2010]. Wang et al. [2016] improved upon GP-UCB by removing the need to tune a parameter that controls exploration and exploitation. Contal et al. [2014] derived a tighter regret bound than that for GP-UCB by using a mutual information acquisition function. However, van der Vaart and van Zanten [2011] showed that the learning rate of GPs are sensitive to the definition of the prior through an example with a poor prior where the learning rate degraded from polynomial to logarithmic in the number of observations n. Additionally, without structural assumptions on the covariance matrix of the GP, fitting the posterior is $O(n^3)$ [Wilson et al., 2015]. Hence, Snoek et al. [2015] and Springenberg et al. [2016] proposed using Bayesian neural networks, which scale linearly with n, to model the posterior.

Adaptive configuration evaluation is not a new idea. Maron and Moore [1997] and Mnih and Audibert [2008] considered a setting where the training time is relatively inexpensive (e.g., k-nearest-neighbor classification) and evaluation on a large validation set is accelerated by evaluating on an increasing subset of the validation set, stopping early configurations that are performing poorly. Since subsets of the validation set provide unbiased estimates of its expected performance,

this is an instance of the *stochastic* best-arm identification problem for multi-armed bandits [see the work by Jamieson and Nowak, 2014, for a brief survey].

In contrast, we address a setting where the evaluation time is relatively inexpensive and the goal is to early-stop long-running training procedures by evaluating partially trained models on the full validation set. Previous approaches in this setting either require strong assumptions or use heuristics to perform adaptive resource allocation. György and Kocsis [2011] and Agarwal et al. [2011] made parametric assumptions on the convergence behavior of training algorithms, providing theoretical performance guarantees under these assumptions. Unfortunately, these assumptions are often hard to verify, and empirical performance can drastically suffer when they are violated. Krueger et al. [2015] proposed a heuristic based on sequential analysis to determine stopping times for training configurations on increasing subsets of the data. However, the theoretical correctness and empirical performance of this method are highly dependent on a user-defined "safety zone."

In another line of work, Sparks et al. [2015] proposed a halving style bandit algorithm that did not require explicit convergence behavior, and Jamieson and Talwalkar [2015] analyzed a similar algorithm originally proposed by Karnin et al. [2013] for a different setting, providing theoretical guarantees and encouraging empirical results. Unfortunately, these halving style algorithms suffer from the "n versus B/n" problem, which we will discuss in Section 2.3.1. HYPERBAND addresses this issue and provides a robust, theoretically principled early-stopping algorithm for hyperparameter optimization.

We note that HYPERBAND can be combined with any hyperparameter sampling approach and does not depend on random sampling; the theoretical results only assume the validation losses of sampled hyperparameter configurations are drawn from some stationary distribution. In fact, we compare to a method combining HYPERBAND with adaptive sampling in Chapter 3.4.

2.2.2 Bandit Problems

Pure exploration bandit problems aim to minimize the simple regret, defined as the distance from the optimal solution, as quickly as possible in any given setting. The pure-exploration multi-armed bandit problem has a long history in the stochastic setting [Bubeck et al., 2009, Even-Dar et al., 2006], and was extended to the non-stochastic setting by Jamieson and Talwalkar [2015]. Relatedly, the stochastic pure-exploration infinite-armed bandit problem was studied by Carpentier and Valko [2015], where a pull of each arm i yields an i.i.d. sample in $[0, 1]$ with expectation ν_i, where ν_i is a loss drawn from a distribution with cumulative distribution function, F. Of course, the value of ν_i is unknown to the player, so the only way to infer its value is to pull arm i many times. Carpentier and Valko [2015] proposed an anytime algorithm, and derived a tight (up to polylog factors) upper bound on its error assuming what we will refer to as the β-parameterization of F described in Section 2.5.3.2. that succeeds with probability at least δ such that after B total pulls from all drawn arms it outputs an arm $\widehat{i}$ that satisfies

$$\mathbb{E}[\nu_{\widehat{i}}] \leq \text{polylog}(B) \log(1/\delta) \max\{B^{-\frac{1}{2}}, B^{-\frac{1}{\beta}}\}. \tag{2.1}$$

Conversely, they show that any algorithm that outputs an arm $\widehat{i}$ after just B pulls must also satisfy $\mathbb{E}[\nu_{\widehat{i}}] \geq \max\{B^{-\frac{1}{2}}, B^{-\frac{1}{\beta}}\}$ showing that their algorithm is tight up to polylog factors

for this β parameterization of F. However, their algorithm was derived specifically for the β-parameterization of F, and furthermore, they must estimate β before running the algorithm, limiting the algorithm's practical applicability. Also, the algorithm assumes stochastic losses from the arms and thus the convergence behavior is known; consequently, it does not apply in our hyperparameter optimization setting.[2] Two related lines of work that both make use of an underlying metric space are Gaussian process optimization [Srinivas et al., 2010] and X-armed bandits [Bubeck et al., 2011], or bandits defined over a metric space. However, these works either assume stochastic rewards or need to know something about the underlying function (e.g. an appropriate kernel or level of smoothness).

In contrast, HYPERBAND is devised for the non-stochastic setting and automatically adapts to unknown F without making any parametric assumptions. Hence, we believe our work to be a generally applicable pure exploration algorithm for infinite-armed bandits. To the best of our knowledge, this is also the first work to test out such an algorithm on a real application.

2.3 HYPERBAND Algorithm

In this section, we present the HYPERBAND algorithm. Moreover, we provide intuition for the algorithm and present a few guidelines on how to deploy HYPERBAND in practice.

2.3.1 Successive Halving

HYPERBAND extends the SUCCESSIVEHALVING algorithm proposed for hyperparameter optimization by Jamieson and Talwalkar [2015] and calls it as a subroutine. The idea behind the original SUCCESSIVEHALVING algorithm follows directly from its name: uniformly allocate a budget to a set of hyperparameter configurations, evaluate the performance of all configurations, throw out the worst half, and repeat until one configuration remains. The algorithm allocates exponentially more resources to more promising configurations. Unfortunately, SUCCESSIVEHALVING requires the number of configurations n as an input to the algorithm. Given some finite budget B (e.g., an hour of training time to choose a hyperparameter configuration), B/n resources are allocated on average across the configurations. However, for a fixed B, it is not clear a priori whether we should (a) consider many configurations (large n) with a small average training time; or (b) consider a small number of configurations (small n) with longer average training times.

We use a simple example to better understand this tradeoff. Figure 2.2 shows the validation loss as a function of total resources allocated for two configurations with terminal validation losses ν_1 and ν_2. The shaded areas bound the maximum deviation of the intermediate losses from the terminal validation loss and will be referred to as "envelope" functions.[3] It is possible to distinguish between the two configurations when the envelopes no longer overlap. Simple arithmetic shows that this happens when the width of the envelopes is less than $\nu_2 - \nu_1$, i.e., when the intermediate losses are guaranteed to be less than $\frac{\nu_2 - \nu_1}{2}$ away from the terminal losses. There

[2] See the work by Jamieson and Talwalkar [2015] for detailed discussion motivating the non-stochastic setting for hyperparameter optimization.

[3] These envelope functions are guaranteed to exist; see discussion in Section 2.5.2 where we formally define these envelope (or γ) functions.

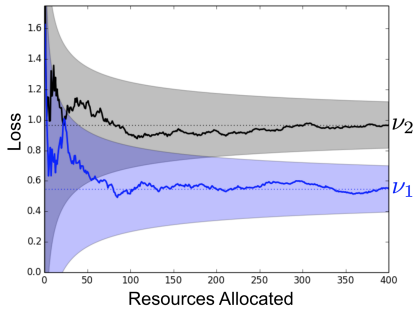


Figure 2.2: The validation loss as a function of total resources allocated for two configurations is shown. ν_1 and ν_2 represent the terminal validation losses at convergence. The shaded areas bound the maximum distance of the intermediate losses from the terminal validation loss and monotonically decrease with the resource.

are two takeaways from this observation: more resources are needed to differentiate between the two configurations when either (1) the envelope functions are wider or (2) the terminal losses are closer together.

However, in practice, the optimal allocation strategy is unknown because we do not have knowledge of the envelope functions nor the distribution of terminal losses. Hence, if more resources are required before configurations can differentiate themselves in terms of quality (e.g., if an iterative training method converges very slowly for a given data set or if randomly selected hyperparameter configurations perform similarly well), then it would be reasonable to work with a small number of configurations. In contrast, if the quality of a configuration is typically revealed after a small number of resources (e.g., if iterative training methods converge very quickly for a given data set or if randomly selected hyperparameter configurations are of low-quality with high probability), then n is the bottleneck and we should choose n to be large.

Certainly, if meta-data or previous experience suggests that a certain tradeoff is likely to work well in practice, one should exploit that information and allocate the majority of resources to that tradeoff. However, without this supplementary information, practitioners are forced to make this tradeoff, severely hindering the applicability of prior configuration evaluation methods.

2.3.2 HYPERBAND

HYPERBAND, shown in Algorithm 1, addresses this "n versus B/n" problem by considering several possible values of n for a fixed B, in essence performing a grid search over feasible value of n. Associated with each value of n is a minimum resource r that is allocated to all configurations before some are discarded; a larger value of n corresponds to a smaller r and hence more aggressive early-stopping. There are two components to HYPERBAND; (1) the inner loop invokes SUCCESSIVEHALVING for fixed values of n and r (lines 3–9) and (2) the outer loop iterates over different values of n and r (lines 1–2). We will refer to each such run of SUCCESSIVEHALVING within HYPERBAND as a "bracket." Each bracket is designed to use approximately B total resources and corresponds to a different tradeoff between n and B/n. Hence, a single execution of HYPERBAND takes a finite budget of $(s_{max} + 1)B$; we recommend repeating it indefinitely.

Algorithm 1: HYPERBAND algorithm for hyperparameter optimization.

input : R, η (default $\eta = 3$)
initialization : $s_{\max} = \lfloor \log_\eta(R) \rfloor$, $B = (s_{\max} + 1)R$
for $s \in \{0, 1, \ldots, s_{\max}\}$ **do**
  $n = \lceil \frac{B}{R} \frac{\eta^{s_{\max}-s}}{(s_{\max}-s+1)} \rceil$, $r = R\eta^{-(s_{\max}-s)}$
  // begin SuccessiveHalving with (n, r) inner loop
  $T =$get_hyperparameter_configuration(n)
  for $i \in \{0, \ldots, s_{\max} - s\}$ **do**
    $n_i = \lfloor n\eta^{-i} \rfloor$
    $r_i = r\eta^i$
    $L = \{\text{run\_then\_return\_val\_loss}(t, r_i) : t \in T\}$
    $T =$top_k$(T, L, \lfloor n_i/\eta \rfloor)$
  end
end
return *Configuration with the smallest intermediate loss seen so far.*

HYPERBAND requires two inputs (1) R, the maximum amount of resource that can be allocated to a single configuration, and (2) η, an input that controls the proportion of configurations discarded in each round of SUCCESSIVEHALVING. The two inputs dictate how many different brackets are considered; specifically, $s_{\max} + 1$ different values for n are considered with $s_{\max} = \lfloor \log_\eta(R) \rfloor$. HYPERBAND begins with the most aggressive bracket $s = 0$, which sets n to maximize exploration, subject to the constraint that at least one configuration is allocated R resources. Each subsequent bracket reduces n by a factor of approximately η until the final bracket, $s = s_{\max}$, in which every configuration is allocated R resources (this bracket simply performs classical random search). Hence, HYPERBAND performs a geometric search in the average budget per configuration and removes the need to select n for a fixed budget at the cost of approximately $s_{\max} + 1$ times more work than running SUCCESSIVEHALVING for a single value of n. By doing so, HYPERBAND is able to exploit situations in which adaptive allocation works well, while protecting itself in situations where more conservative allocations are required.

HYPERBAND requires the following methods to be defined for any given learning problem:

- get_hyperparameter_configuration(n) – a function that returns a set of n i.i.d. samples from some distribution defined over the hyperparameter configuration space. In this thesis, we assume hyperparameters are uniformly sampled from a predefined space (i.e., hypercube with min and max bounds for each hyperparameter), which immediately yields consistency guarantees. However, the more aligned the distribution is towards high quality hyperparameters (i.e., a useful prior), the better HYPERBAND will perform. Since the introduction of HYPERBAND, methods that combine HYPERBAND with adaptive hyperparameter selection have been developed [Falkner et al., 2018, Klein et al., 2017b]. We compare to some of these approaches in Chapter 3.
- run_then_return_val_loss(t, r) – a function that takes a hyperparameter configuration t and resource allocation r as input and returns the validation loss after training the configuration

for the allocated resources.

- $\text{top\_k}(\text{configs}, \text{losses}, k)$ – a function that takes a set of configurations as well as their associated losses and returns the top k performing configurations.

2.3.3 Different Types of Resources

Early-stopping with HYPERBAND can be performed with different notions of training resource.

- **Training Iterations** – Adaptively allocating the number of training iterations for an iterative algorithm like stochastic gradient decent is perhaps the application of hyperband that most naturally comes to mind.
- **Time** – Early-stopping in terms of time can be preferred when various hyperparameter configurations differ in training time and the practitioner's chief goal is to find a good hyperparameter setting in a fixed wall-clock time. For instance, training time could be used as a resource to quickly terminate straggler jobs in distributed computation environments.
- **Data Set Subsampling** – Here we consider the setting of a black-box batch training algorithm that takes a data set as input and outputs a model. In this setting, we treat the resource as the size of a random subset of the data set with R corresponding to the full data set size. Subsampling data set sizes using HYPERBAND, especially for problems with super-linear training times like kernel methods, can provide substantial speedups.
- **Feature Subsampling** – Random features or Nyström-like methods are popular methods for approximating kernels for machine learning applications [Rahimi and Recht, 2007]. In image processing, especially deep-learning applications, filters are usually sampled randomly, with the number of filters having an impact on the performance. Downsampling the number of features is a common tool used when hand-tuning hyperparameters; HYPERBAND can formalize this heuristic.

2.3.4 Setting R

The resource R and η (which we address next) are the only required inputs to HYPERBAND. As mentioned in Section 2.3.2, R represents the maximum amount of resources that can be allocated to any given configuration. In most cases, there is a natural upper bound on the maximum budget per configuration that is often dictated by the resource type (e.g., training set size for data set downsampling; limitations based on memory constraint for feature downsampling; rule of thumb regarding number of epochs when iteratively training neural networks). If there is a range of possible values for R, a smaller R will give a result faster (since the budget B for each bracket is a multiple of R), but a larger R will give a better guarantee of successfully differentiating between the configurations.

Moreover, for settings in which either R is unknown or not desired, we provide an infinite horizon version of HYPERBAND in Section 2.5. This version of the algorithm doubles the budget over time, $B \in \{2, 4, 8, 16, \ldots\}$, and for each B, tries all possible values of $n \in \left\{2^k : k \in \{1, \ldots, \log_2(B)\}\right\}$. For each combination of B and n, the algorithm runs an instance of the (infinite horizon) SUCCESSIVEHALVING algorithm, which implicitly sets $R = \frac{B}{2\log_2(n)}$, thereby

growing R as B increases. The main difference between the infinite horizon algorithm and Algorithm 1 is that the number of unique brackets grows over time instead of staying constant with each outer loop. We will analyze this version of HYPERBAND in more detail in Section 2.5 and use it as the launching point for the theoretical analysis of standard (finite horizon) HYPERBAND.

Note that R is also the number of configurations evaluated in the bracket that performs the most exploration, i.e $s = 0$. In practice one may want $n \leq n_{\max}$ to limit overhead associated with training many configurations on a small budget, i.e., costs associated with initialization, loading a model, and validation. In this case, only run brackets $s \geq s_{\max} - \lfloor \log_\eta(n_{\max}) \rfloor$. Alternatively, one can redefine one unit of resource so that R is artificially smaller (i.e., if the desired maximum iteration is 100k, defining one unit of resource to be 100 iterations will give $R = 1,000$, whereas defining one unit to be 1k iterations will give $R = 100$). Thus, one unit of resource can be interpreted as the minimum desired resource and R as the ratio between maximum resource and minimum resource.

2.3.5 Setting η

The value of η is a knob that can be tuned based on *practical* user constraints. Larger values of η correspond to more aggressive elimination schedules and thus fewer rounds of elimination; specifically, each round retains $1/\eta$ configurations for a total of $\lfloor \log_\eta(n) \rfloor + 1$ rounds of elimination with n configurations. If one wishes to receive a result faster at the cost of a sub-optimal asymptotic constant, one can increase η to reduce the budget per bracket $B = (\lfloor \log_\eta(R) \rfloor + 1)R$. We stress that results are not very sensitive to the choice of η. If our theoretical bounds are optimized (see Section 2.5), they suggest choosing $\eta = e \approx 2.718$, but in practice we suggest taking η to be equal to 3 or 4.

Tuning η will also change the number of brackets and consequently the number of different tradeoffs that HYPERBAND tries. Usually, the possible range of brackets is fairly constrained, since the number of brackets is logarithmic in R; namely, there are $(\lfloor \log_\eta(R) \rfloor + 1) = s_{\max} + 1$ brackets. For our experiments in Section 2.4, we chose η to provide 5 brackets for the specified R; for most problems, 5 is a reasonable number of n versus B/n tradeoffs to explore. However, for large R, using $\eta = 3$ or 4 can give more brackets than desired. The number of brackets can be controlled in a few ways. First, as mentioned in the previous section, if R is too large and overhead is an issue, then one may want to control the overhead by limiting the maximum number of configurations to $n_{\max}$. If overhead is not a concern and aggressive exploration is desired, one can (1) increase η to reduce the number of brackets while maintaining R as the maximum number of configurations in the most exploratory bracket, or (2) still use $\eta = 3$ or 4 but only try brackets that do a baseline level of exploration, i.e., set $n_{\min}$ and only try brackets from 0 to $s = s_{\max} - \lfloor \log_\eta(n_{\min}) \rfloor$. For computationally intensive problems that have long training times and high-dimensional search spaces, we recommend the latter. Intuitively, if the number of configurations that can be trained to completion (i.e., trained using R resources) in a reasonable amount of time is on the order of the dimension of the search space and not exponential in the dimension, then it will be impossible to find a good configuration without using an aggressive exploratory tradeoff between n and B/n.

2.3.6 Overview of Theoretical Results

The theoretical properties of HYPERBAND are best demonstrated through an example. Suppose there are n configurations, each with a given terminal validation error ν_i for $i = 1, \ldots, n$. Without loss of generality, index the configurations by performance so that ν_1 corresponds to the best performing configuration, ν_2 to the second best, and so on. Now consider the task of identifying the best configuration. The optimal strategy would allocate to each configuration i the minimum resource required to distinguish it from ν_1, i.e., enough so that the envelope functions (see Figure 2.2) bound the intermediate loss to be less than $\frac{\nu_i - \nu_1}{2}$ away from the terminal value. In contrast, the naive uniform allocation strategy, which allocates B/n to each configuration, has to allocate to every configuration the maximum resource required to distinguish any arm ν_i from ν_1. Remarkably, the budget required by SUCCESSIVEHALVING is only a small factor of the optimal because it capitalizes on configurations that are easy to distinguish from ν_1.

The relative size of the budget required for uniform allocation and SUCCESSIVEHALVING depends on the envelope functions bounding deviation from terminal losses as well as the distribution from which ν_i's are drawn. The budget required for SUCCESSIVEHALVING is smaller when the optimal n versus B/n tradeoff discussed in Section 2.3.1 requires fewer resources per configuration. Hence, if the envelope functions tighten quickly as a function of resource allocated, or the average distances between terminal losses is large, then SUCCESSIVEHALVING can be substantially faster than uniform allocation. These intuitions are formalized in Section 2.5 and associated theorems/corollaries are provided that take into account the envelope functions and the distribution from which ν_i's are drawn.

In practice, we do not have knowledge of either the envelope functions or the distribution of ν_i's, both of which are integral in characterizing SUCCESSIVEHALVING's required budget. With HYPERBAND we address this shortcoming by hedging our aggressiveness. We show in Section 2.5.3.3 that HYPERBAND, despite having no knowledge of the envelope functions nor the distribution of ν_i's, requires a budget that is only log factors larger than that of SUCCESSIVEHALVING.

2.4 Experiments

In this section, we evaluate the empirical behavior of HYPERBAND with three different resource types: iterations, data set subsamples, and feature samples. For all experiments, we compare HYPERBAND with three well known Bayesian optimization algorithms—SMAC, TPE, and Spearmint—using their default settings. We exclude Spearmint from the comparison set when there are conditional hyperparameters in the search space because it does not natively support them [Eggensperger et al., 2013]. We also show results for SUCCESSIVEHALVING corresponding to repeating the most exploratory bracket of HYPERBAND to provide a baseline for aggressive early-stopping.[4] Additionally, as standard baselines against which to measure all speedups, we consider random search and "random $2\times$," a variant of random search with twice the budget of other methods. Furthermore, we compare to a variant of SMAC using the early termination criterion

[4]This is not done for the experiments in Section 2.4.2.1, since the most aggressive bracket varies from dataset to dataset with the number of training points.

proposed by Domhan et al. [2015] in the deep learning experiments described in Section 2.4.1. Note that we compare HYPERBAND to more sophisticated hybrid methods [Falkner et al., 2018, Jaderberg et al., 2017, Klein et al., 2017a] in Chapter 3.4.

In the experiments below, we followed these loose guidelines when determining how to configuration HYPERBAND:

1. The maximum resource R should be reasonable given the problem, but ideally large enough so that early-stopping is beneficial.
2. η should depend on R and be selected to yield ≈ 5 brackets with a minimum of 3 brackets. This is to guarantee that HYPERBAND will use a baseline degree of early-stopping and prevent too coarse of a grid of n vs B tradeoffs.

2.4.1 Early-Stopping Iterative Algorithms for Deep Learning

For this benchmark, we tuned a convolutional neural network[5] with the same architecture as that used in Snoek et al. [2012] and Domhan et al. [2015]. The search spaces used in the two previous works differ, and we used a search space similar to that of Snoek et al. [2012] with 6 hyperparameters for stochastic gradient decent and 2 hyperparameters for the response normalization layers (see Section 2.6 for details). In line with the two previous works, we used a batch size of 100 for all experiments.

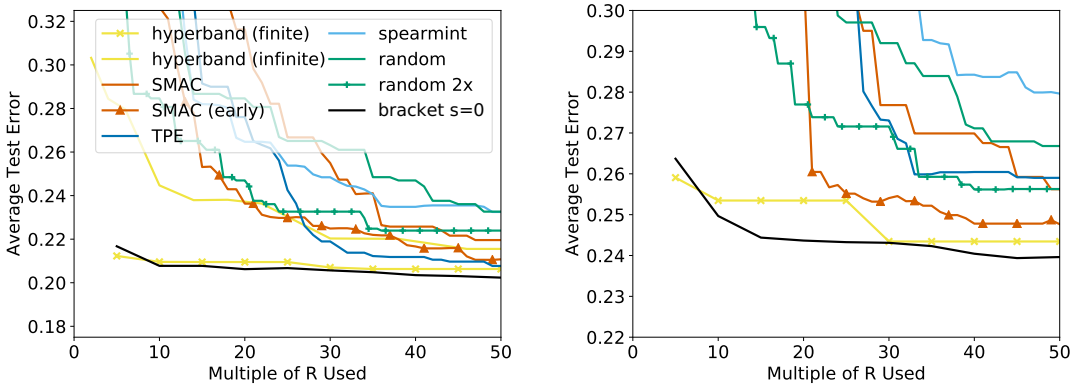
Data sets: We considered three image classification data sets: CIFAR-10 [Krizhevsky, 2009], rotated MNIST with background images (MRBI) [Larochelle et al., 2007], and Street View House Numbers (SVHN) [Netzer et al., 2011]. CIFAR-10 and SVHN contain 32×32 RGB images while MRBI contains 28×28 grayscale images. Each data set was split into a training, validation, and test set: (1) CIFAR-10 has 40k, 10k, and 10k instances; (2) MRBI has 10k, 2k, and 50k instances; and (3) SVHN has close to 600k, 6k, and 26k instances for training, validation, and test respectively. For all data sets, the only preprocessing performed on the raw images was demeaning.

HYPERBAND Configuration: For these experiments, one unit of resource corresponds to 100 mini-batch iterations (10k examples with a batch size of 100). For CIFAR-10 and MRBI, R was set to 300 (or 30k total iterations). For SVHN, R was set to 600 (or 60k total iterations) to accommodate the larger training set. Given R for these experiments, we set $\eta = 4$ to yield five SUCCESSIVEHALVING brackets for HYPERBAND.

Results: Each searcher was given a total budget of $50R$ per trial to return the best possible hyperparameter configuration. For HYPERBAND, the budget is sufficient to run the outer loop twice (for a total of 10 SUCCESSIVEHALVING brackets). For SMAC, TPE, and random search, the budget corresponds to training 50 different configurations to completion. Ten independent trials were performed for each searcher. The experiments took the equivalent of over 1 year of GPU hours on NVIDIA GRID K520 cards available on Amazon EC2 g2.8xlarge instances. We set a total budget constraint in terms of iterations instead of compute time to make comparisons hardware independent.[6] Comparing progress by iterations instead of time ignores overhead

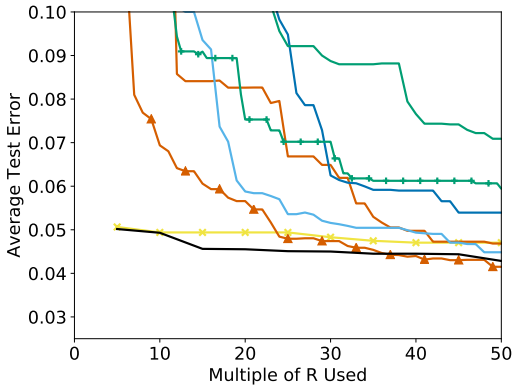
[5]The model specification is available at http://code.google.com/p/cuda-convnet/.

[6]Most trials were run on Amazon EC2 g2.8xlarge instances but a few trials were run on different machines due to the large computational demand of these experiments.



(a) CIFAR-10

(b) MRBI



(c) SVHN

Figure 2.3: Average test error across 10 trials. Label "SMAC (early)" corresponds to SMAC with the early-stopping criterion proposed in Domhan et al. [2015] and label "bracket $s = 0$" corresponds to repeating the most exploratory bracket of HYPERBAND.

costs, e.g. the cost of configuration selection for Bayesian methods and model initialization and validation costs for HYPERBAND. While overhead is hardware dependent, the overhead for HYPERBAND is below 5% on EC2 g2.8xlarge machines, so comparing progress by time passed would not change results significantly.

For CIFAR-10, the results in Figure 2.3(a) show that HYPERBAND is over an order-of-magnitude faster than its competitiors. For MRBI, HYPERBAND is over an order-of-magnitude faster than standard configuration selection approaches and 5× faster than SMAC (early). For SVHN, while HYPERBAND finds a good configuration faster, Bayesian optimization methods are competitive and SMAC (early) outperforms HYPERBAND. The performance of SMAC (early) demonstrates there is merit to combining early-stopping and adaptive configuration selection.

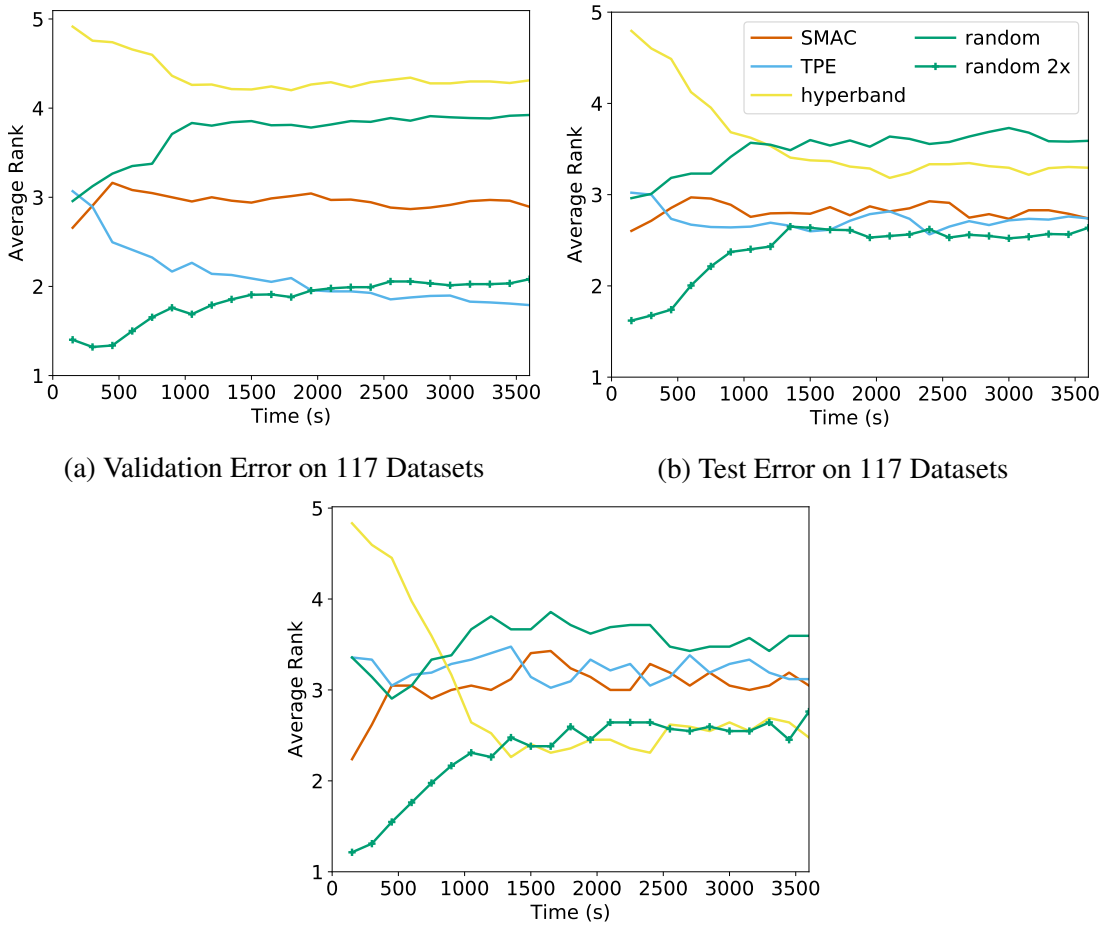
Across the three data sets, HYPERBAND and SMAC (early) are the only two methods that consistently outperform random $2\times$. On these data sets, HYPERBAND is over $20\times$ faster than random search while SMAC (early) is $\leq 7\times$ faster than random search within the evaluation window. In fact, the first result returned by HYPERBAND after using a budget of $5R$ is often competitive with results returned by other searchers after using $50R$. Additionally, HYPERBAND is less variable than other searchers across trials, which is highly desirable in practice (see Section 2.6 for plots with error bars).

As discussed in Section 2.3.5, for computationally expensive problems in high-dimensional search spaces, it may make sense to just repeat the most exploratory brackets. Similarly, if meta-data is available about a problem or it is known that the quality of a configuration is evident after allocating a small amount of resource, then one should just repeat the most exploratory bracket. Indeed, for these experiments, bracket $s = 0$ vastly outperforms all other methods on CIFAR-10 and MRBI and is nearly tied with SMAC (early) for first on SVHN.

While we set R for these experiments to facilitate comparison to Bayesian methods and random search, it is also reasonable to use infinite horizon HYPERBAND to grow the maximum resource until a desired level of performance is reached. We evaluate infinite horizon HYPERBAND on CIFAR-10 using $\eta = 4$ and a starting budget of $B = 2R$. Figure 2.3(a) shows that infinite horizon HYPERBAND is competitive with other methods but does not perform as well as finite horizon HYPERBAND within the $50R$ budget limit. The infinite horizon algorithm underperforms initially because it has to tune the maximum resource R as well and starts with a less aggressive early-stopping rate. This demonstrates that in scenarios where a max resource is known, it is better to use the finite horizon algorithm. Hence, we focus on the finite horizon version of HYPERBAND for the remainder of our empirical studies.

Finally, CIFAR-10 is a very popular data set and state-of-the-art models achieve much lower error rates than what is shown in Figure 2.3. The difference in performance is mainly attributable to higher model complexities and data manipulation (i.e. using reflection or random cropping to artificially increase the data set size). If we limit the comparison to published results that use the same architecture and exclude data manipulation, the best human expert result for the data set is 18% error and the best hyperparameter optimized results are 15.0% for Snoek et al. [2012][7] and 17.2% for Domhan et al. [2015]. These results exceed ours on CIFAR-10 because they train on 25% more data, by including the validation set, and also train for more epochs. When we train the

[7]We were unable to reproduce this result even after receiving the optimal hyperparameters from the authors through a personal communication.



(a) Validation Error on 117 Datasets

(b) Test Error on 117 Datasets

(c) Test Error on 21 Datasets

Figure 2.4: Average rank across all data sets for each searcher. For each data set, the searchers are ranked according to the average validation/test error across 20 trials.

best model found by HYPERBAND on the combined training and validation data for 300 epochs, the model achieved a test error of 17.0%.

2.4.2 Data Set Subsampling

We studied two different hyperparameter search optimization problems for which HYPERBAND uses data set subsamples as the resource. The first adopts an extensive framework presented in Feurer et al. [2015a] that attempts to automate preprocessing and model selection. Due to certain limitations of the framework that fundamentally limited the impact of data set downsampling, we conducted a second experiment using a kernel classification task.

2.4.2.1 117 Data Sets

We used the framework introduced by Feurer et al. [2015a], which explored a structured hyperparameter search space comprised of 15 classifiers, 14 feature preprocessing methods, and 4 data preprocessing methods for a total of 110 hyperparameters. We excluded the meta-learning component introduced in Feurer et al. [2015a] used to warmstart Bayesian methods with promising configurations, in order to perform a fair comparison with random search and HYPERBAND. Similar to Feurer et al. [2015a], we imposed a 3GB memory limit, a 6-minute timeout for each hyperparameter configuration and a one-hour time window to evaluate each searcher on each data set. Twenty trials of each searcher were performed per data set and all trials in aggregate took over a year of CPU time on n1-standard-1 instances from Google Cloud Compute. Additional details about our experimental framework are available in Section 2.6.

Data sets: Feurer et al. [2015a] used 140 binary and multiclass classification data sets from OpenML, but 23 of them are incompatible with the latest version of the OpenML plugin [Feurer, 2015], so we worked with the remaining 117 data sets. Due to the limitations of the experimental setup (discussed in Section 2.6), we also separately considered 21 of these data sets, which demonstrated at least modest (though still sublinear) training speedups due to subsampling. Specifically, each of these 21 data sets showed on average at least a $3\times$ speedup due to $8\times$ downsampling on 100 randomly selected hyperparameter configurations.

HYPERBAND Configuration: Due to the wide range of dataset sizes, with some datasets having fewer than 10k training points, we ran HYPERBAND with $\eta = 3$ to allow for at least 3 brackets without being overly aggressive in downsampling on small datasets. R was set to the full training set size for each data set and the maximum number of configurations for any bracket of SUCCESSIVEHALVING was limited to $n_{\max} = \max\{9, R/1000\}$. This ensured that the most exploratory bracket of HYPERBAND will downsample at least twice. As mentioned in Section 2.3.5, when $n_{\max}$ is specified, the only difference when running the algorithm is $s_{\max} = \lfloor \log_\eta(n_{\max}) \rfloor$ instead of $\lfloor \log_\eta(R) \rfloor$.

Results: The results on all 117 data sets in

Figure 2.4(a,b) show that HYPERBAND outperforms random search in test error rank despite performing worse in validation error rank. Bayesian methods outperform HYPERBAND and random search in test error performance but also exhibit signs of overfitting to the validation set, as they outperform HYPERBAND by a larger margin on the validation error rank. Notably, random $2\times$ outperforms all other methods. However, for the subset of 21 data sets, Figure 2.4(c) shows that HYPERBAND outperforms all other searchers on test error rank, including random $2\times$ by a very small margin. While these results are more promising, the effectiveness of HYPERBAND was restricted in this experimental framework; for smaller data sets, the startup overhead was high relative to total training time, while for larger data sets, only a handful of configurations could be trained within the hour window.

We note that while average rank plots like those in Figure 2.4 are an effective way to aggregate information across many searchers and data sets, they provide no indication about the *magnitude* of the differences between the performance of the methods. Figure 2.5, which charts the difference between the test error for each searcher and that of random search across all 117 datasets, highlights the small difference in the magnitude of the test errors across searchers.

These results are not surprising; as mentioned in Section 2.2.1, vanilla Bayesian optimization

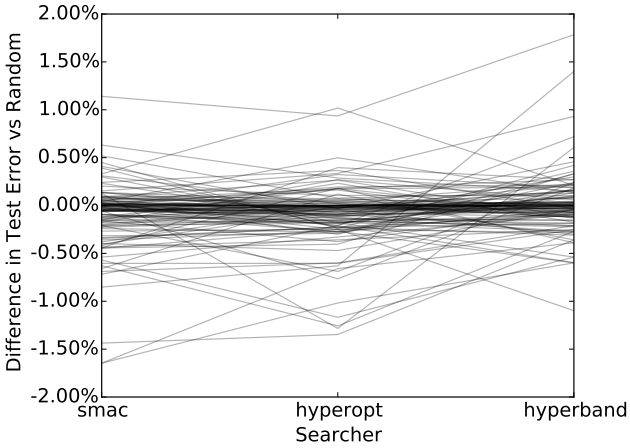


Figure 2.5: Each line plots, for a single data set, the difference in test error versus random search for each search, where lower is better. Nearly all the lines fall within the -0.5% and 0.5% band and, with the exception of a few outliers, the lines are mostly flat.

methods perform similarly to random search in high-dimensional search spaces. Feurer et al. [2015a] showed that using meta-learning to warmstart Bayesian optimization methods improved performance in this high-dimensional setting. Using meta-learning to identify a promising distribution from which to sample configurations as input into HYPERBAND is a direction for future work.

2.4.2.2 Kernel Regularized Least Squares Classification

For this benchmark, we tuned the hyperparameters of a kernel-based classifier on CIFAR-10. We used the multi-class regularized least squares classification model, which is known to have comparable performance to SVMs [Agarwal et al., 2014, Rifkin and Klautau, 2004] but can be trained significantly faster.[8] The hyperparameters considered in the search space include preprocessing method, regularization, kernel type, kernel length scale, and other kernel specific hyperparameters (see Section 2.6 for more details). For HYPERBAND, we set $R = 400$, with each unit of resource representing 100 datapoints, and $\eta = 4$ to yield a total of 5 brackets. Each hyperparameter optimization algorithm was run for ten trials on Amazon EC2 $\mathrm{m4.2xlarge}$ instances; for a given trial, HYPERBAND was allowed to run for two outer loops, bracket $s = 0$ was repeated 10 times, and all other searchers were run for 12 hours.

Figure 2.6 shows that HYPERBAND returned a good configuration after completing the first SUCCESSIVEHALVING bracket in approximately 20 minutes; other searchers failed to reach this error rate on average even after the entire 12 hours. Notably, HYPERBAND was able to evaluate over 250 configurations in this first bracket of SUCCESSIVEHALVING, while competitors were able to evaluate only three configurations in the same amount of time. Consequently, HYPERBAND is over 30$\times$ faster than Bayesian optimization methods and 70$\times$ faster than random

[8]The default SVM method in Scikit-learn is single core and takes hours to train on CIFAR-10, whereas a block coordinate descent least squares solver takes less than 10 minutes on an 8 core machine.

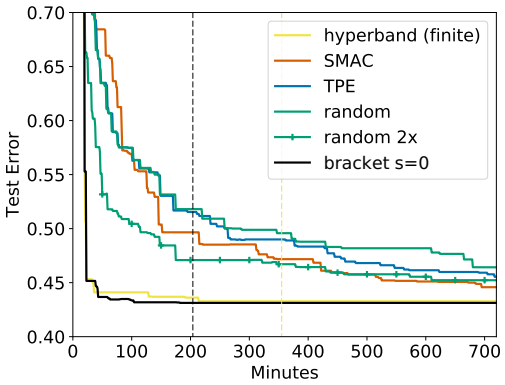


Figure 2.6: Average test error of the best kernel regularized least square classification model found by each searcher on CIFAR-10. The color coded dashed lines indicate when the last trial of a given searcher finished.

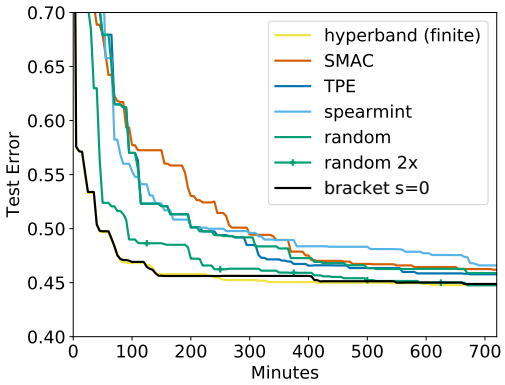


Figure 2.7: Average test error of the best random features model found by each searcher on CIFAR-10. The test error for HYPERBAND and bracket $s = 0$ are calculated in every evaluation instead of at the end of a bracket.

search. Bracket $s = 0$ sightly outperforms HYPERBAND but the terminal performance for the two algorithms are the same. Random $2\times$ is competitive with SMAC and TPE.

2.4.3 Feature Subsampling to Speed Up Approximate Kernel Classification

Next, we examine the performance of HYPERBAND when using features as a resource on a random feature kernel approximations task. Features were randomly generated using the method described in Rahimi and Recht [2007] to approximate the RBF kernel, and these random features were then used as inputs to a ridge regression classifier. The hyperparameter search space included the preprocessing method, kernel length scale, and L_2 penalty. While it may seem natural to use infinite horizon HYPERBAND, since the fidelity of the approximation improves with more random features, in practice, the amount of available machine memory imposes a natural upper bound on the number of features. Thus, we used finite horizion HYPERBAND with a maximum resource of 100k random features, which comfortably fit into a machine with 60GB of memory. Additionally, we set one unit of resource to be 100 features, so $R = 1000$. Again, we set $\eta = 4$ to yield 5 brackets of SUCCESSIVEHALVING. We ran 10 trials of each searcher, with each trial lasting 12 hours on a n1-standard-16 machine from Google Cloud Compute. The results in Figure 2.7 show that HYPERBAND is around $6\times$ faster than Bayesian methods and random search. HYPERBAND performs similarly to bracket $s = 0$. Random $2\times$ outperforms Bayesian optimization algorithms.

2.4.4 Experimental Discussion

While our experimental results show HYPERBAND is a promising algorithm for hyperparameter optimization, a few questions naturally arise:

1. What impacts the speedups provided by HYPERBAND?

2. Why does SUCCESSIVEHALVING seem to outperform HYPERBAND?
3. What about hyperparameters that should depend on the resource?

We next address each of these questions in turn.

2.4.4.1 Factors Impacting the Performance of HYPERBAND

For a given R, the most exploratory SUCCESSIVEHALVING round performed by HYPERBAND evaluates R configurations using a budget of $(\lfloor \log_\eta(R) \rfloor + 1)R$, which gives an upper bound on the potential speedup over random search. If training time scales linearly with the resource, the maximum speedup offered by HYPERBAND compared to random search is $\frac{R}{(\lfloor \log_\eta(R) \rfloor + 1)}$. For the values of η and R used in our experiments, the maximum speedup over random search is approximately $50\times$ given linear training time. However, we observe a range of speedups from $6\times$ to $70\times$ faster than random search. The differences in realized speedup can be explained by three factors:

1. *How training time scales with the given resource*. In cases where training time is superlinear as a function of the resource, HYPERBAND can offer higher speedups. For instance, if training scales like a polynomial of degree $p > 1$, the maximum speedup for HYPERBAND over random search is approximately $\frac{\eta^p - 1}{\eta^{p-1}} R$. In the kernel least square classifier experiment discussed in Section 2.4.2.2, the training time scaled quadratically as a function of the resource, which explains why the realized speedup of $70\times$ is higher than the maximum expected speedup given linear scaling.
2. *Overhead costs associated with training*. Total evaluation time also depends on fixed overhead costs associated with evaluating each hyperparameter configuration, e.g., initializing a model, resuming previously trained models, and calculating validation error. For example, in the downsampling experiments on 117 data sets presented in Section 2.4.2.1, HYPERBAND did not provide significant speedup because many data sets could be trained in a matter of a few seconds and the initialization cost was high relative to training time.
3. *The difficulty of finding a good configuration*. Hyperparameter optimization problems can vary in difficulty. For instance, an 'easy' problem is one where a randomly sampled configuration is likely to result in a high-quality model, and thus we only need to evaluate a small number of configurations to find a good setting. In contrast, a 'hard' problem is one where an arbitrary configuration is likely to be bad, in which case many configurations must be considered. HYPERBAND leverages downsampling to boost the number of configurations that are evaluated, and thus is better suited for 'hard' problems where more evaluations are actually necessary to find a good setting. Generally, the difficulty of a problem scales with the dimensionality of the search space. For low-dimensional problems, the number of configurations evaluated by random search and Bayesian methods is exponential in the number of dimensions so good coverage can be achieved. For instance, the low-dimensional ($d = 3$) search space in our feature subsampling experiment in Section 2.4.3 helps explain why HYPERBAND is only $6\times$ faster than random search. In contrast, for the neural network experiments in Section 2.4.1, we hypothesize that faster speedups are observed for HYPERBAND because the dimension of the search space is higher.

2.4.4.2 Comparison to SUCCESSIVEHALVING

With the exception of the 117 Datasets experiment (Section 2.4.2.1), the most aggressive bracket of SUCCESSIVEHALVING outperformed HYPERBAND in all of our experiments. In hindsight, we should have just run bracket $s = 0$, since aggressive early-stopping provides massive speedups on many of these benchmarking tasks. However, as previously mentioned, it was unknown a priori that bracket $s = 0$ would perform the best and that is why we have to cycle through all possible brackets with HYPERBAND. Another question is what happens when one increases s even further, i.e. instead of 4 rounds of elimination, why not 5 or even more with the same maximum resource R? In our case, $s = 0$ was the most aggressive bracket we could run given the minimum resource per configuration limits imposed for the previous experiments. However, for larger data sets, it is possible to extend the range of possible values for s, in which case, HYPERBAND may either provide even faster speedups if more aggressive early-stopping helps or be slower by a small factor if the most aggressive brackets are essentially throwaways.

We believe prior knowledge about a task can be particularly useful for limiting the range of brackets explored by HYPERBAND. In our experience, aggressive early-stopping is generally safe for neural network tasks and even more aggressive early-stopping may be reasonable for larger data sets and longer training horizons. However, when pushing the degree of early-stopping by increasing s, one has to consider the additional overhead cost associated with examining more models. Hence, one way to leverage meta-learning would be to use learning curve convergence rate, difficulty of different search spaces, and overhead costs of related tasks to determine the brackets considered by HYPERBAND.

2.4.4.3 Resource Dependent Hyperparameters

In certain cases, the setting for a given hyperparameter should depend on the allocated resource. For example, the maximum tree depth regularization hyperparameter for random forests should be higher with more data and more features. However, the optimal tradeoff between maximum tree depth and the resource is unknown and can be data set specific. In these situations, the rate of convergence to the true loss is usually slow because the performance on a smaller resource is not indicative of that on a larger resource. Hence, these problems are particularly difficult for HYPERBAND, since the benefit of early-stopping can be muted. Again, while HYPERBAND will only be a small factor slower than that of SUCCESSIVEHALVING with the optimal early-stopping rate, we recommend removing the dependence of the hyperparameter on the resource if possible. For the random forest example, an alternative regularization hyperparameter is minimum samples per leaf, which is less dependent on the training set size. Additionally, the dependence can oftentimes be removed with simple normalization. For example, the regularization term for our kernel least squares experiments were normalized by the training set size to maintain a constant tradeoff between the mean-squared error and the regularization term.

2.5 Theory

In this section, we introduce the pure-exploration non-stochastic infinite-armed bandit (NIAB) problem, a very general setting which encompasses our hyperparameter optimization problem

of interest. As we will show, HYPERBAND is in fact applicable to problems far beyond just hyperparameter optimization. We begin by formalizing the hyperparameter optimization problem and then reducing it to the pure-exploration NIAB problem. We subsequently present a detailed analysis of HYPERBAND in both the infinite and finite horizon settings.

2.5.1 Hyperparameter Optimization Problem Statement

Let $\mathcal{X}$ denote the space of valid hyperparameter configurations, which could include continuous, discrete, or categorical variables that can be constrained with respect to each other in arbitrary ways (i.e. $\mathcal{X}$ need not be limited to a subset of $[0,1]^d$). For $k = 1, 2, \ldots$ let $\ell_k \colon \mathcal{X} \to [0,1]$ be a sequence of loss functions defined over $\mathcal{X}$. For any hyperparameter configuration $x \in \mathcal{X}$, $\ell_k(x)$ represents the validation error of the model trained using x with k units of resources (e.g. iterations). In addition, for some $R \in \mathbb{N} \cup \{\infty\}$, define $\ell_* = \lim_{k \to R} \ell_k$ and $\nu_* = \inf_{x \in \mathcal{X}} \ell_*(x)$. Note that $\ell_k(\cdot)$ for all $k \in \mathbb{N}$, $\ell_*(\cdot)$, and ν_* are all unknown to the algorithm a priori. In particular, it is uncertain how quickly $\ell_k(x)$ varies as a function of x for any fixed k, and how quickly $\ell_k(x) \to \ell_*(x)$ as a function of k for any fixed $x \in \mathcal{X}$.

We assume hyperparameter configurations are sampled randomly from a known probability distribution $p(x) \colon \mathcal{X} \to [0, \infty)$, with support on $\mathcal{X}$. In our experiments, $p(x)$ is simply the uniform distribution, but the algorithm can be used with any sampling method. If $X \in \mathcal{X}$ is a random sample from this probability distribution, then $\ell_*(X)$ is a random variable whose distribution is unknown since $\ell_*(\cdot)$ is unknown. Additionally, since it is unknown how $\ell_k(x)$ varies as a function of x or k, one cannot necessarily infer anything about $\ell_k(x)$ given knowledge of $\ell_j(y)$ for any $j \in \mathbb{N}$, $y \in \mathcal{X}$. As a consequence, we reduce the hyperparmeter optimization problem down to a much simpler problem that ignores all underlying structure of the hyperparameters: we only interact with some $x \in \mathcal{X}$ through its loss sequence $\ell_k(x)$ for $k = 1, 2, \ldots$. With this reduction, the particular value of $x \in \mathcal{X}$ does nothing more than index or uniquely identify the loss sequence.

Without knowledge of how fast $\ell_k(\cdot) \to \ell_*(\cdot)$ or how $\ell_*(X)$ is distributed, the goal of HYPERBAND is to identify a hyperparameter configuration $x \in \mathcal{X}$ that minimizes $\ell_*(x) - \nu_*$ by drawing as many random configurations as desired while using as few total resources as possible.

2.5.2 The Pure-Exploration Non-stochastic Infinite-Armed Bandit Problem

We now formally define the bandit problem of interest, and relate it to the problem of hyperparameter optimization. Each "arm" in the NIAB game is associated with a sequence that is drawn randomly from a distribution over sequences. If we "pull" the ith drawn arm exactly k times, we observe a loss $\ell_{i,k}$. At each time, the player can either draw a new arm (sequence) or pull a previously drawn arm an additional time. There is no limit on the number of arms that can be drawn. We assume the arms are identifiable only by their index i (i.e. we have no side-knowledge or feature representation of an arm), and we also make the following two additional assumptions:

Assumption 1. *For each $i \in \mathbb{N}$ the limit $\lim_{k \to \infty} \ell_{i,k}$ exists and is equal to ν_i.*[9]

[9] We can always define $\ell_{i,k}$ so that convergence is guaranteed, i.e. taking the infimum of a sequence.

Assumption 2. *Each ν_i is a bounded i.i.d. random variable with cumulative distribution function F.*

The objective of the NIAB problem is to identify an arm $\hat{\imath}$ with small $\nu_{\hat{\imath}}$ using as few total pulls as possible. We are interested in characterizing $\nu_{\hat{\imath}}$ as a function of the total number of pulls from all the arms. Clearly, the hyperparameter optimization problem described above is an instance of the NIAB problem. In this case, arm i correspondes to a configuration $x_i \in \mathcal{X}$, with $\ell_{i,k} = \ell_k(x_i)$; Assumption 1 is equivalent to requiring that $\nu_i = \ell_*(x_i)$ exists; and Assumption 2 follows from the fact that the arms are drawn i.i.d. from $\mathcal{X}$ according to distribution function $p(x)$. F is simply the cumulative distribution function of $\ell_*(X)$, where X is a random variable drawn from the distribution $p(x)$ over $\mathcal{X}$. Note that since the arm draws are independent, the ν_i's are also independent. Again, this is not to say that the validation losses do not depend on the settings of the hyperparameters; the validation loss could well be correlated with certain hyperparameters, but this is not used in the algorithm and no assumptions are made regarding the correlation structure.

In order to analyze the behavior of HYPERBAND in the NIAB setting, we must define a few additional objects. Let $\nu_* = \inf\{m : \mathbb{P}(\nu \leq m) > 0\} > -\infty$, since the domain of the distribution F is bounded. Hence, the cumulative distribution function F satisfies

$$\mathbb{P}(\nu_i - \nu_* \leq \epsilon) = F(\nu_* + \epsilon) \tag{2.2}$$

and let $F^{-1}(y) = \inf_x\{x : F(x) \leq y\}$. Define $\gamma\colon \mathbb{N} \to \mathbb{R}$ as the pointwise smallest, monotonically decreasing function satisfying

$$\sup_i |\ell_{i,j} - \ell_{i,*}| \leq \gamma(j)\ , \quad \forall j \in \mathbb{N}. \tag{2.3}$$

The function γ is guaranteed to exist by Assumption 1 and bounds the deviation from the limit value as the sequence of iterates j increases. For hyperparameter optimization, this follows from the fact that ℓ_k uniformly converges to ℓ_* for all $x \in \mathcal{X}$. In addition, γ can be interpretted as the deviation of the validation error of a configuration trained on a subset of resources versus the maximum number of allocatable resources. Finally, define R as the first index such that $\gamma(R) = 0$ if it exists, otherwise set $R = \infty$. For $y \geq 0$ let $\gamma^{-1}(y) = \min\{j \in \mathbb{N}\colon \gamma(j) \leq y\}$, using the convention that $\gamma^{-1}(0) := R$ which we recall can be infinite.

As previously discussed, there are many real-world scenarios in which R is finite and known. For instance, if increasing subsets of the full data set is used as a resource, then the maximum number of resources cannot exceed the full data set size, and thus $\gamma(k) = 0$ for all $k \geq R$ where R is the (known) full size of the data set. In other cases such as iterative training problems, one might not want to or know how to bound R. We separate these two settings into the *finite horizon* setting where R is finite and known, and the *infinite horizon* setting where no bound on R is known and it is assumed to be infinite. While our empirical results suggest that the finite horizon may be more practically relevant for the problem of hyperparameter optimization, the infinite horizon case has natural connections to the literature, and we begin by analyzing this setting.

2.5.3 Infinite Horizon Setting ($R = \infty$)

Consider the HYPERBAND algorithm of Figure 2.8. The algorithm uses SUCCESSIVEHALVING (Figure 2.8) as a subroutine that takes a finite set of arms as input and outputs an estimate of the

SUCCESSIVEHALVING (Infinite horizon)

Input: Budget B, n arms where $\ell_{i,k}$ denotes the kth loss from the ith arm

Initialize: $S_0 = [n]$.

For $k = 0, 1, \ldots, \lceil \log_2(n) \rceil - 1$

Pull each arm in S_k for $r_k = \lfloor \frac{B}{|S_k| \lceil \log_2(n) \rceil} \rfloor$ times.

Keep the best $\lfloor |S_k|/2 \rfloor$ arms in terms of the r_kth observed loss as S_{k+1}.

Output : $\hat{\imath}$, $\ell_{\hat{\imath}, \lfloor \frac{B/2}{\lceil \log_2(n) \rceil} \rfloor}$ where $\hat{\imath} = S_{\lceil \log_2(n) \rceil}$

HYPERBAND (Infinite horizon)

Input: None

For $k = 1, 2, \ldots$

For $s \in \mathbb{N}$ s.t. $k - s \geq \log_2(s)$

$B_{k,s} = 2^k$, $n_{k,s} = 2^s$

$\hat{\imath}_{k,s}, \ell_{\hat{\imath}_{k,s}, \lfloor \frac{2^{k-1}}{s} \rfloor} \leftarrow$ SUCCESSIVEHALVING $(B_{k,s}, n_{k,s})$

Figure 2.8: (Top) The SUCCESSIVEHALVING algorithm proposed and analyzed in Jamieson and Talwalkar [2015] for the non-stochastic setting. Note this algorithm was originally proposed for the stochastic setting in Karnin et al. [2013]. (Bottom) The HYPERBAND algorithm for the infinite horizon setting. HYPERBAND calls SUCCESSIVEHALVING as a subroutine.

best performing arm in the set. We first analyze SUCCESSIVEHALVING (SHA) for a given set of limits ν_i and then consider the performance of SHA when ν_i are drawn randomly according to F. We then analyze the HYPERBAND algorithm. We note that the algorithm of Figure 2.8 was originally proposed by Karnin et al. [2013] for the stochastic setting. However, Jamieson and Talwalkar [2015] analyzed it in the non-stochastic setting and also found it to work well in practice. Extending the result of Jamieson and Talwalkar [2015] we have the following theorem:

Theorem 1. *Fix n arms. Let $\nu_i = \lim_{\tau \to \infty} \ell_{i,\tau}$ and assume $\nu_1 \leq \cdots \leq \nu_n$. For any $\epsilon > 0$ let*

$$\begin{aligned} z_{SHA} &= 2\lceil \log_2(n) \rceil \max_{i=2,\ldots,n} i \left(1 + \gamma^{-1}\left(\max\left\{\tfrac{\epsilon}{4}, \tfrac{\nu_i - \nu_1}{2}\right\}\right)\right) \\ &\leq 2\lceil \log_2(n) \rceil \left(n + \sum_{i=1,\ldots,n} \gamma^{-1}\left(\max\left\{\tfrac{\epsilon}{4}, \tfrac{\nu_i - \nu_1}{2}\right\}\right)\right) \end{aligned}$$

If the SUCCESSIVEHALVING *algorithm of Figure 2.8 is run with any budget $B > z_{SHA}$ then an arm $\hat{\imath}$ is returned that satisfies $\nu_i - \nu_1 \leq \epsilon/2$. Moreover, $|\ell_{\hat{\imath}, \lfloor \frac{B/2}{\lceil \log_2(n) \rceil} \rfloor} - \nu_1| \leq \epsilon$.*

The next technical lemma will be used to characterize $\sum_{i=1,\ldots,n} \gamma^{-1}\left(\max\left\{\frac{\epsilon}{4}, \frac{\nu_i - \nu_1}{2}\right\}\right)$, a problem dependent term that captures the effectiveness of early-stopping when the sequences are drawn from a probability distribution.

Lemma 2. *Fix $\delta \in (0, 1)$. Let $p_n = \frac{\log(2/\delta)}{n}$. For any $\epsilon \geq 4(F^{-1}(p_n) - \nu_*)$ define*

$$\mathbf{H}(F, \gamma, n, \delta, \epsilon) := 2n \int_{\nu_* + \epsilon/4}^{\infty} \gamma^{-1}\left(\tfrac{t - \nu_*}{4}\right) dF(t) + \left(\tfrac{4}{3}\log(2/\delta) + 2nF(\nu_* + \epsilon/4)\right) \gamma^{-1}\left(\tfrac{\epsilon}{16}\right)$$

and $\mathbf{H}(F,\gamma,n,\delta) := \mathbf{H}(F,\gamma,n,\delta,4(F^{-1}(p_n)-\nu_*))$ *so that*

$$\mathbf{H}(F,\gamma,n,\delta) = 2n\int_{p_n}^{1}\gamma^{-1}(\tfrac{F^{-1}(t)-\nu_*}{4})dt + \tfrac{10}{3}\log(2/\delta)\gamma^{-1}\left(\tfrac{F^{-1}(p_n)-\nu_*}{4}\right).$$

For n *arms with limits* $\nu_1 \le \cdots \le \nu_n$ *drawn from* F*, then*

$$\nu_1 \le F^{-1}(p_n) \quad \textit{and} \quad \sum_{i=1}^{n}\gamma^{-1}\left(\max\left\{\tfrac{\epsilon}{4}, \tfrac{\nu_i-\nu_1}{2}\right\}\right) \le \mathbf{H}(F,\gamma,n,\delta,\epsilon)$$

for any $\epsilon \ge 4(F^{-1}(p_n)-\nu_*)$ *with probability at least* $1-\delta$*.*

Setting $\epsilon = 4(F^{-1}(p_n)-\nu_*)$ in Theorem 1 and using the result of Lemma 2 that $\nu_* \le \nu_1 \le \nu_* + (F^{-1}(p_n)-\nu_*)$, we immediately obtain the following corollary.

Corollary 3. *Fix* $\delta \in (0,1)$ *and* $\epsilon \ge 4(F^{-1}(\frac{\log(2/\delta)}{n})-\nu_*)$*. Let* $B = 4\lceil\log_2(n)\rceil\mathbf{H}(F,\gamma,n,\delta,\epsilon)$ *where* $\mathbf{H}(F,\gamma,n,\delta,\epsilon)$ *is defined in Lemma 2. If the* SUCCESSIVEHALVING *algorithm of Figure 2.8 is run with the specified* B *and* n *arm configurations drawn randomly according to* F*, then an arm* $\hat{\imath} \in [n]$ *is returned such that with probability at least* $1-\delta$ *we have* $\nu_{\hat{\imath}} - \nu_* \le \left(F^{-1}(\frac{\log(2/\delta)}{n}) - \nu_*\right) + \epsilon/2$*. In particular, if* $B = 4\lceil\log_2(n)\rceil\mathbf{H}(F,\gamma,n,\delta)$ *and* $\epsilon = 4(F^{-1}(\frac{\log(2/\delta)}{n})-\nu_*)$ *then* $\nu_{\hat{\imath}} - \nu_* \le 3\left(F^{-1}(\frac{\log(2/\delta)}{n}) - \nu_*\right)$ *with probability at least* $1-\delta$*.*

Note that for any fixed $n \in \mathbb{N}$ we have for any $\Delta > 0$

$$\mathbb{P}(\min_{i=1,\ldots,n}\nu_i - \nu_* \ge \Delta) = (1-F(\nu_*+\Delta))^n \approx e^{-nF(\nu_*+\Delta)}$$

which implies $\mathbb{E}[\min_{i=1,\ldots,n}\nu_i - \nu_*] \approx F^{-1}(\frac{1}{n}) - \nu_*$. That is, n needs to be sufficiently large so that it is probable that a good limit is sampled. On the other hand, for any fixed n, Corollary 3 suggests that the total resource budget B needs to be large enough in order to overcome the rates of convergence of the sequences described by γ. Next, we relate SHA to a naive approach that uniformly allocates resources to a fixed set of n arms.

2.5.3.1 Non-Adaptive Uniform Allocation

The non-adaptive uniform allocation strategy takes as inputs a budget B and n arms, allocates B/n to each of the arms, and picks the arm with the lowest loss. The following results allow us to compare with SUCCESSIVEHALVING.

Proposition 4. *Suppose we draw* n *random configurations from* F*, train each with* $j = \min\{B/n, R\}$ *iterations, and let* $\hat{\imath} = \arg\min_{i=1,\ldots,n}\ell_j(X_i)$*. Without loss of generality assume* $\nu_1 \le \ldots \le \nu_n$*. If*

$$B \ge n\gamma^{-1}\left(\tfrac{1}{2}(F^{-1}(\tfrac{\log(1/\delta)}{n}) - \nu_*)\right) \tag{2.4}$$

then with probability at least $1-\delta$ *we have* $\nu_{\hat{\imath}} - \nu_* \le 2\left(F^{-1}\left(\frac{\log(1/\delta)}{n}\right) - \nu_*\right)$*. In contrast, there exists a sequence of functions* ℓ_j *that satisfy* F *and* γ *such that if*

$$B \le n\gamma^{-1}\left(2(F^{-1}(\tfrac{\log(c/\delta)}{n+\log(c/\delta)}) - \nu_*)\right)$$

then with probability at least δ*, we have* $\nu_{\hat{\imath}} - \nu_* \ge 2(F^{-1}(\frac{\log(c/\delta)}{n+\log(c/\delta)}) - \nu_*)$*, where* c *is a constant that depends on the regularity of* F*.*

For any fixed n and sufficiently large B, Corollary 3 shows that SUCCESSIVEHALVING outputs an $\hat{\imath} \in [n]$ that satisfies $\nu_{\hat{\imath}} - \nu_* \lesssim F^{-1}(\frac{\log(2/\delta)}{n}) - \nu_*$ with probability at least $1 - \delta$. This guarantee is similar to the result in Proposition 4. However, SUCCESSIVEHALVING achieves its guarantee as long as[10]

$$B \simeq \log_2(n) \left[\log(1/\delta)\gamma^{-1}\left(F^{-1}(\tfrac{\log(1/\delta)}{n}) - \nu_*\right) + n \int_{\frac{\log(1/\delta)}{n}}^{1} \gamma^{-1}(F^{-1}(t) - \nu_*)dt \right], \quad (2.5)$$

and this sample complexity may be substantially smaller than the budget required by uniform allocation shown in Eq. (2.4) of Proposition 4. Essentially, the first term in Eq. (2.5) represents the budget allocated to the constant number of arms with limits $\nu_i \approx F^{-1}(\frac{\log(1/\delta)}{n})$ while the second term describes the number of times the sub-optimal arms are sampled before discarded. The next section uses a particular parameterization for F and γ to help better illustrate the difference between the sample complexity of uniform allocation (Equation 2.4) versus that of SUCCESSIVEHALVING (Equation 2.5).

2.5.3.2 A Parameterization of F and γ for Interpretability

To gain some intuition and relate the results back to the existing literature, we make explicit parametric assumptions on F and γ. We stress that all of our results hold for general F and γ as previously stated, and this parameterization is simply a tool to provide intuition. First assume that there exists a constant $\alpha > 0$ such that

$$\gamma(j) \simeq \left(\frac{1}{j}\right)^{1/\alpha}. \quad (2.6)$$

Note that a large value of α implies that the convergence of $\ell_{i,k} \to \nu_i$ is very slow.

We will consider two possible parameterizations of F. First, assume there exists positive constants β such that

$$F(x) \simeq \begin{cases} (x - \nu_*)^\beta & \text{if } x \geq \nu_* \\ 0 & \text{if } x < \nu_* \end{cases}. \quad (2.7)$$

Here, a large value of β implies that it is very rare to draw a limit close to the optimal value ν_*. The same model was studied in Carpentier and Valko [2015]. Fix some $\Delta > 0$. As discussed in the preceding section, if $n = \frac{\log(1/\delta)}{F(\nu_* + \Delta)} \simeq \Delta^{-\beta}\log(1/\delta)$ arms are drawn from F then with probability at least $1 - \delta$ we have $\min_{i=1,\ldots,n} \nu_i \leq \nu_* + \Delta$. Predictably, both uniform allocation and SUCCESSIVEHALVING output a $\nu_{\hat{\imath}}$ that satisfies $\nu_{\hat{\imath}} - \nu_* \lesssim \left(\frac{\log(1/\delta)}{n}\right)^{1/\beta}$ with probability at least $1 - \delta$ provided their measurement budgets are large enough. Thus, if $n \simeq \Delta^{-\beta}\log(1/\delta)$ and the measurement budgets of the uniform allocation (Equation 2.4) and SUCCESSIVEHALVING

[10] We say $f \simeq g$ if there exist constants c, c' such that $cg(x) \leq f(x) \leq c'g(x)$.

(Equation 2.5) satisfy

$$\begin{aligned}
\text{Uniform allocation} \qquad & B \simeq \Delta^{-(\alpha+\beta)} \log(1/\delta) \\
\textsc{SuccessiveHalving} \qquad & B \simeq \log_2(\Delta^{-\beta} \log(1/\delta)) \left[\Delta^{-\alpha} \log(1/\delta) + \frac{\Delta^{-\beta} - \Delta^{-\alpha}}{1 - \alpha/\beta} \log(1/\delta)\right] \\
& \simeq \log(\Delta^{-1} \log(1/\delta)) \log(\Delta^{-1}) \, \Delta^{-\max\{\beta,\alpha\}} \log(1/\delta)
\end{aligned}$$

then both also satisfy $\nu_{\hat{\imath}} - \nu_* \lesssim \Delta$ with probability at least $1 - \delta$.[11] SUCCESSIVEHALVING's budget scales like $\Delta^{-\max\{\alpha,\beta\}}$, which can be significantly smaller than the uniform allocation's budget of $\Delta^{-(\alpha+\beta)}$. However, because α and β are unknown in practice, neither method knows how to choose the optimal n or B to achieve this Δ accuracy. In Section 2.5.3.3, we show how HYPERBAND addresses this issue.

The second parameterization of F is the following discrete distribution:

$$F(x) = \frac{1}{K} \sum_{j=1}^{K} \mathbf{1}\{x \le \mu_j\} \qquad \text{with} \qquad \Delta_j := \mu_j - \mu_1 \tag{2.8}$$

for some set of unique scalars $\mu_1 < \mu_2 < \cdots < \mu_K$. Note that by letting $K \to \infty$ this discrete CDF can approximate any piecewise-continuous CDF to arbitrary accuracy. In particular, this model can have multiple means take the same value so that α mass is on μ_1 and $1 - \alpha$ mass is on $\mu_2 > \mu_1$, capturing the stochastic infinite-armed bandit model of Jamieson et al. [2016]. In this setting, both uniform allocation and SUCCESSIVEHALVING output a $\nu_{\hat{\imath}}$ that is within the top $\frac{\log(1/\delta)}{n}$ fraction of the K arms with probability at least $1 - \delta$ if their budgets are sufficiently large. Thus, let $q > 0$ be such that $n \simeq q^{-1} \log(1/\delta)$. Then, if the measurement budgets of the uniform allocation (Equation 2.4) and SUCCESSIVEHALVING (Equation 2.5) satisfy

$$\begin{aligned}
\text{Uniform allocation} \qquad & B \simeq \log(1/\delta) \begin{cases} K \max\limits_{j=2,\ldots,K} \Delta_j^{-\alpha} & \text{if } q = 1/K \\ q^{-1} \Delta_{\lceil qK \rceil}^{-\alpha} & \text{if } q > 1/K \end{cases} \\
\textsc{SuccessiveHalving} \qquad & B \simeq \log(q^{-1} \log(1/\delta)) \log(1/\delta) \begin{cases} \Delta_2^{-\alpha} + \sum\limits_{j=2}^{K} \Delta_j^{-\alpha} & \text{if } q = 1/K \\ \Delta_{\lceil qK \rceil}^{-\alpha} + \frac{1}{qK} \sum\limits_{j=\lceil qK \rceil}^{K} \Delta_j^{-\alpha} & \text{if } q > 1/K, \end{cases}
\end{aligned}$$

an arm that is in the best q-fraction of arms is returned, i.e. $\hat{\imath}/K \approx q$ and $\nu_{\hat{\imath}} - \nu_* \lesssim \Delta_{\lceil \max\{2, qK\} \rceil}$, with probability at least $1 - \delta$. This shows that the average resource per arm for uniform allocation is that required to distinguish the top q-fraction from the best, while that for SUCCESSIVEHALVING is a small multiple of the average resource required to distinguish an arm from the best; the difference between the max and the average can be very large in practice. We remark that the value of ϵ in Corollary 3 is carefully chosen to make the SUCCESSIVEHALVING budget and guarantee work out. Also note that one would never take $q < 1/K$ because $q = 1/K$ is sufficient to return the best arm.

[11] These quantities are intermediate results in the proofs of the theorems of Section 2.5.3.3.

2.5.3.3 HYPERBAND Guarantees

The HYPERBAND algorithm of Figure 2.8 addresses the tradeoff between the number of arms n versus the average number of times each one is pulled B/n by performing a two-dimensional version of the so-called "doubling trick." For each fixed B, we non-adaptively search a predetermined grid of values of n spaced geometrically apart so that the incurred loss of identifying the "best" setting takes a budget no more than $\log(B)$ times the budget necessary if the best setting of n were known ahead of time. Then, we successively double B so that the cumulative number of measurements needed to arrive at the necessary B is no more than $2B$. The idea is that even though we do not know the optimal setting for B, n to achieve some desired error rate, the hope is that by trying different values in a particular order, we will not waste too much effort.

Fix $\delta \in (0,1)$. For all (k,s) pairs defined in the HYPERBAND algorithm of Figure 2.8, let $\delta_{k,s} = \frac{\delta}{2k^3}$. For all (k,s) define

$$\mathcal{E}_{k,s} := \{B_{k,s} > 4\lceil \log_2(n_{k,s})\rceil \mathbf{H}(F,\gamma,n_{k,s},\delta_{k,s})\} = \{2^k > 4s\mathbf{H}(F,\gamma,2^s,2k^3/\delta)\}$$

Then by Corollary 3 we have

$$\mathbb{P}\left(\bigcup_{k=1}^{\infty}\bigcup_{s=1}^{k}\{\nu_{\hat{i}_{k,s}} - \nu_* > 3\big(F^{-1}(\tfrac{\log(4k^3/\delta)}{2^s}) - \nu_*\big)\} \cap \mathcal{E}_{k,s}\right) \le \sum_{k=1}^{\infty}\sum_{s=1}^{k}\delta_{k,s} = \sum_{k=1}^{\infty}\frac{\delta}{2k^2} \le \delta.$$

For sufficiently large k we will have $\bigcup_{s=1}^{k}\mathcal{E}_{k,s} \neq \emptyset$, so assume $B = 2^k$ is sufficiently large. Let $\hat{i}_B$ be the empirically best-performing arm output from SUCCESSIVEHALVING of round $k_B = \lfloor \log_2(B)\rfloor$ of HYPERBAND of Figure 2.8 and let $s_B \le k_B$ be the largest value such that $\mathcal{E}_{k_B,s_B}$ holds. Then

$$\nu_{\hat{i}_B} - \nu_* \le 3\big(F^{-1}(\frac{\log(4\lfloor\log_2(B)\rfloor^3/\delta)}{2^{s_B}}) - \nu_*\big) + \gamma(\lfloor\tfrac{2^{\lfloor\log_2(B)\rfloor-1}}{\lfloor\log_2(B)\rfloor}\rfloor).$$

Also note that on stage k at most $\sum_{i=1}^{k} iB_{i,1} \le k\sum_{i=1}^{k}B_{i,1} \le 2kB_{k,s} = 2\log_2(B_{k,s})B_{k,s}$ total samples have been taken. While this guarantee holds for general F,γ, the value of s_B, and consequently the resulting bound, is difficult to interpret. The following corollary considers the β,α parameterizations of F and γ, respectively, of Section 2.5.3.2 for better interpretation.

Theorem 5. *Assume that Assumptions 1 and 2 of Section 2.5.2 hold and that the sampled loss sequences obey the parametric assumptions of Equations 2.6 and 2.7. Fix $\delta \in (0,1)$. For any $T \in \mathbb{N}$, let $\hat{i}_T$ be the empirically best-performing arm output from* SUCCESSIVEHALVING *from the last round k of* HYPERBAND *of Figure 2.8 after exhausting a total budget of T from all rounds, then*

$$\nu_{\hat{i}_T} - \nu_* \le c\left(\frac{\overline{\log}(T)^3\overline{\log}(\log(T)/\delta)}{T}\right)^{1/\max\{\alpha,\beta\}}$$

for some constant $c = \exp(O(\max\{\alpha,\beta\}))$ where $\overline{\log}(x) = \log(x)\log\log(x)$.

By a straightforward modification of the proof, one can show that if uniform allocation is used in place of SUCCESSIVEHALVING in HYPERBAND, the uniform allocation version achieves $\nu_{\hat{i}_T} - \nu_* \le c\left(\frac{\log(T)\overline{\log}(\log(T)/\delta)}{T}\right)^{1/(\alpha+\beta)}$. We apply the above theorem to the stochastic infinite-armed bandit setting in the following corollary.

Corollary 6. *[Stochastic Infinite-armed Bandits] For any step k, s in the infinite horizon* HYPERBAND *algorithm with $n_{k,s}$ arms drawn, consider the setting where the jth pull of the ith arm results in a stochastic loss $Y_{i,j} \in [0,1]$ such that $\mathbb{E}[Y_{i,j}] = \nu_i$ and $\mathbb{P}(\nu_i - \nu_* \leq \epsilon) = c_1^{-1}\epsilon^\beta$. If $\ell_j(i) = \frac{1}{j}\sum_{s=1}^{j} Y_{i,s}$ then with probability at least $1 - \delta/2$ we have $\forall k \geq 1, 0 \leq s \leq k, 1 \leq i \leq n_{k,s}, 1 \leq j \leq B_k$,*

$$|\nu_i - \ell_{i,j}| \leq \sqrt{\frac{\log(B_k n_{k,s}/\delta_{k,s})}{2j}} \leq \sqrt{\log(\tfrac{16B_k}{\delta})}\left(\tfrac{2}{j}\right)^{1/2}.$$

Consequently, if after B total pulls we define $\widehat{\nu}_B$ as the mean of the empirically best arm output from the last fully completed round k, then with probability at least $1 - \delta$

$$\widehat{\nu}_B - \nu_* \leq \text{polylog}(B/\delta)\max\{B^{-1/2}, B^{-1/\beta}\}.$$

The result of this corollary matches the anytime result of Section 4.3 of Carpentier and Valko [2015] whose algorithm was built specifically for the case of stochastic arms and the β parameterization of F defined in Eq. (2.7). Notably, this result also matches the *lower bounds* shown in that work up to poly-logarithmic factors, revealing that HYPERBAND is nearly tight for this important special case. However, we note that this earlier work has a more careful analysis for the fixed budget setting.

Theorem 7. *Assume that Assumptions 1 and 2 of Section 2.5.2 hold and that the sampled loss sequences obey the parametric assumptions of Equations 2.6 and 2.8. For any $T \in \mathbb{N}$, let $\hat{\imath}_T$ be the empirically best-performing arm output from* SUCCESSIVEHALVING *from the last round k of* HYPERBAND *of Figure 2.8 after exhausting a total budget of T from all rounds. Fix $\delta \in (0,1)$ and $q \in (1/K, 1)$ and let $z_q = \log(q^{-1})(\Delta^{-\alpha}_{\lceil \max\{2,qK\}\rceil} + \frac{1}{qK}\sum_{i=\lceil \max\{2,qK\}\rceil}^{K} \Delta_i^{-\alpha})$. Once $T = \widetilde{\Omega}\left(z_q \log(z_q)\log(1/\delta)\right)$ total pulls have been made by* HYPERBAND *we have $\widehat{\nu}_T - \nu_* \leq \Delta_{\lceil \max\{2,qK\}\rceil}$ with probability at least $1 - \delta$ where $\widetilde{\Omega}(\cdot)$ hides $\log\log(\cdot)$ factors.*

Appealing to the stochastic setting of Corollary 6 so that $\alpha = 2$, we conclude that the sample complexity sufficient to identify an arm within the best q proportion with probabiltiy $1 - \delta$, up to log factors, scales like $\log(1/\delta)\log(q^{-1})(\Delta^{-\alpha}_{\lceil qK\rceil} + \frac{1}{qK}\sum_{i=\lceil qK\rceil}^{K}\Delta_i^{-\alpha})$. One may interpret this result as an extension of the distribution-dependent pure-exploration results of Bubeck et al. [2009]; but in our case, our bounds hold when the number of pulls is potentially much smaller than the number of arms K. When $q = 1/K$ this implies that the best arm is identified with about $\log(1/\delta)\log(K)\{\Delta_2^{-2} + \sum_{i=2}^{K}\Delta_i^{-2}\}$ which matches known upper bounds [Jamieson et al., 2014, Karnin et al., 2013] and lower bounds [Kaufmann et al., 2015] up to $\log$ factors. Thus, for the stochastic K-armed bandit problem HYPERBAND recovers many of the known sample complexity results up to $\log$ factors.

2.5.4 Finite Horizon Setting ($R < \infty$)

In this section we analyze the algorithm described in Section 2.3, i.e. finite horizon HYPERBAND. We present similar theoretical guarantees as in Section 2.5.3 for infinite horizon HYPERBAND, and fortunately much of the analysis will be recycled. We state the finite horizon version of the SUCCESSIVEHALVING and HYPERBAND algorithms in Figure 2.9.

SUCCESSIVEHALVING (Finite horizon)

input: Budget B, and n arms where $\ell_{i,k}$ denotes the kth loss from the ith arm, maximum size R, $\eta \geq 2$ ($\eta = 3$ by default).

Initialize: $S_0 = [n]$, $s = \min\{t \in \mathbb{N} : nR(t+1)\eta^{-t} \leq B, t \leq \log_\eta(\min\{R, n\})\}$.

For $k = 0, 1, \ldots, s$

Set $n_k = \lfloor n\eta^{-k} \rfloor$, $r_k = \lfloor R\eta^{k-s} \rfloor$

Pull each arm in S_k for r_k times.

Keep the best $\lfloor n\eta^{-(k+1)} \rfloor$ arms in terms of the r_kth observed loss as S_{k+1}.

Output : $\hat{\imath}, \ell_{\hat{\imath},R}$ where $\hat{\imath} = \arg\min_{i \in S_{s+1}} \ell_{i,R}$

HYPERBAND (Finite horizon)

Input: Budget B, maximum size R, $\eta \geq 2$ ($\eta = 3$ by default)

Initialize: $s_{\max} = \lfloor \log(R)/\log(\eta) \rfloor$

For $k = 1, 2, \ldots$

For $s = s_{\max}, s_{\max} - 1, \ldots, 0$

$B_{k,s} = 2^k$, $n_{k,s} = \lceil \frac{2^k \eta^s}{R(s+1)} \rceil$

$\hat{\imath}_s, \ell_{\hat{\imath}_s,R} \leftarrow$ SUCCESSIVEHALVING ($B_{k,s}$,$n_{k,s}$,R,η)

Figure 2.9: The finite horizon SUCCESSIVEHALVING and HYPERBAND algorithms are inspired by their infinite horizon counterparts of Figure 2.8 to handle practical constraints. HYPERBAND calls SUCCESSIVEHALVING as a subroutine.

The finite horizon setting differs in two major ways. First, in each bracket at least one arm will be pulled R times, but no arm will be pulled more than R times. Second, the number of brackets, each representing SUCCESSIVEHALVING with a different tradeoff between n and B, is fixed at $\log_\eta(R)+1$. Hence, since we are sampling sequences randomly i.i.d., increasing B over time would just multiply the number of arms in each bracket by a constant, affecting performance only by a small constant.

Theorem 8. *Fix n arms. Let $\nu_i = \ell_{i,R}$ and assume $\nu_1 \leq \cdots \leq \nu_n$. For any $\epsilon > 0$ let*

$$z_{SHA} = \eta(\log_\eta(R)+1)\left[n + \sum_{i=1}^{n} \min\left\{R, \gamma^{-1}\left(\max\left\{\tfrac{\epsilon}{4}, \tfrac{\nu_i-\nu_1}{2}\right\}\right)\right\}\right]$$

If the Successive Halving algorithm of Figure 2.9 is run with any budget $B \geq z_{SHA}$ then an arm $\hat{\imath}$ is returned that satisfies $\nu_{\hat{\imath}} - \nu_1 \leq \epsilon/2$.

Recall that $\gamma(R) = 0$ in this setting and by definition $\sup_{y\geq 0}\gamma^{-1}(y) \leq R$. Note that Lemma 2 still applies in this setting and just like above we obtain the following corollary.

Corollary 9. *Fix $\delta \in (0,1)$ and $\epsilon \geq 4(F^{-1}(\frac{\log(2/\delta)}{n}) - \nu_*)$. Let $\mathbf{H}(F,\gamma,n,\delta,\epsilon)$ be as defined in Lemma 2 and $B = \eta\log_\eta(R)(n + \max\{R, \mathbf{H}(F,\gamma,n,\delta,\epsilon)\})$. If the* SUCCESSIVEHALVING *algorithm of Figure 2.9 is run with the specified B and n arm configurations drawn randomly according to F then an arm $\hat{\imath} \in [n]$ is returned such that with probability at least $1-\delta$ we have $\nu_{\hat{\imath}} - \nu_* \leq \left(F^{-1}(\frac{\log(2/\delta)}{n}) - \nu_*\right) + \epsilon/2$. In particular, if $B = 4\lceil\log_2(n)\rceil\mathbf{H}(F,\gamma,n,\delta)$ and $\epsilon = 4(F^{-1}(\frac{\log(2/\delta)}{n}) - \nu_*)$ then $\nu_{\hat{\imath}} - \nu_* \leq 3\left(F^{-1}(\frac{\log(2/\delta)}{n}) - \nu_*\right)$ with probability at least $1-\delta$.*

As in Section 2.5.3.2 we can apply the α, β parameterization for interpretability, with the added constraint that $\sup_{y\geq 0}\gamma^{-1}(y) \leq R$ so that $\gamma(j) \simeq \mathbf{1}_{j<R}\left(\frac{1}{j}\right)^{1/\alpha}$. Note that the approximate sample complexity of SUCCESSIVEHALVING given in Eq. (2.5) is still valid for the finite horizon algorithm.

Fixing some $\Delta > 0$, $\delta \in (0,1)$, and applying the parameterization of Eq. (2.7) we recognize that if $n \simeq \Delta^{-\beta}\log(1/\delta)$ and the sufficient sampling budgets (treating η as an absolute constant) of the uniform allocation (Equation 2.4) and SUCCESSIVEHALVING (Eq. (2.5)) satisfy

$$\begin{aligned} \text{Uniform allocation} \quad & B \simeq R\Delta^{-\beta}\log(1/\delta) \\ \text{SUCCESSIVEHALVING} \quad & B \simeq \log(\Delta^{-1}\log(1/\delta))\log(1/\delta)\left[R + \Delta^{-\beta}\frac{1-(\alpha/\beta)R^{1-\beta/\alpha}}{1-\alpha/\beta}\right] \end{aligned}$$

then both also satisfy $\nu_{\hat{\imath}} - \nu_* \lesssim \Delta$ with probability at least $1-\delta$. Recalling that a larger α means slower convergence and that a larger β means a greater difficulty of sampling a good limit, note that when $\alpha/\beta < 1$ the budget of SUCCESSIVEHALVING behaves like $R + \Delta^{-\beta}\log(1/\delta)$ but as $\alpha/\beta \to \infty$ the budget asymptotes to $R\Delta^{-\beta}\log(1/\delta)$.

We can also apply the discrete-CDF parameterization of Eq. (2.8). For any $q \in (0,1)$, if $n \simeq q^{-1}\log(1/\delta)$ and the measurement budgets of the uniform allocation (Equation 2.4) and

SUCCESSIVEHALVING (Equation 2.5) satisfy

$$\text{Uniform allocation:} \qquad B \simeq \log(1/\delta) \begin{cases} K \min\left\{R, \max_{j=2,\dots,K} \Delta_j^{-\alpha}\right\} & \text{if } q = 1/K \\ q^{-1} \min\{R, \Delta_{\lceil qK \rceil}^{-\alpha}\} & \text{if } q > 1/K \end{cases}$$

SUCCESSIVEHALVING:

$$B \simeq \log(q^{-1}\log(1/\delta))\log(1/\delta) \begin{cases} \min\{R, \Delta_2^{-\alpha}\} + \sum_{j=2}^{K} \min\{R, \Delta_j^{-\alpha}\} & \text{if } q = 1/K \\ \min\{R, \Delta_{\lceil qK \rceil}^{-\alpha}\} + \frac{1}{qK} \sum_{j=\lceil qK \rceil}^{K} \min\{R, \Delta_j^{-\alpha}\} & \text{if } q > 1/K \end{cases}$$

then an arm that is in the best q-fraction of arms is returned, i.e. $\hat{\imath}/K \approx q$ and $\nu_{\hat{\imath}} - \nu_* \lesssim \Delta_{\lceil \max\{2, qK\} \rceil}$, with probability at least $1 - \delta$. Once again we observe a stark difference between uniform allocation and SUCCESSIVEHALVING, particularly when $\Delta_j^{-\alpha} \ll R$ for many values of $j \in \{1, \dots, n\}$.

Armed with Corollary 9, all of the discussion of Section 2.5.3.3 preceding Theorem 5 holds for the finite case ($R < \infty$) as well. Predictably analogous theorems also hold for the finite horizon setting, but their specific forms (with the polylog factors) provide no additional insights beyond the sample complexities sufficient for SUCCESSIVEHALVING to succeed, given immediately above.

It is important to note that in the finite horizon setting, for all sufficiently large B (e.g. $B > 3R$) and all distributions F, the budget B of SUCCESSIVEHALVING should scale *linearly* with $n \simeq \Delta^{-\beta} \log(1/\delta)$ as $\Delta \to 0$. Contrast this with the infinite horizon setting in which the ratio of B to n can become unbounded based on the values of α, β as $\Delta \to 0$. One consequence of this observation is that in the finite horizon setting it suffices to set B large enough to identify an Δ-good arm with just constant probability, say $1/10$, and then repeat SUCCESSIVEHALVING m times to boost this constant probability to probability $1 - (\frac{9}{10})^m$. While in this theoretical treatment of HYPERBAND we grow B over time, in practice we recommend fixing B as a multiple of R as we have done in Section 2.3. The fixed budget version of finite horizon HYPERBAND is more suitable for practical application due to the constant time, instead of exponential time, between configurations trained to completion in each outer loop.

2.6 Experimental Details

Additional details for experiments presented in 2.4 are provided below.

2.6.1 Experiments Using Alex Krizhevsky's CNN Architecture

For the experiments discussed in Section 2.4.1, the exact architecture used is the 18% model provided on cuda-convnet for CIFAR-10.[12]

[12]The model specification is available at http://code.google.com/p/cuda-convnet/.

Search Space: The search space used for the experiments is shown in Table 2.1. The learning rate reductions hyperparameter indicates how many times the learning rate was reduced by a factor of 10 over the maximum iteration window. For example, on CIFAR-10, which has a maximum iteration of 30,000, a learning rate reduction of 2 corresponds to reducing the learning every 10,000 iterations, for a total of 2 reductions over the 30,000 iteration window. All hyperparameters, with the exception of the learning rate decay reduction, overlap with those in Snoek et al. [2012]. Two hyperparameters in Snoek et al. [2012] were excluded from our experiments: (1) the width of the response normalization layer was excluded due to limitations of the Caffe framework and (2) the number of epochs was excluded because it is incompatible with dynamic resource allocation.

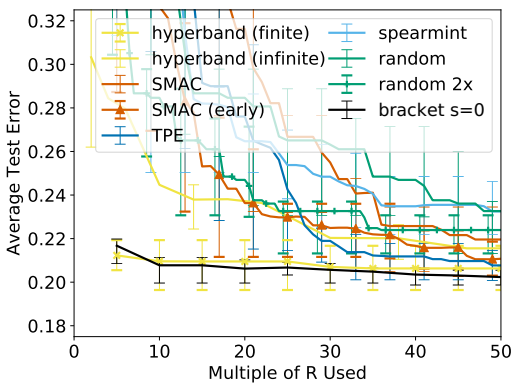
Hyperparameter	Scale	Min	Max
Learning Parameters			
Initial Learning Rate	log	$5 * 10^{-5}$	5
Conv1 L_2 Penalty	log	$5 * 10^{-5}$	5
Conv2 L_2 Penalty	log	$5 * 10^{-5}$	5
Conv3 L_2 Penalty	log	$5 * 10^{-5}$	5
FC4 L_2 Penalty	log	$5 * 10^{-3}$	500
Learning Rate Reductions	integer	0	3
Local Response Normalization			
Scale	log	$5 * 10^{-6}$	5
Power	linear	0.01	3

Table 2.1: Hyperparameters and associated ranges for the three-layer convolutional network.

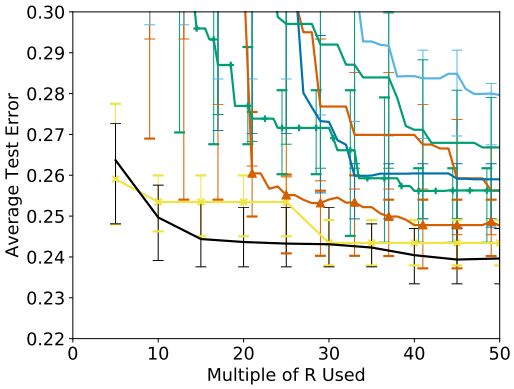
Data Splits: For CIFAR-10, the training (40,000 instances) and validation (10,000 instances) sets were sampled from data batches 1-5 with balanced classes. The original test set (10,000 instances) was used for testing. For MRBI, the training (10,000 instances) and validation (2,000 instances) sets were sampled from the original training set with balanced classes. The original test set (50,000 instances) was used for testing. Lastly, for SVHN, the train, validation, and test splits were created using the same procedure as that in Sermanet et al. [2012].

Comparison with Early-Stopping: Domhan et al. [2015] proposed an early-stopping method for neural networks and combined it with SMAC to speed up hyperparameter optimization. Their method stops training a configuration if the probability of the configuration beating the current best is below a specified threshold. This probability is estimated by extrapolating learning curves fit to the intermediate validation error losses of a configuration. If a configuration is terminated early, the predicted terminal value from the estimated learning curves is used as the validation error passed to the hyperparameter optimization algorithm. Hence, if the learning curve fit is poor, it could impact the performance of the configuration selection algorithm. While this approach is heuristic in nature, it could work well in practice so we compare HYPERBAND to SMAC with early termination (labeled SMAC (early) in Figure 2.10). We used the conservative termination criterion with default parameters and recorded the validation loss every 400 iterations and evaluated the termination criterion 3 times within the training period (every 8k iterations for CIFAR-10 and MRBI and every 16k iterations for SVHN).[13] Comparing the performance by the number of total

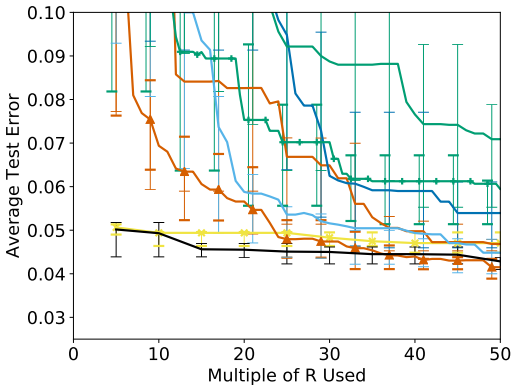
[13]We used the code provided at https://github.com/automl/pylearningcurvepredictor.



(a) CIFAR-10



(b) MRBI



(c) SVHN

Figure 2.10: Average test error across 10 trials is shown in all plots. Error bars indicate the top and bottom quartiles of the test error corresponding to the model with the best validation error

iterations as mulitple of R is conservative because it does not account for the time spent fitting the learning curve in order to check the termination criterion.

2.6.2 117 Data Sets Experiment

For the experiments discussed in Section 2.4.2.1, we followed Feurer et al. [2015a] and imposed a 3GB memory limit, a 6-minute timeout for each hyperparameter configuration and a one-hour time window to evaluate each searcher on each data set. Moreover, we evaluated the performance of each searcher by aggregating results across all data sets and reporting the average rank of each method. Specifically, the hour training window is divided up into 30 second intervals and, at each time point, the model with the best validation error at that time is used in the calculation of the average error across all trials for each (data set-searcher) pair. Then, the performance of each searcher is ranked by data set and averaged across all data sets. All experiments were performed on Google Cloud Compute n1-standard-1 instances in us-central1-f region with 1 CPU

and 3.75GB of memory.

Data Splits: Feurer et al. [2015a] split each data set into 2/3 training and 1/3 test set, whereas we introduce a validation set to avoid overfitting to the test data. We also used 2/3 of the data for training, but split the rest of the data into two equally sized validation and test sets. We reported results on both the validation and test data. Moreover, we performed 20 trials of each (data set-searcher) pair, and as in Feurer et al. [2015a] we kept the same data splits across trials, while using a different random seed for each searcher in each trial.

Shortcomings of the Experimental Setup: The benchmark contains a large variety of training set sizes and feature dimensions[14] resulting in random search being able to test 600 configurations on some data sets but just dozens on others. HYPERBAND was designed under the implicit assumption that computation scaled at least linearly with the data set size. For very small data sets that are trained in seconds, the initialization overheads dominate the computation and subsampling provides no computational benefit. In addition, many of the classifiers and preprocessing methods under consideration return memory errors as they require storage quadratic in the number of features (e.g., covariance matrix) or the number of observations (e.g., kernel methods). These errors usually happen immediately (thus wasting little time); however, they often occur on the full data set and not on subsampled data sets. A searcher like HYPERBAND that uses a subsampled data set could spend significant time training on a subsample only to error out when attempting to train it on the full data set.

2.6.3 Kernel Classification Experiments

Table 2.2 shows the hyperparameters and associated ranges considered in the kernel least squares classification experiment discussed in Section 2.4.2.2.

Hyperparameter	Type	Values
preprocessor	Categorical	min/max, standardize, normalize
kernel	Categorical	rbf, polynomial, sigmoid
C	Continuous	log $[10^{-3}, 10^{5}]$
gamma	Continuous	log $[10^{-5}, 10]$
degree	if kernel=poly	integer [2, 5]
coef0	if kernel=poly, sigmoid	uniform [-1.0, 1.0]

Table 2.2: Hyperparameter space for kernel regularized least squares classification problem discussed in Section 2.4.2.2.

Table 2.3 shows the hyperparameters and associated ranges considered in the random features kernel approximation classification experiment discussed in Section 2.4.3. The regularization term λ is divided by the number of features so that the tradeoff between the squared error and the L_2 penalty would remain constant as the resource increased. Additionally, the average test error with the top and bottom quartiles across 10 trials are show in Figure 2.12.

[14]Training set size ranges from 670 to 73,962 observations, and number of features ranges from 1 to 10,935.

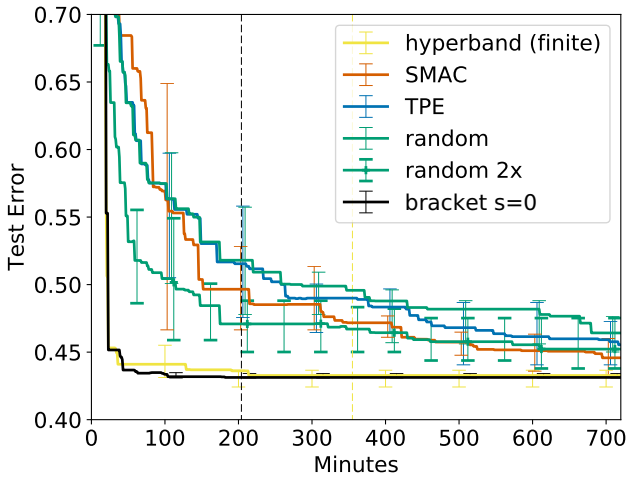


Figure 2.11: Average test error of the best kernel regularized least square classification model found by each searcher on CIFAR-10. The color coded dashed lines indicate when the last trial of a given searcher finished. Error bars correspond to the top and bottom quartiles of the test error across 10 trials.

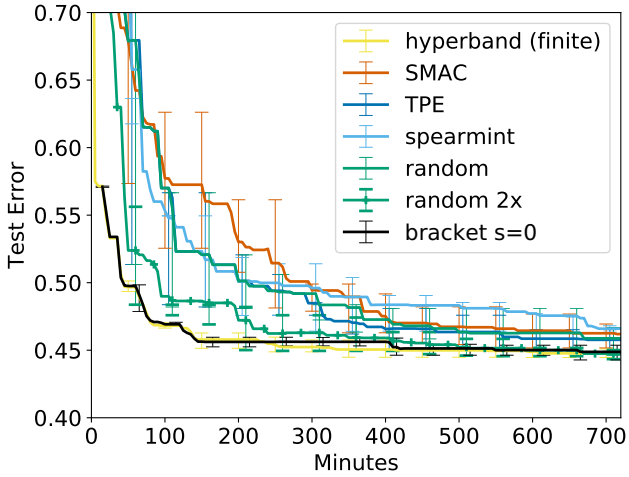


Figure 2.12: Average test error of the best random features model found by each searcher on CIFAR-10. The test error for HYPERBAND and bracket $s = 0$ are calculated in every evaluation instead of at the end of a bracket. Error bars correspond to the top and bottom quartiles of the test error across 10 trials.

Hyperparameter	Type	Values
preprocessor	Categorical	none, min/max, standardize, normalize
λ	Continuous	log $[10^{-3}, 10^{5}]$
gamma	Continuous	log $[10^{-5}, 10]$

Table 2.3: Hyperparameter space for random feature kernel approximation classification problem discussed in Section 2.4.3.

The cost term C is divided by the number of samples so that the tradeoff between the squared error and the L_2 penalty would remain constant as the resource increased (squared error is summed across observations and not averaged). The regularization term λ is equal to the inverse of the scaled cost term C. Additionally, the average test error with the top and bottom quartiles across 10 trials are show in Figure 2.11.

2.7 Proofs of Theoretical Results

In this section, we provide proofs for the theorems presented in Section 2.5.

2.7.1 Proof of Theorem 1

Proof. First, we verify that the algorithm never takes a total number of samples that exceeds the budget B:

$$\sum_{k=0}^{\lceil \log_2(n) \rceil - 1} |S_k| \left\lfloor \frac{B}{|S_k| \lceil \log(n) \rceil} \right\rfloor \leq \sum_{k=0}^{\lceil \log_2(n) \rceil - 1} \frac{B}{\lceil \log(n) \rceil} \leq B \, .$$

For notational ease, let $\ell_{i,j} := \ell_j(X_i)$. Again, for each $i \in [n] := \{1, \ldots, n\}$ we assume the limit $\lim_{k \to \infty} \ell_{i,k}$ exists and is equal to ν_i. As a reminder, $\gamma : \mathbb{N} \to \mathbb{R}$ is defined as the pointwise smallest, monotonically decreasing function satisfying

$$\max_i |\ell_{i,j} - \nu_i| \leq \gamma(j) \, , \quad \forall j \in \mathbb{N}. \tag{2.9}$$

Note γ is guaranteed to exist by the existence of ν_i and bounds the deviation from the limit value as the sequence of iterates j increases.

Without loss of generality, order the terminal losses so that $\nu_1 \leq \nu_2 \leq \cdots \leq \nu_n$. Assume that

$B \geq z_{SH}$. Then we have for each round k

$$\begin{aligned}
r_k &\geq \frac{B}{|S_k|\lceil \log_2(n) \rceil} - 1 \\
&\geq \frac{2}{|S_k|} \max_{i=2,\ldots,n} i\left(1 + \gamma^{-1}\big(\max\big\{\frac{\epsilon}{4}, \frac{\nu_i - \nu_1}{2}\big\}\big)\right) - 1 \\
&\geq \frac{2}{|S_k|}\left(\lfloor |S_k|/2 \rfloor + 1\right)\left(1 + \gamma^{-1}\big(\max\big\{\frac{\epsilon}{4}, \frac{\nu_{\lfloor |S_k|/2 \rfloor + 1} - \nu_1}{2}\big\}\big)\right) - 1 \\
&\geq \left(1 + \gamma^{-1}\big(\max\big\{\frac{\epsilon}{4}, \frac{\nu_{\lfloor |S_k|/2 \rfloor + 1} - \nu_1}{2}\big\}\big)\right) - 1 \\
&= \gamma^{-1}\big(\max\big\{\frac{\epsilon}{4}, \frac{\nu_{\lfloor |S_k|/2 \rfloor + 1} - \nu_1}{2}\big\}\big),
\end{aligned}$$

where the fourth line follows from $\lfloor |S_k|/2 \rfloor \geq |S_k|/2 - 1$.

First we show that $\ell_{i,t} - \ell_{1,t} > 0$ for all $t \geq \tau_i := \gamma^{-1}\big(\frac{\nu_i - \nu_1}{2}\big)$. Given the definition of γ, we have for all $i \in [n]$ that $|\ell_{i,t} - \nu_i| \leq \gamma(t) \leq \frac{\nu_i - \nu_1}{2}$ where the last inequality holds for $t \geq \tau_i$. Thus, for $t \geq \tau_i$ we have

$$\begin{aligned}
\ell_{i,t} - \ell_{1,t} &= \ell_{i,t} - \nu_i + \nu_i - \nu_1 + \nu_1 - \ell_{1,t} \\
&= \ell_{i,t} - \nu_i - (\ell_{1,t} - \nu_1) + \nu_i - \nu_1 \\
&\geq -2\gamma(t) + \nu_i - \nu_1 \\
&\geq -2\frac{\nu_i - \nu_1}{2} + \nu_i - \nu_1 \\
&= 0.
\end{aligned}$$

Under this scenario, we will eliminate arm i before arm 1 since on each round the arms are sorted by their empirical losses and the top half are discarded. Note that by the assumption the ν_i limits are non-decreasing in i so that the τ_i values are non-increasing in i.

Fix a round k and assume $1 \in S_k$ (note, $1 \in S_0$). The above calculation shows that

$$t \geq \tau_i \implies \ell_{i,t} \geq \ell_{1,t}. \tag{2.10}$$

Consequently,

$$\begin{aligned}
\{1 \in S_k,\ \ 1 \notin S_{k+1}\} &\iff \left\{\sum_{i \in S_k} \mathbf{1}\{\ell_{i,r_k} < \ell_{1,r_k}\} \geq \lfloor |S_k|/2 \rfloor\right\} \\
&\implies \left\{\sum_{i \in S_k} \mathbf{1}\{r_k < \tau_i\} \geq \lfloor |S_k|/2 \rfloor\right\} \\
&\implies \left\{\sum_{i=2}^{\lfloor |S_k|/2 \rfloor + 1} \mathbf{1}\{r_k < \tau_i\} \geq \lfloor |S_k|/2 \rfloor\right\} \\
&\iff \left\{r_k < \tau_{\lfloor |S_k|/2 \rfloor + 1}\right\}.
\end{aligned}$$

where the first line follows by the definition of the algorithm, the second by Equation 2.10, and the third by τ_i being non-increasing (for all $i < j$ we have $\tau_i \geq \tau_j$ and consequently, $\mathbf{1}\{r_k < \tau_i\} \geq \mathbf{1}\{r_k < \tau_j\}$ so the *first* indicators of the sum not including 1 would be on before any other i's in $S_k \subset [n]$ sprinkled throughout $[n]$). This implies

$$\{1 \in S_k, \;\; r_k \geq \tau_{\lfloor |S_k|/2 \rfloor + 1}\} \implies \{1 \in S_{k+1}\}. \tag{2.11}$$

Recalling that $r_k \geq \gamma^{-1}\big(\max\big\{\frac{\epsilon}{4}, \frac{\nu_{\lfloor |S_k|/2 \rfloor + 1} - \nu_1}{2}\big\}\big)$ and $\tau_{\lfloor |S_k|/2 \rfloor + 1} = \gamma^{-1}\big(\frac{\nu_{\lfloor |S_k|/2 \rfloor + 1} - \nu_1}{2}\big)$, we examine the following three exhaustive cases:

- **Case 1:** $\frac{\nu_{\lfloor |S_k|/2 \rfloor + 1} - \nu_1}{2} \geq \frac{\epsilon}{4}$ and $1 \in S_k$
  In this case, $r_k \geq \gamma^{-1}\big(\frac{\nu_{\lfloor |S_k|/2 \rfloor + 1} - \nu_1}{2}\big) = \tau_{\lfloor |S_k|/2 \rfloor + 1}$. By Equation 2.11 we have that $1 \in S_{k+1}$ since $1 \in S_k$.
- **Case 2:** $\frac{\nu_{\lfloor |S_k|/2 \rfloor + 1} - \nu_1}{2} < \frac{\epsilon}{4}$ and $1 \in S_k$
  In this case $r_k \geq \gamma^{-1}\big(\frac{\epsilon}{4}\big)$ but $\gamma^{-1}\big(\frac{\epsilon}{4}\big) < \tau_{\lfloor |S_k|/2 \rfloor + 1}$. Equation 2.11 suggests that it may be possible for $1 \in S_k$ but $1 \notin S_{k+1}$. On the good event that $1 \in S_{k+1}$, the algorithm continues and on the next round either case 1 or case 2 could be true. So assume $1 \notin S_{k+1}$. Here we show that $\{1 \in S_k, \;\; 1 \notin S_{k+1}\} \implies \max_{i \in S_{k+1}} \nu_i \leq \nu_1 + \epsilon/2$. Because $1 \in S_0$, this guarantees that SUCCESSIVEHALVING either exits with arm $\widehat{i} = 1$ or some arm $\widehat{i}$ satisfying $\nu_{\widehat{i}} \leq \nu_1 + \epsilon/2$.
  Let $p = \min\{i \in [n] : \frac{\nu_i - \nu_1}{2} \geq \frac{\epsilon}{4}\}$. Note that $p > \lfloor |S_k|/2 \rfloor + 1$ by the criterion of the case and
  $$r_k \geq \gamma^{-1}\left(\frac{\epsilon}{4}\right) \geq \gamma^{-1}\left(\frac{\nu_i - \nu_1}{2}\right) = \tau_i, \quad \forall i \geq p.$$
  Thus, by Equation 2.10 ($t \geq \tau_i \implies \ell_{i,t} \geq \ell_{1,t}$) we have that arms $i \geq p$ would always have $\ell_{i,r_k} \geq \ell_{1,r_k}$ and be eliminated before or at the same time as arm 1, presuming $1 \in S_k$. In conclusion, if arm 1 is eliminated so that $1 \in S_k$ but $1 \notin S_{k+1}$ then $\max_{i \in S_{k+1}} \nu_i \leq \max_{i<p} \nu_i < \nu_1 + \epsilon/2$ by the definition of p.
- **Case 3:** $1 \notin S_k$
  Since $1 \in S_0$, there exists some $r < k$ such that $1 \in S_r$ and $1 \notin S_{r+1}$. For this r, only case 2 is possible since case 1 would proliferate $1 \in S_{r+1}$. However, under case 2, if $1 \notin S_{r+1}$ then $\max_{i \in S_{r+1}} \nu_i \leq \nu_1 + \epsilon/2$.

Because $1 \in S_0$, we either have that 1 remains in S_k (possibly alternating between cases 1 and 2) for all k until the algorithm exits with the best arm 1, or there exists some k such that case 3 is true and the algorithm exits with an arm $\widehat{i}$ such that $\nu_{\widehat{i}} \leq \nu_1 + \epsilon/2$. The proof is complete by noting that

$$|\ell_{\widehat{i}, \lfloor \frac{B/2}{\lceil \log_2(n) \rceil} \rfloor} - \nu_1| \leq |\ell_{\widehat{i}, \lfloor \frac{B/2}{\lceil \log_2(n) \rceil} \rfloor} - \nu_{\widehat{i}}| + |\nu_{\widehat{i}} - \nu_1| \leq \epsilon/4 + \epsilon/2 \leq \epsilon$$

by the triangle inequality and because $B \geq 2\lceil \log_2(n) \rceil \gamma^{-1}(\epsilon/4)$ by assumption.

The second, looser, but perhaps more interpretable form of z_{SH} follows from the fact that $\gamma^{-1}(x)$ is non-increasing in x so that

$$\max_{i=2,\ldots,n} i\, \gamma^{-1}\left(\max\left\{\tfrac{\epsilon}{4}, \tfrac{\nu_i - \nu_1}{2}\right\}\right) \leq \sum_{i=1,\ldots,n} \gamma^{-1}\left(\max\left\{\tfrac{\epsilon}{4}, \tfrac{\nu_i - \nu_1}{2}\right\}\right).$$

□

2.7.2 Proof of Lemma 2

Proof. Let $p_n = \frac{\log(2/\delta)}{n}$, $M = \gamma^{-1}\left(\frac{\epsilon}{16}\right)$, and $\mu = \mathbb{E}[\min\{M, \gamma^{-1}\left(\frac{\nu_i - \nu_*}{4}\right)\}]$. Define the events

$$\xi_1 = \{\nu_1 \le F^{-1}(p_n)\}$$
$$\xi_2 = \left\{\sum_{i=1}^{n} \min\{M, \gamma^{-1}\left(\tfrac{\nu_i-\nu_*}{4}\right)\} \le n\mu + \sqrt{2n\mu M \log(2/\delta)} + \tfrac{2}{3}M\log(2/\delta)\right\}$$

Note that $\mathbb{P}(\xi_1^c) = \mathbb{P}(\min_{i=1,\dots,n}\nu_i > F^{-1}(p_n)) = (1-p_n)^n \le \exp(-np_n) \le \frac{\delta}{2}$. Moreover, $\mathbb{P}(\xi_2^c) \le \frac{\delta}{2}$ by Bernstein's inequality since

$$\mathbb{E}\left[\min\{M, \gamma^{-1}\left(\tfrac{\nu_i-\nu_*}{4}\right)\}^2\right] \le \mathbb{E}\left[M\min\{M, \gamma^{-1}\left(\tfrac{\nu_i-\nu_*}{4}\right)\}\right] = M\mu.$$

Thus, $\mathbb{P}(\xi_1 \cap \xi_2) \ge 1-\delta$ so in what follows assume these events hold.

First we show that if $\nu_* \le \nu_1 \le F^{-1}(p_n)$, which we will refer to as equation $(*)$, then $\max\left\{\frac{\epsilon}{4}, \frac{\nu_i-\nu_1}{2}\right\} \ge \max\left\{\frac{\epsilon}{4}, \frac{\nu_i-\nu_*}{4}\right\}$.

Case 1: $\frac{\epsilon}{4} \le \frac{\nu_i-\nu_1}{2}$ and $\epsilon \ge 4(F^{-1}(p_n)-\nu_*)$.

$$\frac{\nu_i-\nu_1}{2} \overset{(*)}{\ge} \frac{\nu_i-\nu_*+\nu_*-F^{-1}(p_n)}{2} = \frac{\nu_i-\nu_*}{4} + \frac{\nu_i-\nu_*}{4} - \frac{F^{-1}(p_n)-\nu_*}{2} \overset{(*)}{\ge} \frac{\nu_i-\nu_*}{4} + \frac{\nu_i-\nu_1}{4} - \frac{F^{-1}(p_n)-\nu_*}{2} \overset{\text{Case 1}}{\ge} \frac{\nu_i-\nu_*}{4}.$$

Case 2: $\frac{\epsilon}{4} > \frac{\nu_i-\nu_1}{2}$ and $\epsilon \ge 4(F^{-1}(p_n)-\nu_*)$.

$$\frac{\nu_i-\nu_*}{4} = \frac{\nu_i-\nu_1}{4} + \frac{\nu_1-\nu_*}{4} \overset{\text{Case 2}}{<} \frac{\epsilon}{8} + \frac{\nu_1-\nu_*}{4} \overset{(*)}{\le} \frac{\epsilon}{8} + \frac{F^{-1}(p_n)-\nu_*}{4} \overset{\text{Case 2}}{<} \frac{\epsilon}{4}$$

which shows the desired result.

Consequently, for any $\epsilon \ge 4(F^{-1}(p_n)-\nu_*)$ we have

$$\begin{aligned}
\sum_{i=1}^{n}\gamma^{-1}\left(\max\left\{\tfrac{\epsilon}{4}, \tfrac{\nu_i-\nu_1}{2}\right\}\right) &\le \sum_{i=1}^{n}\gamma^{-1}\left(\max\left\{\tfrac{\epsilon}{4}, \tfrac{\nu_i-\nu_*}{4}\right\}\right) \\
&\le \sum_{i=1}^{n}\gamma^{-1}\left(\max\left\{\tfrac{\epsilon}{16}, \tfrac{\nu_i-\nu_*}{4}\right\}\right) \\
&= \sum_{i=1}^{n}\min\{M, \gamma^{-1}\left(\tfrac{\nu_i-\nu_*}{4}\right)\} \\
&\le n\mu + \sqrt{2n\mu M\log(1/\delta)} + \tfrac{2}{3}M\log(1/\delta) \\
&\le \left(\sqrt{n\mu} + \sqrt{\tfrac{2}{3}M\log(2/\delta)}\right)^2 \le 2n\mu + \tfrac{4}{3}M\log(2/\delta).
\end{aligned}$$

A direct computation yields

$$\begin{aligned}
\mu &= \mathbb{E}[\min\{M, \gamma^{-1}\left(\tfrac{\nu_i-\nu_*}{4}\right)\}] \\
&= \mathbb{E}[\gamma^{-1}\left(\max\left\{\tfrac{\epsilon}{16}, \tfrac{\nu_i-\nu_*}{4}\right\}\right)] \\
&= \gamma^{-1}\left(\tfrac{\epsilon}{16}\right)F(\nu_*+\epsilon/4) + \int_{\nu_*+\epsilon/4}^{\infty}\gamma^{-1}(\tfrac{t-\nu_*}{4})dF(t)
\end{aligned}$$

so that

$$\sum_{i=1}^{n}\gamma^{-1}\left(\max\left\{\tfrac{\epsilon}{4},\tfrac{\nu_i-\nu_1}{2}\right\}\right) \leq 2n\mu + \tfrac{4}{3}M\log(2/\delta)$$
$$= 2n\int_{\nu_*+\epsilon/4}^{\infty}\gamma^{-1}(\tfrac{t-\nu_*}{4})dF(t) + \left(\tfrac{4}{3}\log(2/\delta) + 2nF(\nu_*+\epsilon/4)\right)\gamma^{-1}\left(\tfrac{\epsilon}{16}\right)$$

which completes the proof. □

2.7.3 Proof of Proposition 4

We break the proposition up into upper and lower bounds and prove them seperately.

2.7.4 Uniform Allocation

Proposition 10. *Suppose we draw n random configurations from F, train each with a budget of j,[15] and let $\hat{\imath} = \arg\min_{i=1,\dots,n}\ell_j(X_i)$. Let $\nu_i = \ell_*(X_i)$ and without loss of generality assume $\nu_1 \leq \ldots \leq \nu_n$. If*

$$B \geq n\gamma^{-1}\left(\tfrac{1}{2}(F^{-1}(\tfrac{\log(1/\delta)}{n}) - \nu_*)\right) \tag{2.12}$$

then with probability at least $1-\delta$ we have $\nu_{\hat{\imath}} - \nu_* \leq 2\left(F^{-1}\left(\frac{\log(1/\delta)}{n}\right) - \nu_*\right)$.

Proof. Note that if we draw n random configurations from F and $i_* = \arg\min_{i=1,\dots,n}\ell_*(X_i)$ then

$$\mathbb{P}\left(\ell_*(X_{i_*}) - \nu_* \leq \epsilon\right) = \mathbb{P}\Big(\bigcup_{i=1}^{n}\{\ell_*(X_i) - \nu_* \leq \epsilon\}\Big)$$
$$= 1 - (1 - F(\nu_*+\epsilon))^n \geq 1 - e^{-nF(\nu_*+\epsilon)},$$

which is equivalent to saying that with probability at least $1-\delta$, $\ell_*(X_{i_*}) - \nu_* \leq F^{-1}(\log(1/\delta)/n) - \nu_*$. Furthermore, if each configuration is trained for j iterations then with probability at least $1-\delta$

$$\ell_*(X_{\hat{\imath}}) - \nu_* \leq \ell_j(X_{\hat{\imath}}) - \nu_* + \gamma(j) \leq \ell_j(X_{i_*}) - \nu_* + \gamma(j)$$
$$\leq \ell_*(X_{i_*}) - \nu_* + 2\gamma(j) \leq F^{-1}\left(\tfrac{\log(1/\delta)}{n}\right) - \nu_* + 2\gamma(j).$$

If our measurement budget B is constrained so that $B = nj$ then solving for j in terms of B and n yields the result. □

The following proposition demonstrates that the upper bound on the error of the uniform allocation strategy in Proposition 4 is in fact tight. That is, for any distribution F and function γ there exists a loss sequence that requires the budget described in Eq. (2.4) in order to avoid a loss of more than ϵ with high probability.

[15] Here j can be bounded (finite horizon) or unbounded (infinite horizon).

Proposition 11. *Fix any $\delta \in (0,1)$ and $n \in \mathbb{N}$. For any $c \in (0,1]$, let $\mathcal{F}_c$ denote the space of continuous cumulative distribution functions F satisfying*[16] $\inf_{x\in[\nu_*,1-\nu_*]} \inf_{\Delta\in[0,1-x]} \frac{F(x+\Delta)-F(x+\Delta/2)}{F(x+\Delta)-F(x)} \geq c$*. And let Γ denote the space of monotonically decreasing functions over $\mathbb{N}$. For any $F \in \mathcal{F}_c$ and $\gamma \in \Gamma$ there exists a probability distribution μ over $\mathcal{X}$ and a sequence of functions $\ell_j : \mathcal{X} \to \mathbb{R}$ $\forall j \in \mathbb{N}$ with $\ell_* := \lim_{j\to\infty} \ell_j$, $\nu_* = \inf_{x\in\mathcal{X}} \ell_*(x)$ such that $\sup_{x\in\mathcal{X}} |\ell_j(x) - \ell_*(x)| \leq \gamma(j)$ and $\mathbb{P}_\mu(\ell_*(X) - \nu_* \leq \epsilon) = F(\epsilon)$. Moreover, if n configurations $X_1, \ldots, X_n$ are drawn from μ and $\hat{\imath} = \arg\min_{i\in 1,\ldots,n} \ell_{B/n}(X_i)$ then with probability at least δ*

$$\ell_*(X_{\hat{\imath}}) - \nu_* \geq 2(F^{-1}(\tfrac{\log(c/\delta)}{n+\log(c/\delta)}) - \nu_*)$$

whenever $B \leq n\gamma^{-1}\left(2(F^{-1}(\frac{\log(c/\delta)}{n+\log(c/\delta)}) - \nu_*)\right)$.

Proof. Let $\mathcal{X} = [0,1]$, $\ell_*(x) = F^{-1}(x)$, and μ be the uniform distribution over $[0,1]$. Define $\widehat{\nu} = F^{-1}(\frac{\log(c/\delta)}{n+\log(c/\delta)})$ and set

$$\ell_j(x) = \begin{cases} \widehat{\nu} + \frac{1}{2}\gamma(j) + (\widehat{\nu} + \frac{1}{2}\gamma(j) - \ell_*(x)) & \text{if } |\widehat{\nu} + \frac{1}{2}\gamma(j) - \ell_*(x)| \leq \frac{1}{2}\gamma(j) \\ \ell_*(x) & \text{otherwise.} \end{cases}$$

Essentially, if $\ell_*(x)$ is within $\frac{1}{2}\gamma(j)$ of $\widehat{\nu} + \frac{1}{2}\gamma(j)$ then we set $\ell_j(x)$ equal to $\ell_*(x)$ reflected across the value $2\widehat{\nu} + \gamma(j)$. Clearly, $|\ell_j(x) - \ell_*(x)| \leq \gamma(j)$ for all $x \in \mathcal{X}$.

Since each $\ell_*(X_i)$ is distributed according to F, we have

$$\mathbb{P}\Big(\bigcap_{i=1}^{n}\{\ell_*(X_i) - \nu_* \geq \epsilon\}\Big) = (1 - F(\nu_* + \epsilon))^n \geq e^{-nF(\nu_*+\epsilon)/(1-F(\nu_*+\epsilon))}.$$

Setting the right-hand-side greater than or equal to δ/c and solving for ϵ, we find $\nu_* + \epsilon \geq F^{-1}(\frac{\log(c/\delta)}{n+\log(c/\delta)}) = \widehat{\nu}$.

Define $I_0 = [\nu_*, \widehat{\nu})$, $I_1 = [\widehat{\nu}, \widehat{\nu}+\frac{1}{2}\gamma(B/n))$ and $I_2 = [\widehat{\nu}+\frac{1}{2}\gamma(B/n), \widehat{\nu}+\gamma(B/n)]$. Furthermore, for $j \in \{0,1,2\}$ define $N_j = \sum_{i=1}^{n} \mathbf{1}_{\ell_*(X_i)\in I_j}$. Given $N_0 = 0$ (which occurs with probability at least δ/c), if $N_1 = 0$ then $\ell_*(X_{\hat{\imath}}) - \nu_* \geq F^{-1}(\frac{\log(c/\delta)}{n+\log(c/\delta)}) + \frac{1}{2}\gamma(B/n)$ and the claim is true.

Below we will show that if $N_2 > 0$ whenever $N_1 > 0$ then the claim is also true. We now show that this happens with at least probability c whenever $N_1 + N_2 = m$ for any $m > 0$. Observe that

$$\begin{aligned}\mathbb{P}(N_2 > 0 | N_1 + N_2 = m) &= 1 - \mathbb{P}(N_2 = 0 | N_1 + N_2 = m) \\ &= 1 - (1 - \mathbb{P}(\nu_i \in I_2 | \nu_i \in I_1 \cup I_2))^m \geq 1 - (1-c)^m \geq c\end{aligned}$$

since

$$\mathbb{P}(\nu_i \in I_2 | \nu_i \in I_1 \cup I_2) = \frac{\mathbb{P}(\nu_i \in I_2)}{\mathbb{P}(\nu_i \in I_1 \cup I_2)} = \frac{\mathbb{P}(\nu_i \in [\widehat{\nu}+\frac{1}{2}\gamma, \widehat{\nu}+\gamma])}{\mathbb{P}(\nu_i \in [\widehat{\nu}, \widehat{\nu}+\gamma])} = \frac{F(\widehat{\nu}+\gamma) - F(\widehat{\nu}+\frac{1}{2}\gamma)}{F(\widehat{\nu}+\gamma) - F(\widehat{\nu})} \geq c\,.$$

Thus, the probability of the event that $N_0 = 0$ and $N_2 > 0$ whenever $N_1 > 0$ occurs with probability at least $\delta/c \cdot c = \delta$, so assume this is the case in what follows.

Since $N_0 = 0$, for all $j \in \mathbb{N}$, each X_i must fall into one of three cases:

[16] Note that this condition is met whenever F is convex. Moreover, if $F(\nu_* + \epsilon) = c_1^{-1}\epsilon^\beta$ then it is easy to verify that $c = 1 - 2^{-\beta} \geq \frac{1}{2}\min\{1, \beta\}$.

1. $\ell_*(X_i) > \widehat{\nu} + \gamma(j) \iff \ell_j(X_i) > \widehat{\nu} + \gamma(j)$
2. $\widehat{\nu} \le \ell_*(X_i) < \widehat{\nu} + \frac{1}{2}\gamma(j) \iff \widehat{\nu} + \frac{1}{2}\gamma(j) < \ell_j(X_i) \le \widehat{\nu} + \gamma(j)$
3. $\widehat{\nu} + \frac{1}{2}\gamma(j) \le \ell_*(X_i) \le \widehat{\nu} + \gamma(j) \iff \widehat{\nu} \le \ell_j(X_i) \le \widehat{\nu} + \frac{1}{2}\gamma(j)$

The first case holds since within that regime we have $\ell_j(x) = \ell_*(x)$, while the last two cases hold since they consider the regime where $\ell_j(x) = 2\widehat{\nu} + \gamma(j) - \ell_*(x)$. Thus, for any i such that $\ell_*(X_i) \in I_2$ it must be the case that $\ell_j(X_i) \in I_1$ and vice versa. Because $N_2 \ge N_1 > 0$, we conclude that if $\hat{\imath} = \arg\min_i \ell_{B/n}(X_i)$ then $\ell_{B/n}(X_{\hat{\imath}}) \in I_1$ and $\ell_*(X_i) \in I_2$. That is, $\nu_{\hat{\imath}} - \nu_* \ge \widehat{\nu} - \nu_* + \frac{1}{2}\gamma(j) = F^{-1}(\frac{\log(c/\delta)}{n+\log(c/\delta)}) - \nu_* + \frac{1}{2}\gamma(j)$. So if we wish $\nu_{\hat{\imath}} - \nu_* \le 2(F^{-1}(\frac{\log(c/\delta)}{n+\log(c/\delta)}) - \nu_*)$ with probability at least δ then we require $B/n = j \ge \gamma^{-1}\left(2(F^{-1}(\frac{\log(c/\delta)}{n+\log(c/\delta)}) - \nu_*)\right)$. □

2.7.5 Proof of Theorem 5

Proof. **Step 1: Simplify $\mathbf{H}(F,\gamma,n,\delta)$.** We begin by simplifying $\mathbf{H}(F,\gamma,n,\delta)$ in terms of just n,δ,α,β. In what follows, we use a constant c that may differ from one inequality to the next but remains an absolute constant that depends on α,β only. Let $p_n = \frac{\log(2/\delta)}{n}$ so that

$$\gamma^{-1}\left(\tfrac{F^{-1}(p_n)-\nu_*}{4}\right) \le c\left(F^{-1}(p_n) - \nu_*\right)^{-\alpha} \le c\, p_n^{-\alpha/\beta}$$

and

$$\int_{p_n}^1 \gamma^{-1}(\tfrac{F^{-1}(t)-\nu_*}{4})dt \le c\int_{p_n}^1 t^{-\alpha/\beta}dt \le \begin{cases} c\log(1/p_n) & \text{if } \alpha=\beta \\ c\frac{1-p_n^{1-\alpha/\beta}}{1-\alpha/\beta} & \text{if } \alpha\neq\beta. \end{cases}$$

We conclude that

$$\begin{aligned}\mathbf{H}(F,\gamma,n,\delta) &= 2n\int_{p_n}^1 \gamma^{-1}(\tfrac{F^{-1}(t)-\nu_*}{4})dt + \tfrac{10}{3}\log(2/\delta)\gamma^{-1}\left(\tfrac{F^{-1}(p_n)-\nu_*}{4}\right) \\ &\le c p_n^{-\alpha/\beta}\log(1/\delta) + cn\begin{cases}\log(1/p_n) & \text{if } \alpha=\beta \\ \frac{1-p_n^{1-\alpha/\beta}}{1-\alpha/\beta} & \text{if } \alpha\neq\beta.\end{cases}\end{aligned}$$

Step 2: Solve for $(B_{k,l}, n_{k,l})$ in terms of Δ. Fix $\Delta > 0$. Our strategy is to describe $n_{k,l}$ in terms of Δ. In particular, parameterize $n_{k,l}$ such that $p_{n_{k,l}} = c\frac{\log(4k^3/\delta)}{n_{k,l}} = \Delta^\beta$ so that $n_{k,l} = c\Delta^{-\beta}\log(4k^3/\delta)$ so

$$\begin{aligned}\mathbf{H}(F,\gamma,n_{k,l},\delta_{k,l}) &\le c p_{n_{k,l}}^{-\alpha/\beta}\log(1/\delta_{k,l}) + cn_{k,l}\begin{cases}\log(1/p_{n_{k,l}}) & \text{if } \alpha=\beta \\ \frac{1-p_{n_{k,l}}^{1-\alpha/\beta}}{1-\alpha/\beta} & \text{if } \alpha\neq\beta.\end{cases} \\ &\le c\log(k/\delta)\left[\Delta^{-\alpha} + \begin{cases}\Delta^{-\beta}\log(\Delta^{-1}) & \text{if } \alpha=\beta \\ \frac{\Delta^{-\beta}-\Delta^{-\alpha}}{1-\alpha/\beta} & \text{if } \alpha\neq\beta\end{cases}\right] \\ &\le c\log(k/\delta)\min\{\tfrac{1}{|1-\alpha/\beta|}, \log(\Delta^{-1})\}\Delta^{-\max\{\beta,\alpha\}}\end{aligned}$$

where the last line follows from

$$\Delta^{\max\{\beta,\alpha\}}\frac{\Delta^{-\beta}-\Delta^{-\alpha}}{1-\alpha/\beta} = \beta\frac{\Delta^{\max\{0,\alpha-\beta\}}-\Delta^{\max\{0,\beta-\alpha\}}}{\beta-\alpha}$$
$$= \beta\begin{cases}\frac{1-\Delta^{\beta-\alpha}}{\beta-\alpha} & \text{if } \beta>\alpha \\ \frac{1-\Delta^{\alpha-\beta}}{\alpha-\beta} & \text{if } \beta<\alpha\end{cases} \le c\min\{\tfrac{1}{|1-\alpha/\beta|},\log(\Delta^{-1})\}.$$

Using the upperbound $\lceil\log(n_{k,l})\rceil \le c\log(\log(k/\delta)\Delta^{-1}) \le c\log(\log(k/\delta))\log(\Delta^{-1})$ and letting $z_\Delta = \log(\Delta^{-1})^2\Delta^{-\max\{\beta,\alpha\}}$, we conclude that

$$\begin{aligned}B_{k,l} &< \min\{2^k : 2^k > 4\lceil\log(n_{k,l})\rceil\mathbf{H}(F,\gamma,n_{k,l},\delta_{k,l})\} \\ &< \min\{2^k : 2^k > c\log(k/\delta)\log(\log(k/\delta))z_\Delta\} \\ &\le cz_\Delta\log(\log(z_\Delta)/\delta)\log(\log(\log(z_\Delta)/\delta)) \\ &= cz_\Delta\overline{\log}(\log(z_\Delta)/\delta).\end{aligned}$$

Step 3: Count the total number of measurements. Moreover, the total number of measurements before $\hat{\imath}_{k,l}$ is output is upperbounded by

$$T = \sum_{i=1}^{k}\sum_{j=l}^{i}B_{i,j} \le k\sum_{i=1}^{k}B_{i,1} \le 2kB_{k,1} = 2B_{k,1}\log_2(B_{k,1})$$

where we have employed the so-called "doubling trick": $\sum_{i=1}^{k}B_{i,1} = \sum_{i=1}^{k}2^i \le 2^{k+1} = 2B_{k,i}$. Simplifying,

$$T \le cz_\Delta\overline{\log}(\log(z_\Delta)/\delta)\overline{\log}(z_\Delta\log(\log(z_\Delta)/\delta)) \le c\Delta^{-\max\{\beta,\alpha\}}\overline{\log}(\Delta^{-1})^3\overline{\log}(\log(\Delta^{-1})/\delta)$$

Solving for Δ in terms of T obtains

$$\Delta = c\left(\frac{\overline{\log}(T)^3\overline{\log}(\log(T)/\delta)}{T}\right)^{1/\max\{\alpha,\beta\}}.$$

Because the output arm is just the empirical best, there is some error associated with using the empirical estimate. The arm returned returned on round (k,l) is pulled $\lfloor\frac{2^{k-1}}{l}\rfloor \gtrsim B_{k,l}/\log(B_{k,l})$ times so the possible error is bounded by $\gamma(B_{k,l}/\log(B_{k,l})) \le c\left(\frac{\log(B_{k,l})}{B_{k,l}}\right)^{1/\alpha} \le c\left(\frac{\log(B)^2\log(\log(B))}{B}\right)^{1/\alpha}$ which is dominated by the value of Δ solved for above. □

2.7.6 Proof of Theorem 7

Proof. **Step 1: Simplify $\mathbf{H}(F,\gamma,n,\delta,\epsilon)$.** We begin by simplifying $\mathbf{H}(F,\gamma,n,\delta,\epsilon)$ in terms of just n,δ,α,β. As before, we use a constant c that may differ from one inequality to the next but remains an absolute constant. Let $p_n = \frac{\log(2/\delta)}{n}$ First we solve for ϵ by noting that we identify the best arm if $\nu_{\hat{\imath}} - \nu_* < \Delta_2$. Thus, if $\nu_{\hat{\imath}} - \nu_* \le \big(F^{-1}(p_n) - \nu_*\big) + \epsilon/2$ then we set

$$\epsilon = \max\big\{2\big(\Delta_2 - \big(F^{-1}(p_n) - \nu_*\big)\big), 4\big(F^{-1}(p_n) - \nu_*\big)\big\}$$

so that

$$\nu_{\hat{\imath}} - \nu_* \leq \max\left\{3\big(F^{-1}(p_n) - \nu_*\big), \Delta_2\right\} = \Delta_{\lceil \max\{2, cKp_n\}\rceil}.$$

We treat the case when $3\big(F^{-1}(p_n) - \nu_*\big) \leq \Delta_2$ and the alternative separately.

First assume $3\big(F^{-1}(p_n) - \nu_*\big) > \Delta_2$ so that $\epsilon = 4\big(F^{-1}(p_n) - \nu_*\big)$ and $\mathbf{H}(F, \gamma, n, \delta, \epsilon) = \mathbf{H}(F, \gamma, n, \delta)$. We also have

$$\gamma^{-1}\left(\tfrac{F^{-1}(p_n)-\nu_*}{4}\right) \leq c\left(F^{-1}\left(p_n\right) - \nu_*\right)^{-\alpha} \leq c\,\Delta^{-\alpha}_{\lceil p_n K\rceil}$$

and

$$\int_{p_n}^1 \gamma^{-1}(\tfrac{F^{-1}(t)-\nu_*}{4})dt = \int_{F^{-1}(p_n)}^1 \gamma^{-1}(\tfrac{x-\nu_*}{4})dF(x) \leq \frac{c}{K}\sum_{i=\lceil p_n K\rceil}^K \Delta_i^{-\alpha}$$

so that

$$\begin{aligned}\mathbf{H}(F, \gamma, n, \delta) &= 2n\int_{p_n}^1 \gamma^{-1}(\tfrac{F^{-1}(t)-\nu_*}{4})dt + \tfrac{10}{3}\log(2/\delta)\gamma^{-1}\left(\tfrac{F^{-1}(p_n)-\nu_*}{4}\right)\\ &\leq c\Delta^{-\alpha}_{\lceil p_n K\rceil}\log(1/\delta) + \frac{cn}{K}\sum_{i=\lceil p_n K\rceil}^K \Delta_i^{-\alpha}.\end{aligned}$$

Now consider the case when $3\big(F^{-1}(p_n) - \nu_*\big) \leq \Delta_2$. In this case $F(\nu_* + \epsilon/4) = 1/K$, $\gamma^{-1}\left(\frac{\epsilon}{16}\right) \leq c\Delta_2^{-\alpha}$, and $\int_{\nu_*+\epsilon/4}^\infty \gamma^{-1}(\frac{t-\nu_*}{4})dF(t) \leq c\sum_{i=2}^K \Delta_i^{-\alpha}$ so that

$$\begin{aligned}\mathbf{H}(F, \gamma, n, \delta, \epsilon) &= 2n\int_{\nu_*+\epsilon/4}^\infty \gamma^{-1}(\tfrac{t-\nu_*}{4})dF(t) + \left(\tfrac{4}{3}\log(2/\delta) + 2nF(\nu_* + \epsilon/4)\right)\gamma^{-1}\left(\tfrac{\epsilon}{16}\right)\\ &\leq c(\log(1/\delta) + n/K)\Delta_2^{-\alpha} + \frac{cn}{K}\sum_{i=2}^K \Delta_i^{-\alpha}.\end{aligned}$$

Step 2: Solve for $(B_{k,l}, n_{k,l})$ in terms of Δ. Note there is no improvement possible once $p_{n_{k,l}} \leq 1/K$ since $3\big(F^{-1}(1/K) - \nu_*\big) \leq \Delta_2$. That is, when $p_{n_{k,l}} \leq 1/K$ the algorithm has found the best arm but will continue to take samples indefinetely. Thus, we only consider the case when $q = 1/K$ and $q > 1/K$. Fix $\Delta > 0$. Our strategy is to describe $n_{k,l}$ in terms of q. In particular, parameterize $n_{k,l}$ such that $p_{n_{k,l}} = c\frac{\log(4k^3/\delta)}{n_{k,l}} = q$ so that $n_{k,l} = cq^{-1}\log(4k^3/\delta)$ so

$$\begin{aligned}\mathbf{H}(F, \gamma, n_{k,l}, \delta_{k,l}, \epsilon_{k,l}) &\leq c\begin{cases}(\log(1/\delta_{k,l}) + \frac{n_{k,l}}{K})\Delta_2^{-\alpha} + \frac{n_{k,l}}{K}\sum_{i=2}^K \Delta_i^{-\alpha} & \text{if } 5\big(F^{-1}(p_{n_{k,l}}) - \nu_*\big) \leq \Delta_2\\ \Delta^{-\alpha}_{\lceil p_{n_{k,l}} K\rceil}\log(1/\delta_{k,l}) + \frac{n_{k,l}}{K}\sum_{i=\lceil p_{n_{k,l}} K\rceil}^K \Delta_i^{-\alpha} & \text{if otherwise}\end{cases}\\ &\leq c\log(k/\delta)\begin{cases}\Delta_2^{-\alpha} + \sum_{i=2}^K \Delta_i^{-\alpha} & \text{if } q = 1/K\\ \Delta^{-\alpha}_{\lceil qK\rceil} + \frac{1}{qK}\sum_{i=\lceil qK\rceil}^K \Delta_i^{-\alpha} & \text{if } q > 1/K.\end{cases}\\ &\leq c\log(k/\delta)\Delta^{-\alpha}_{\lceil \max\{2, qK\}\rceil} + \frac{1}{qK}\sum_{i=\lceil \max\{2, qK\}\rceil}^K \Delta_i^{-\alpha}\end{aligned}$$

Using the upperbound $\lceil \log(n_{k,l}) \rceil \leq c \log(\log(k/\delta)q^{-1}) \leq c \log(\log(k/\delta)) \log(q^{-1})$ and letting $z_q = \log(q^{-1})(\Delta^{-\alpha}_{\lceil \max\{2,qK\} \rceil} + \frac{1}{qK} \sum_{i=\lceil \max\{2,qK\} \rceil}^{K} \Delta_i^{-\alpha})$, we apply the exact sequence of steps as in the proof of Theorem 5 to obtain

$$T \leq c z_q \overline{\log}(\log(z_q)/\delta) \overline{\log}(z_q \log(\log(z_q)/\delta))$$

Because the output arm is just the empirical best, there is some error associated with using the empirical estimate. The arm returned on round (k, l) is pulled $\lceil \frac{2^{k-1}}{l} \rceil \geq c B_{k,l} / \log(B_{k,l})$ times so the possible error is bounded by $\gamma(B_{k,l}/\log(B_{k,l})) \leq c \left(\frac{\log(B_{k,l})}{B_{k,l}} \right)^{1/\alpha} \leq c \left(\frac{\log(T)^2 \log(\log(T))}{T} \right)^{1/\alpha}$. This is dominated by $\Delta_{\lceil \max\{2,qK\} \rceil}$ for the value of T prescribed by the above calculation, completing the proof. □

2.7.7 Proof of Theorem 8

Proof. Let s denote the index of the last stage, to be determined later. If $\widetilde{r}_k = R\eta^{k-s}$ and $\widetilde{n}_k = n\eta^{-k}$ so that $r_k = \lfloor \widetilde{r}_k \rfloor$ and $n_k = \lfloor \widetilde{n}_k \rfloor$ then

$$\sum_{k=0}^{s} n_k r_k \leq \sum_{k=0}^{s} \widetilde{n}_k \widetilde{r}_k = nR(s+1)\eta^{-s} \leq B$$

since, by definition, $s = \min\{t \in \mathbb{N} : nR(t+1)\eta^{-t} \leq B, t \leq \log_\eta(\min\{R, n\})\}$. It is straightforward to verify that $B \geq z_{SH}$ ensures that $r_0 \geq 1$ and $n_s \geq 1$.

The proof for Theorem 1 holds here with a few modifications. First, we derive a lower bound on the resource per arm r_k per round if $B \geq z_{SH}$ with generalized elimination rate η:

$$\begin{aligned}
r_k &\geq \frac{B}{|S_k|(\lfloor \log_\eta(n) \rfloor + 1)} - 1 \\
&\geq \frac{\eta}{|S_k|} \max_{i=2,\ldots,n} i \left(1 + \min\left\{ R, \gamma^{-1}\left(\max\left\{ \frac{\epsilon}{4}, \frac{\nu_i - \nu_1}{2} \right\} \right) \right\} \right) - 1 \\
&\geq \frac{\eta}{|S_k|} \left(\lfloor |S_k|/\eta \rfloor + 1 \right) \left(1 + \min\left\{ R, \gamma^{-1}\left(\max\left\{ \frac{\epsilon}{4}, \frac{\nu_{\lfloor |S_k|/2 \rfloor + 1} - \nu_1}{2} \right\} \right) \right\} \right) - 1 \\
&\geq \left(1 + \min\left\{ R, \gamma^{-1}\left(\max\left\{ \frac{\epsilon}{4}, \frac{\nu_{\lfloor |S_k|/2 \rfloor + 1} - \nu_1}{2} \right\} \right) \right\} \right) - 1 \\
&= \min\left\{ R, \gamma^{-1}\left(\max\left\{ \frac{\epsilon}{4}, \frac{\nu_{\lfloor |S_k|/2 \rfloor + 1} - \nu_1}{2} \right\} \right) \right\}.
\end{aligned}$$

Also, note that $\gamma(R) = 0$, hence if the minimum is ever active, $\ell_{i,R} = \nu_i$ and we know the true loss. The rest of the proof is same as that for Theorem 1 for η in place of 2.

In addition, we note that

$$\max_{i=n_s+1,\ldots,n} i\, \gamma^{-1}\left(\max\left\{ \tfrac{\epsilon}{4}, \tfrac{\nu_i - \nu_1}{2} \right\} \right) \leq n_s \gamma^{-1}\left(\max\left\{ \tfrac{\epsilon}{4}, \tfrac{\nu_{n_s+1} - \nu_1}{2} \right\} \right) + \sum_{i > n_s} \gamma^{-1}\left(\max\left\{ \tfrac{\epsilon}{4}, \tfrac{\nu_i - \nu_1}{2} \right\} \right).$$

□

Chapter 3

Hyperparameter Optimization in the Large-Scale Regime

In the previous chapter, we presented HYPERBAND for fast hyperparameter optimization with early-stopping. However, as we will discuss in Section 3.3, HYPERBAND is poorly suited for the parallel setting since it is susceptible to stragglers and dropped jobs. In this chapter, we address these issues with a massively parallel asynchronous algorithm called ASHA for **A**synchronous **S**uccessive **H**alving **A**lgorithm. Recall from Chapter 2, the successive halving algorithm (SHA) serves as the inner loop for HYPERBAND, with HYPERBAND automating the choice of the early-stopping rate by running different variants of SHA. While the appropriate choice of early stopping rate is problem dependent, our empirical results in the previous chapter demonstrated that aggressive early-stopping works well for a wide variety of tasks. Hence, we focus on parallelizing successive halving in this chapter. Additionally, the experiments in Section 3.4 provide a refresh over those in Chapter 2.4 by comparing to more recent hyperparameter optimization approaches on more difficult hyperparameter tuning problems.

3.1 Introduction

As alluded to in Chapter 1.4, three trends in modern machine learning motivate a new approach to hyperparameter tuning:

1. **High-dimensional hyperparameter spaces**. Machine learning models are becoming increasingly complex, as evidenced by modern neural networks with dozens of hyperparameters. For such complex models with hyperparameters that interact in unknown ways, a practitioner is forced to evaluate potentially thousands of different hyperparameter settings.
2. **Increasing training times**. As datasets grow larger and models become more complex, training a model has become dramatically more expensive, often taking days or weeks on specialized high-performance hardware. This trend is particularly onerous in the context of hyperparameter tuning, as a new model must be trained to evaluate each candidate hyperparameter configuration.
3. **Rise of parallel computing**. The combination of a growing number of hyperparameters

and longer training time per model precludes evaluating configurations sequentially; we simply cannot wait months or years to find a suitable hyperparameter setting. Leveraging parallel and distributed computational resources presents a solution to the increasingly challenging problem of hyperparameter optimization.

Our goal is to design a hyperparameter tuning algorithm that can effectively leverage parallel resources. This goal seems trivial with standard methods like random search, where we could train different configurations in an embarrassingly parallel fashion. However, for high-dimensional search spaces, the number of candidate configurations required to find a good configuration often dwarfs the number of available parallel resources, and when possible, our goal is to:

Evaluate orders of magnitude more hyperparameter configurations than available parallel workers in a small multiple of the wall-clock time needed to train a single model.

We denote this setting the *large-scale regime* for hyperparameter optimization.

In the remainder of this chapter, after discussing related work, we present ASHA and show that it successfully addresses the large-scale regime. In particular, we perform a thorough comparison of several hyperparameter tuning methods in both the sequential and parallel settings. We focus on 'mature' methods, i.e., well-established techniques that have been empirically and/or theoretically studied to an extent that they could be considered for adoption in a production-grade system. In the sequential setting, we compare SHA with Fabolas [Klein et al., 2017a], Population Based Tuning (PBT) [Jaderberg et al., 2017], and BOHB [Falkner et al., 2018], state-of-the-art methods that exploit partial training. Our results show that SHA outperforms these methods, which when coupled with SHA's simplicity and theoretical grounding, motivate the use of a SHA-based method in production. We further verify that SHA and ASHA achieve similar results. In the parallel setting, our experiments demonstrate that ASHA addresses the intrinsic issues of parallelizing SHA, scales linearly with the number of workers, and exceeds the performance of PBT, BOHB, and Vizier [Golovin et al., 2017], Google's internal hyperparameter optimization service.

Finally, based on our experience developing ASHA within Determined AI's production-quality machine learning system that offers hyperparameter tuning as a service, we describe several systems design decisions and optimizations that we explored as part of the implementation. We focus on four key considerations: (1) streamlining the user interface to enhance usability; (2) autoscaling parallel training to systematically balance the tradeoff between lower latency in individual model training and higher throughput in total configuration evaluation; (3) efficiently scheduling ML jobs to optimize multi-tenant cluster utilization; and (4) tracking parallel hyperparameter tuning jobs for reproducibility.

3.2 Related Work

We will first discuss related work that motivated our focus on parallelizing SHA for the large-scale regime. We then provide an overview of methods for parallel hyperparameter tuning, from which we identify a mature subset to compare to in our empirical studies (Section 3.4). Finally, we discuss related work on systems for hyperparameter optimization.

Sequential Methods. Klein et al. [2017a] reported state-of-the-art performance for Fabolas on several tasks in comparison to Hyperband and other leading methods. However, our results in Section 3.4.1 demonstrate that under an appropriate experimental setup, SHA and Hyperband

in fact outperform Fabolas. Moreover, we note that Fabolas, along with most other Bayesian optimization approaches, can be parallelized using a constant liar (CL) type heuristic [Ginsbourger et al., 2010, González et al., 2016]. However, the parallel version will underperform the sequential version, since the latter uses a more accurate posterior to propose new points. Hence, our comparisons to these methods are restricted to the sequential setting.

Other hybrid approaches combine Hyperband with adaptive sampling. For example, Klein et al. [2017b] combined Bayesian neural networks with Hyperband by first training a Bayesian neural network to predict learning curves and then using the model to select promising configurations to use as inputs to Hyperband. More recently, Falkner et al. [2018] introduced BOHB, a hybrid method combining Bayesian optimization with Hyperband. They also propose a parallelization scheme for SHA that retains synchronized eliminations of underperforming configurations. We discuss the drawbacks of this parallelization scheme in Section 3.3 and demonstrate that ASHA outperforms this version of parallel SHA as well as BOHB in Section 3.4.2. We note that similar to SHA/Hyperband, ASHA can be combined with adaptive sampling for more robustness to certain challenges of parallel computing that we discuss in Section 3.3.

Parallel Methods. Established parallel methods for hyperparameter tuning include PBT [Jaderberg et al., 2017, Li et al., 2019a] and Vizier [Golovin et al., 2017]. PBT is a state-of-the-art hybrid evolutionary approach that exploits partial training to iteratively increase the fitness of a population of models. In contrast to Hyperband, PBT lacks any theoretical guarantees. Additionally, PBT is primarily designed for neural networks and is not a general approach for hyperparameter tuning. We further note that PBT is more comparable to SHA than to Hyperband since both PBT and SHA require the user to set the early-stopping rate via internal hyperparameters.

Vizier is Google's black-box optimization service with support for multiple hyperparameter optimization methods and early-stopping options. For succinctness, we will refer to Vizier's default algorithm as "Vizier" although it is simply one of methods available on the Vizier platform. While Vizier provides early-stopping rules, the strategies only offer approximately $3\times$ speedup in contrast to the order of magnitude speedups observed for SHA. We compare to PBT and Vizier in Section 3.4.2 and Section 3.4.5, respectively.

Hyperparameter Optimization Systems. While there is a large body of work on systems for machine learning, we narrow our focus to systems for hyperparameter optimization. AutoWEKA [Kotthoff et al., 2017] and AutoSklearn [Feurer et al., 2015a] are two established single-machine, single-user systems for hyperparameter optimization. Existing systems for distributed hyperparameter optimization include Vizier [Golovin et al., 2017], RayTune [Liaw et al., 2018], CHOPT [Kim et al., 2018] and Optuna [Akiba et al.]. These existing systems provide generic support for a wide range of hyperparameter tuning algorithms; both RayTune and Optuna in fact have support for ASHA. In contrast, our work focuses on a specific algorithm—ASHA—that we argue is particularly well-suited for massively parallel hyperparameter optimization. We further introduce a variety of systems optimizations designed specifically to improve the performance, usability, and robustness of ASHA in production environments. We believe that these optimizations would directly benefit existing systems to effectively support ASHA, and generalizations of these optimizations could also be beneficial in supporting other hyperparameter tuning algorithms.

Similarly, we note that Kim et al. [2018] address the problem of resource management for generic hyperparameter optimization methods in a shared compute environment, while we focus

on efficient resource allocation with adaptive scheduling specifically for ASHA in Section 3.5.3. Additionally, in contrast to the user-specified automated scaling capability for parallel training presented in Xiao et al. [2018], we propose to automate appropriate autoscaling limits by using the performance prediction framework by Qi et al. [2017].

3.3 ASHA Algorithm

We start with a discussion of naive ways of parallelizing synchronous successive halving. Then we present ASHA and discuss how it addresses issues with synchronous SHA and improves upon the original algorithm.

3.3.1 Synchronous SHA

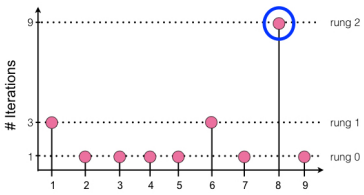
Algorithm 2: Successive Halving Algorithm.

Input: number of configurations n, minimum resource r, maximum resource R, reduction factor η, minimum early-stopping rate s
$s_{\max} = \lfloor \log_\eta(R/r) \rfloor$
assert $n \geq \eta^{s_{\max}-s}$ so that at least one configuration will be allocated R.
$T = \text{get\_hyperparameter\_configuration}(n)$
for $i \in \{0, \ldots, s_{\max} - s\}$ **do**
  // All configurations trained for a given i constitute a "rung."
  $n_i = \lfloor n\eta^{-i} \rfloor$
  $r_i = r\eta^{i+s}$
  $L = \text{run\_then\_return\_val\_loss}(\theta, r_i) : \theta \in T$
  $T = \text{top\_k}(T, L, n_i/\eta)$
end
return *best configuration in* T

To keep this chapter self-contained and for consistency, we present SHA in Algorithm 2 using the same notation and terminology that we will use for ASHA in the next section. Here, n denotes the number of configurations, r the minimum resource per configuration, R the maximum resource per configuration, $\eta \geq 2$ the reduction factor, and s the early-stopping rate. Additionally, the $\text{get\_hyperparameter\_configuration}(n)$ subroutine returns n configurations sampled randomly from a given search space; and the run_then_ $\text{return\_val\_loss}(\theta, r)$ subroutine returns the validation loss after training the model with the hyperparameter setting θ and for r resources. For a given early-stopping rate s, a minimum resource of $r_0 = r\eta^s$ will be allocated to each configuration. Hence, lower s corresponds to more aggressive early-stopping, with $s = 0$ prescribing a minimum resource of r. As in the previous chapter, we will refer to SHA with different values of s as *brackets* and, within a bracket, we will refer to each round of promotion as a *rung* with the base rung numbered 0 and increasing. In order to have a concrete example to refer to later, Figure 3.1(a) shows the rungs for bracket 0 for an example setting with $n = 9, r = 1, R = 9$,

and $\eta = 3$, while Figure 3.1(b) shows how resource allocations change for different brackets s. Namely, the starting budget per configuration $r_0 \leq R$ increases by a factor of η per increment of s.

Straightforward ways of parallelizing SHA are not well suited for the parallel regime. We could consider the embarrassingly parallel approach of running multiple instances of SHA, one on each worker. However, this strategy is not well suited for the large-scale regime, where we would like results in little more than the time to train one configuration. To see this, assume that training time for a configuration scales linearly with the allocated resource and $time(R)$ represents the time required to train a configuration for the maximum resource R. In general, for a given bracket s, the minimum time to return a configuration trained to completion is $(\log_\eta(R/r) - s + 1) \times time(R)$, where $\log_\eta(R/r) - s + 1$ counts the number of rungs. For example, consider Bracket 0 in the toy example in Figure 3.1. The time needed to return a fully trained configuration is $3 \times time(R)$, since there are three rungs and each rung is allocated R resource. In contrast, as we will see in the next section, our parallelization scheme for SHA can return an answer in just $time(R)$.



(a) Visual depiction of the promotion scheme for bracket $s = 0$.

bracket s	rung i	n_i	r_i	total budget
0	0	9	1	9
	1	3	3	9
	2	1	9	9
1	0	9	3	27
	1	3	9	27
2	0	9	9	81

(b) Promotion scheme for different brackets s.

Figure 3.1: **Promotion scheme for SHA** with $n = 9$, $r = 1$, $R = 9$, and $\eta = 3$.

Another naive way of parallelizing SHA is to distribute the training of the n/η^k surviving configurations on each rung k as is done by Falkner et al. [2018] and add brackets when there are no jobs available in existing brackets. We will refer to this method as "synchronous" SHA. The efficacy of this strategy is severely hampered by two issues: (1) SHA's synchronous nature is sensitive to stragglers and dropped jobs as every configuration within a rung must complete before proceeding to the next rung, and (2) the estimate of the top $1/\eta$ configurations for a given early-stopping rate does not improve as more brackets are run since promotions are performed independently for each bracket. We will demonstrate the susceptibility of synchronous SHA to stragglers and dropped jobs on simulated workloads in Section 3.3.3.

3.3.2 Asynchronous SHA (ASHA)

We now introduce ASHA as an effective technique to parallelize SHA, leveraging asynchrony to mitigate stragglers and maximize parallelism. Intuitively, ASHA promotes configurations to the next rung whenever possible instead of waiting for a rung to complete before proceeding to the next rung. Additionally, if no promotions are possible, ASHA simply adds a configuration to the base rung, so that more configurations can be promoted to the upper rungs. ASHA is formally defined in Algorithm 3. Given its asynchronous nature it does not require the user to pre-specify

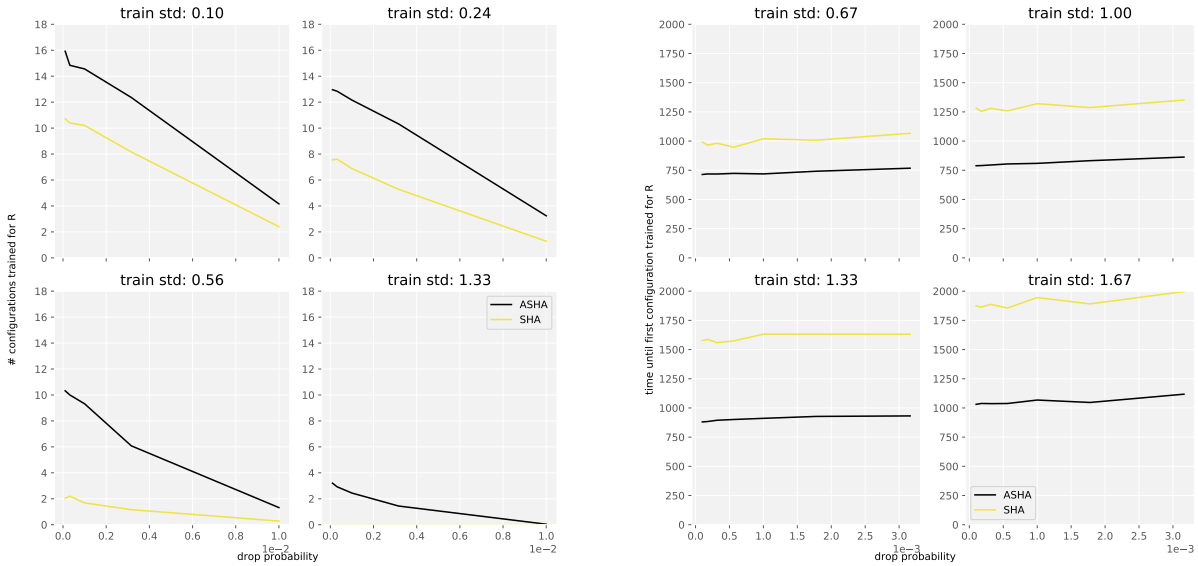
Algorithm 3: Asynchronous Successive Halving Algorithm.

Input: minimum resource r, maximum resource R, reduction factor η, minimum early-stopping rate s
Algorithm ASHA()
  repeat
    for each free worker **do**
      $(\theta, k) =$ get_job()
      run_then_return_val_loss$(\theta, r\eta^{s+k})$
    end
    for completed job (θ, k) with loss l **do**
      Update configuration θ in rung k with loss l.
    end
Procedure get_job()
  // Check to see if there is a promotable config.
  for $k = \lfloor \log_\eta(R/r) \rfloor - s - 1, \ldots, 1, 0$ **do**
    candidates = top_k(rung k, |rung k|$/\eta$)
    promotable = $\{t$ for $t \in$ candidates if t not already promoted$\}$
    if |promotable| > 0 **then**
      return promotable$[0], k+1$
    end
  end
  Draw random configuration θ. // If not, grow bottom rung.
  return $\theta, 0$

the number of configurations to evaluate, but it otherwise requires the same inputs as SHA. Note that the run_then_return_val_loss subroutine in ASHA is asynchronous and the code execution continues after the job is passed to the worker. ASHA's promotion scheme is laid out in the get_job subroutine.

ASHA is well-suited for the large-scale regime, where wall-clock time is constrained to a small multiple of the time needed to train a single model. For ease of comparison with SHA, assume training time scales linearly with the resource. Consider the example of Bracket 0 shown in Figure 3.1, and assume we can run ASHA with 9 machines. Then ASHA returns a fully trained configuration in $^{13}/_{9} \times time(R)$, since 9 machines are sufficient to promote configurations to the next rung in the same time it takes to train a single configuration in the rung. Hence, the training time for a configuration in rung 0 is $^{1}/_{9} \times time(R)$, for rung 1 it is $^{1}/_{3} \times time(R)$, and for rung 2 it is $time(R)$. In general, $\eta^{\log_\eta(R)-s}$ machines are needed to advance a configuration to the next rung in the same time it takes to train a single configuration in the rung, and it takes $\eta^{s+i-\log_\eta(R)} \times time(R)$ to train a configuration in rung i. Hence, ASHA can return a configuration trained to completion in time

$$\left(\sum_{i=s}^{\log_\eta(R)} \eta^{i-\log_\eta(R)} \right) \times time(R) \leq 2\, time(R).$$



(a) Average number of configurations trained on R resource.

(b) Average time before a configuration is trained on R resource.

Figure 3.2: **Simulated workloads comparing impact of stragglers and dropped jobs.** The number of configurations trained for R resource (left) is higher for ASHA than synchronous SHA when the standard deviation is high. Additionally, the average time before a configuration is trained for R resource (right) is lower for ASHA than for synchronous SHA when there is high variability in training time (i.e., stragglers). Hence, ASHA is more robust to stragglers and dropped jobs than synchronous SHA since it returns a completed configuration faster and returns more configurations trained to completion.

Moreover, when training is iterative, ASHA can return an answer in $time(R)$, since incrementally trained configurations can be checkpointed and resumed.

Finally, since Hyperband simply runs multiple SHA brackets, we can asynchronously parallelize Hyperband by either running multiple brackets of ASHA or looping through brackets of ASHA sequentially as is done in the original Hyperband. We employ the latter looping scheme for asynchronous Hyperband in the next section.

3.3.3 Comparison of Synchronous SHA and ASHA

We now verify that ASHA is indeed more robust to stragglers and dropped jobs using simulated workloads. For these simulations, we run synchronous SHA with $\eta = 4$, $r = 1$, $R = 256$, and $n = 256$ and ASHA with the same values and the maximum early-stopping rate $s = 0$. Note that BOHB [Falkner et al., 2018], one of the competitors we empirically compare to in Section 3.4, is also susceptible to stragglers and dropped jobs since it uses synchronous SHA as its parallelization scheme but leverages Bayesian optimization to perform adaptive sampling.

For these synthetic experiments, we assume that the expected training time for each job is the same as the allocated resource. We simulate stragglers by multiplying the expected training time

by $(1 + |z|)$ where z is drawn from a normal distribution with mean 0 and a specified standard deviation. We simulated dropped jobs by assuming that there is a given p probability that a job will be dropped at each time unit, hence, for a job with a runtime of 256 units, the probability that it is not dropped is $1 - (1 - p)^{256}$.

Figure 3.2 shows the number of configurations trained to completion (left) and time required before one configuration is trained to completion (right) when running synchronous SHA and ASHA using 25 workers. For each combination of training time standard deviation and drop probability, we simulate ASHA and synchronous SHA 25 times and report the average. As can be seen in Figure 3.2a, ASHA trains many more configurations to completion than synchronous SHA when the standard deviation is high; we hypothesize that this is one reason ASHA performs significantly better than synchronous SHA and BOHB for the second benchmark in Section 3.4.2. Figure 3.2b shows that ASHA returns a configuration trained for the maximum resource R much faster than synchronous SHA when there is high variability in training time (i.e., stragglers) and high risk of dropped jobs. Although ASHA is more robust than synchronous SHA to stragglers and dropped jobs on these simulated workloads, we nonetheless compare synchronous SHA in Section 3.4.5 and show that ASHA performs better.

3.3.4 Algorithm Discussion

ASHA is able to remove the bottleneck associated with synchronous promotions by incurring a small number of incorrect promotions, i.e. configurations that were promoted early on but are not in the top $1/\eta$ of configurations in hindsight. By the law of large numbers, we expect to erroneously promote a vanishing fraction of configurations in each rung as the number of configurations grows. Intuitively, in the first rung with n evaluated configurations, the number of mispromoted configurations is roughly $\sqrt{n}$, since the process resembles the convergence of an empirical cumulative distribution function (CDF) to its expected value [Dvoretzky et al., 1956]. For later rungs, although the configurations are no longer i.i.d. since they were advanced based on the empirical CDF from the rung below, we expect this dependence to be weak.

We further note that ASHA improves upon SHA in two ways. First, in Chapter 2.5 we discussed two SHA variants: finite horizon (bounded resource R per configuration) and infinite horizon (unbounded resources R per configuration). ASHA consolidates these settings into one algorithm. In Algorithm 3, we do not promote configurations that have been trained for R, thereby restricting the number of rungs. However, this algorithm trivially generalizes to the infinite horizon; we can remove this restriction so that the maximum resource per configuration increases naturally as configurations are promoted to higher rungs. In contrast, SHA does not naturally extend to the infinite horizon setting, as it relies on the doubling trick and must rerun brackets with larger budgets to increase the maximum resource.

Additionally, SHA does not return an output until a single bracket completes. In the finite horizon this means that there is a constant interval of (# of rungs $\times$ $time(R)$) between receiving outputs from SHA. In the infinite horizon this interval doubles between outputs. In contrast, ASHA grows the bracket incrementally instead of in fixed budget intervals. To further reduce latency, ASHA uses intermediate losses to determine the current best performing configuration, as opposed to only considering the final SHA outputs.

3.4 Experiments

We start with experiments in the sequential setting to verify that SHA/Hyperband achieves state-of-the-art performance and study the impact of asynchronous promotions on the performance of ASHA. We next evaluate ASHA in parallel environments on three benchmark tasks.

3.4.1 Sequential Experiments

We benchmark Hyperband and SHA against PBT, BOHB (synchronous SHA with Bayesian optimization as introduced by Falkner et al. [2018]), and Fabolas, and examine the relative performance of SHA versus ASHA and Hyperband versus asynchronous Hyperband. As mentioned previously, asynchronous Hyperband loops through brackets of ASHA with different early-stopping rates.

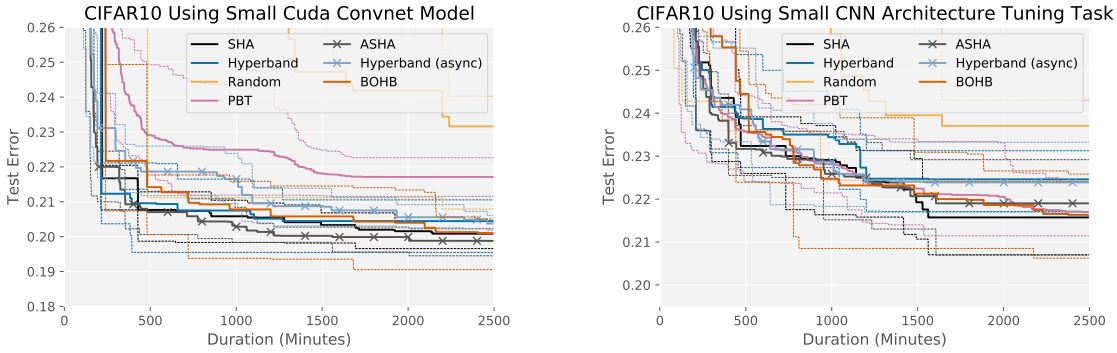


Figure 3.3: **Sequential experiments** (1 worker). Average across 10 trials is shown for each hyperparameter optimization method. Gridded lines represent top and bottom quartiles of trials.

We compare ASHA against PBT, BOHB, and synchronous SHA on two benchmarks for CIFAR-10: (1) tuning a convolutional neural network (CNN) with the cuda-convnet architecture and the same search space as that considered in the experiments in Chapter 2.4.1; and (2) tuning a CNN architecture with varying number of layers, batch size, and number of filters. The details for the search spaces considered and the settings we used for each search method can be found in Section 3.6.2. Note that BOHB uses SHA to perform early-stopping and differs only in how configurations are sampled; while SHA uses random sampling, BOHB uses Bayesian optimization to adaptively sample new configurations. In the following experiments, we run BOHB using the same early-stopping rate as SHA and ASHA instead of looping through brackets with different early-stopping rates as is done by Hyperband.

The results on these two benchmarks are shown in Figure 3.3. On benchmark 1, Hyperband and all variants of SHA (i.e., SHA, ASHA, and BOHB) outperform PBT by $3\times$. On benchmark 2, while all methods comfortably beat random search, SHA, ASHA, BOHB and PBT performed similarly and slightly outperform Hyperband and asynchronous Hyperband. This last observation (i) corroborates the results in Chapter 2.4, which found that the brackets with the most aggressive early-stopping rates performed the best; and (ii) follows from the discussion in Section 3.2 noting that PBT is more similar in spirit to SHA than Hyperband, as PBT / SHA both require

user-specified early-stopping rates (and are more aggressive in their early-stopping behavior in these experiments). We observe that SHA and ASHA are competitive with BOHB, despite the adaptive sampling scheme used by BOHB. Additionally, for both tasks, introducing asynchrony does not consequentially impact the performance of ASHA (relative to SHA) or asynchronous Hyperband (relative to Hyperband). This not surprising; as discussed in Section 3.3.4, we expect the number of ASHA mispromotions to be square root in the number of configurations.

Finally, due to the nuanced nature of the evaluation framework used by Klein et al. [2017a], we present our results on 4 different benchmarks comparing Hyperband to Fabolas in Section 3.6.1. In summary, our results show that Hyperband, specifically the first bracket of SHA, tends to outperform Fabolas while also exhibiting lower variance across experimental trials.

3.4.2 Limited-Scale Distributed Experiments

We next compare ASHA to synchronous SHA, the parallelization scheme discussed in Section 3.3.1; BOHB; and PBT on the same two tasks. For each experiment, we run each search method with 25 workers for 150 minutes. We use the same setups for ASHA and PBT as in the previous section. We run synchronous SHA and BOHB with default settings and the same η and early-stopping rate as ASHA.

Figure 3.4 shows the average test error across 5 trials for each search method. On benchmark 1, ASHA evaluated over 1000 configurations in just over 40 minutes with 25 workers and found a good configuration (error rate below 0.21) in approximately the time needed to train a single model, whereas it took ASHA nearly 400 minutes to do so in the sequential setting (Figure 3.3). Notably, we only achieve a $10\times$ speedup on 25 workers due to the relative simplicity of this task, i.e., it only required evaluating a few hundred configurations to identify a good one in the sequential setting. In contrast, for the more difficult search space used in benchmark 2, we observe linear speedups with ASHA, as the ~ 700 minutes needed in the sequential setting (Figure 3.3) to reach a test error below 0.23 is reduced to under 25 minutes in the distributed setting.

Compared to synchronous SHA and BOHB, ASHA finds a good configuration $1.5\times$ as fast on benchmark 1 while BOHB finds a slightly better final configuration. On benchmark 2, ASHA performs significantly better than synchronous SHA and BOHB due to the higher variance in training times between configurations (the average time required to train a configuration on the maximum resource R is 30 minutes with a standard deviation of 27 minutes), which exacerbates the sensitivity of synchronous SHA to stragglers (see Section 3.3.3). BOHB actually underperforms synchronous SHA on benchmark 2 due to its bias towards more computationally expensive configurations, reducing the number of configurations trained to completion within the given time frame.

We further note that ASHA outperforms PBT on benchmark 1; in fact the minimum and maximum range for ASHA across 5 trials does not overlap with the average for PBT. On benchmark 2, PBT slightly outperforms asynchronous Hyperband and performs comparably to ASHA. However, note that the ranges for the searchers share large overlap and the result is likely not significant. Overall, ASHA outperforms PBT, BOHB and SHA on these two tasks. This improved performance, coupled with the fact that it is a more principled and general approach than either BOHB or PBT (e.g., agnostic to resource type and robust to hyperparameters that change the size of the model), further motivates its use for the large-scale regime.

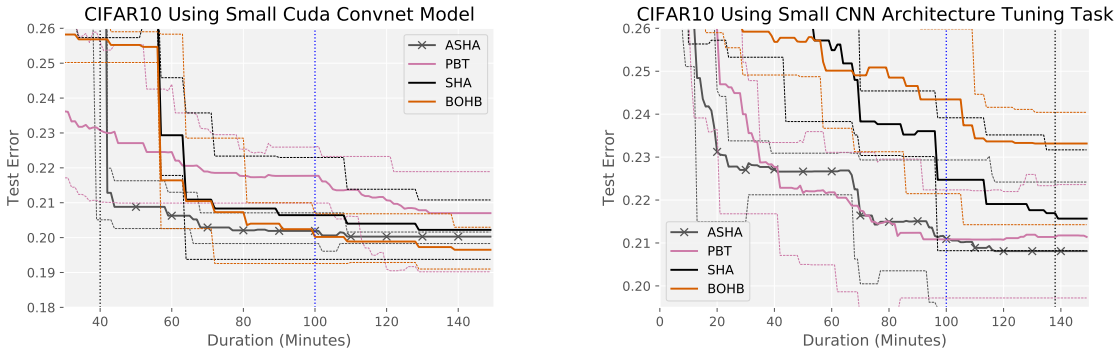


Figure 3.4: **Limited-scale distributed experiments** with 25 workers. For each searcher, the average test error across 5 trials is shown in each plot. The light dashed lines indicate the min/max ranges. The dotted black line represents the time needed to train the most expensive model in the search space for the maximum resource R. The dotted blue line represents the point at which 25 workers in parallel have performed as much work as a single machine in the sequential experiments (Figure 3.3).

3.4.3 Tuning Neural Network Architectures

Motivated by the emergence of neural architecture search (NAS) as a specialized hyperparameter optimization problem, we evaluate ASHA and competitors on two NAS benchmarks: (1) designing convolutional neural networks (CNN) for CIFAR-10 and (2) designing recurrent neural networks (RNN) for Penn Treebank [Marcus et al., 1993]. We use the same search spaces as that considered by Liu et al. [2019] (see Section 3.6.4 for more details).

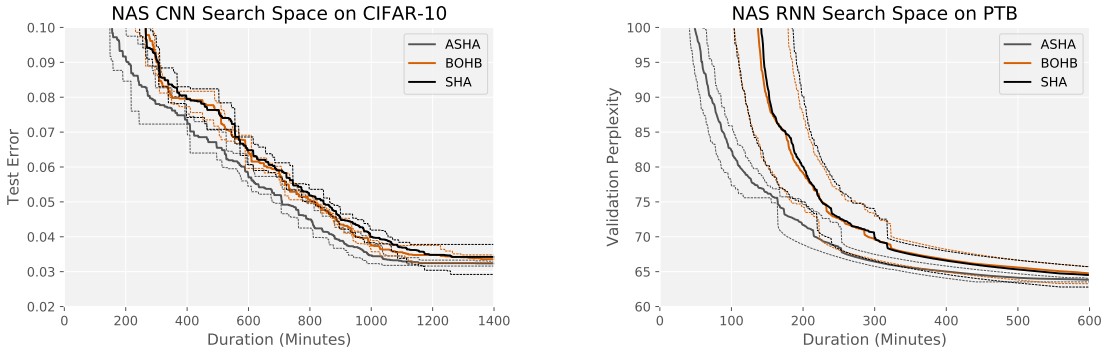


Figure 3.5: **Tuning neural network architectures** with 16 workers. For each searcher, the average test error across 4 trials is shown in each plot. The light dashed lines indicate the min/max ranges.

For both benchmarks, we ran ASHA, SHA, and BOHB on 16 workers with $\eta = 4$ and a maximum resource of $R = 300$ epochs. The results in Figure 3.5 shows ASHA outperforms SHA and BOHB on both benchmarks. Our results for CNN search show that ASHA finds an architecture with test error below 10% nearly twice as fast as SHA and BOHB. ASHA also finds final architectures with lower test error on average: 3.24% for ASHA vs 3.42% for SHA

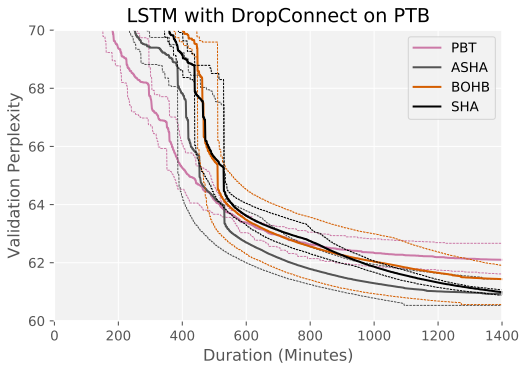


Figure 3.6: **Modern LSTM benchmark** with DropConnect [Merity et al., 2018] using 16 GPUs. The average across 5 trials is shown, with dashed lines indicating min/max ranges.

and 3.36% for BOHB. Our results for RNN search show that ASHA finds an architecture with validation perplexity below 80 nearly trice as fast as SHA and BOHB and also converges an architecture with lower perplexity: 63.5 for ASHA vs 64.3 for SHA and 64.2 for BOHB. Note that vanilla PBT is incompatible with these search spaces since it is not possible to warmstart training with weights from a different architecture.

3.4.4 Tuning Modern LSTM Architectures

As a followup to the experiment in Section 3.4.3, we consider a search space for language modeling that is able to achieve near state-of-the-art performance. Our starting point was the work of Merity et al. [2018], which introduced a near state-of-the-art LSTM architecture with a more effective regularization scheme called DropConenct. We constructed a search space around their configuration and ran ASHA and PBT, each with 16 GPUS on one $\mathrm{p2.16xlarge}$ instance on AWS. Additional details for this experiment along with the hyperparameters that we considered are available in Section 3.6.5.

Figure 3.6 shows that while PBT performs better initially, ASHA soon catches up and finds a better final configuration; in fact, the min/max ranges for ASHA and PBT do not overlap at the end. ASHA is also faster than SHA and BOHB at finding a good configuration and converges to a better hyperparameter setting.

We then trained the best configuration found by ASHA for more epochs and reached validation and test perplexities of 60.2 and 58.1 respectively before fine-tuning and 58.7 and 56.3 after fine-tuning. For reference, Merity et al. [2018] reported validation and test perplexities respectively of 60.7 and 58.8 without fine-tuning and 60.0 and 57.3 with fine-tuning. This demonstrates the effectiveness of ASHA in the large-scale regime for modern hyperparameter optimization problems.

3.4.5 Tuning Large-Scale Language Models

In this experiment, we increase the number of workers to 500 to evaluate ASHA for massively parallel hyperparameter tuning. Our search space is constructed based off of the LSTMs considered

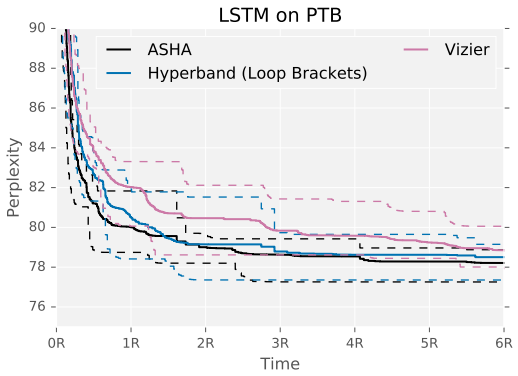


Figure 3.7: **Large-scale ASHA benchmark** requiring weeks to run with 500 workers. The x-axis is measured in units of average time to train a single configuration for R resource. The average across 5 trials is shown, with dashed lines indicating min/max ranges.

in Zaremba et al. [2014], with the largest model in our search space matching their large LSTM (see Section 3.6.6). For ASHA, we set $\eta = 4$, $r = R/64$, and $s = 0$; asynchronous Hyperband loops through brackets $s = 0, 1, 2, 3$. We compare to Vizier without the performance curve early-stopping rule [Golovin et al., 2017].[1]

The results in Figure 3.7 show that ASHA and asynchronous Hyperband found good configurations for this task in $1 \times time(R)$. Additionally, ASHA and asynchronous Hyperband are both about $3\times$ faster than Vizier at finding a configuration with test perplexity below 80, despite being much simpler and easier to implement. Furthermore, the best model found by ASHA achieved a test perplexity of 76.6, which is significantly better than 78.4 reported for the large LSTM in Zaremba et al. [2014]. We also note that asynchronous Hyperband initially lags behind ASHA, but eventually catches up at around $1.5 \times time(R)$.

Notably, we observe that certain hyperparameter configurations in this benchmark induce perplexities that are orders of magnitude larger than the average case perplexity. Model-based methods that make assumptions on the data distribution, such as Vizier, can degrade in performance without further care to adjust this signal. We attempted to alleviate this by capping perplexity scores at 1000 but this still significantly hampered the performance of Vizier. We view robustness to these types of scenarios as an additional benefit of ASHA and Hyperband.

3.5 Productionizing ASHA

While integrating ASHA in Determined AI's software platform to deliver production-quality hyperparameter tuning functionality, we encountered several fundamental design decisions that impacted usability, computational performance, and reproducibility. We next discuss each of these design decisions along with proposed systems optimizations for each decision.

[1] At the time of running the experiment, it was brought to our attention by the team maintaining the Vizier service that the early-stopping code contained a bug which negatively impacted its performance. Hence, we omit the results here.

3.5.1 Usability

Ease of use is one of the most important considerations in production; if an advanced method is too cumbersome to use, its benefits may never be realized. In the context of hyperparameter optimization, classical methods like random or grid search require only two intuitive inputs: *number of configurations* (n) and *training resources per configuration* (R). In contrast, as a byproduct of adaptivity, all of the modern methods we considered in this chapter have many internal hyperparameters. ASHA in particular has the following internal settings: elimination rate η, early-stopping rate s, and, in the case of asynchronous Hyperband, the brackets of ASHA to run. To facilitate use and increase adoption of ASHA, we simplify its user interface to require the same inputs as random search and grid search, exposing the internal hyperparameters of ASHA only to advanced users.

Selecting ASHA default settings. Our experiments in Section 3.4, as well as those in Chapter 2.4, show that aggressive early-stopping is effective across a variety of different hyperparameter tuning tasks. Hence, using both works as guidelines, we propose the following default settings for ASHA:

- Elimination rate: we set $\eta = 4$ so that the top 1/4 of configurations are promoted to the next rung.
- Maximum early-stopping rate: we set the maximum early-stopping rate for bracket s_0 to allow for a maximum of 5 rungs which indicates a minimum resource of $r = (1/4^4)R = R/256$. Then the minimum resource per configuration for a given bracket s is $r_s = r\eta^s$.
- Brackets to run: to increase robustness to misspecification of the early-stopping rate, we default to running the three most aggressively early-stopping brackets $s = 0, 1, 2$ of ASHA. We exclude the two least aggressive brackets (i.e. s_4 with $r_4 = R$ and s_3 with $r_3 = R/4$) to allow for higher speedups from early-stopping. We define this default set of brackets as the 'standard' set of early-stopping brackets, though we also expose the options for more conservative or more aggressive bracket sets.

Using n as ASHA's stopping criterion. Algorithm 3 does not specify a stopping criterion; instead, it relies on the user to stop once an implicit condition is met, e.g., number of configurations evaluated, compute time, or minimum performance achieved. In a production environment, we decided to use the number of configurations n as an explicit stopping criterion both to match the user interface for random and grid search, and to provide an intuitive connection to the underlying difficulty of the search space. In contrast, setting a maximum compute time or minimum performance threshold requires prior knowledge that may not be available.

From a technical perspective, n is allocated to the different brackets while maintaining the same total training resources across brackets. We do this by first calculating the average budget per configuration for a bracket (assuming no incorrect promotions), and then allocating configurations to brackets according to the inverse of this ratio. For concreteness, let B be the set of brackets we are considering, then the average resource for a given bracket s is $\bar{r}_s = {\#\text{ of Rungs}}/{\eta^{\lfloor \log_\eta R/r \rfloor - s}}$. For the default settings described above, this corresponds to $\bar{r}_0 = 5/256$, $\bar{r}_1 = 4/64$, and $\bar{r}_2 = 3/16$, and further translates to 70.5%, 22.1%, and 7.1% of the configurations being allocated to brackets s_0, s_1, and s_2, respectively.

Note that we still run each bracket asynchronously; the allocated number of configurations n_s

for a particular bracket s simply imposes a limit on the width of the bottom rung. In particular, upon reaching the limit n_s in the bottom rung, the number of pending configurations in the bottom rung is at most equal to the number of workers, k. Therefore, since blocking occurs once a bracket can no longer add configuration to the bottom rung and must wait for promotable configurations, for large-scale problems where $n_s \gg k$, limiting the width of rungs will not block promotions until the bracket is near completion. In contrast, synchronous SHA is susceptible to blocking from stragglers throughout the entire process, which can greatly reduce both the latency and throughput of configurations promoted to the top rung (e.g. Section 3.4.2 and Section 3.3.3).

3.5.2 Automatic Scaling of Parallel Training

The promotion schedule for ASHA geometrically increases the resource per configuration as we move up the rungs of a bracket. Hence, the average training time for configurations in higher rungs increases drastically for computation that scales linearly or super-linearly with the training resource, presenting an opportunity to speed up training by using multiple GPUs. We explore autoscaling of parallel training to exploit this opportunity when resources are available.

We determine the maximum degree of parallelism for autoscaling a training task using an efficiency criteria motivated by the observation that speedups from parallel training do not scale linearly with the number of GPUs [Goyal et al., 2017, Krizhevsky, 2014, Szegedy et al., 2015, You et al., 2017, You et al., 2017]. More specifically, we can use the Paleo framework, introduced by Qi et al. [2017], to estimate the cost of training neural networks in parallel given different specifications. Qi et al. [2017] demonstrated that the speedups from parallel training computed using Paleo are fairly accurate when compared to the actual observed speedups for a variety of models.

Figure 3.8 shows Paleo applied to Inception on ImageNet [Murray et al., 2016] to estimate the training time with different numbers of GPUs under strong scaling (i.e. fixed batch size with increasing parallelism), Butterfly AllReduce communication scheme, specified hardware settings (namely Tesla K80 GPU and 20G Ethernet), and a batch size of 1024.

The diminishing returns when using more GPUs to train a single model is evident in Figure 3.8. Additionally, there is a tradeoff between using resources to train a model faster to reduce latency versus evaluating more configurations to increase throughput. Using the predicted tradeoff curves generated using Paleo, we can automatically limit the number of GPUs per configuration to control efficiency relative to perfect linear scaling, e.g., if the desired level of efficiency is at least 75%, then we would limit the number of GPUs per configuration for Inception to at most 16 GPUs.

3.5.3 Resource Allocation

Whereas research clusters often require users to specify the number of workers requested and allocate workers on a first-in-first-out (FIFO) fashion, this scheduling mechanism is poorly suited for production settings for two main reasons. First, as we discuss below in the context of ASHA, machine learning workflows can have variable resource requirements over the lifetime of a job, and forcing users to specify static resource requirements can result in suboptimal cluster utilization. Second, FIFO scheduling can result in poor sharing of cluster resources among users, as a single large job could saturate the cluster and block all other user jobs.

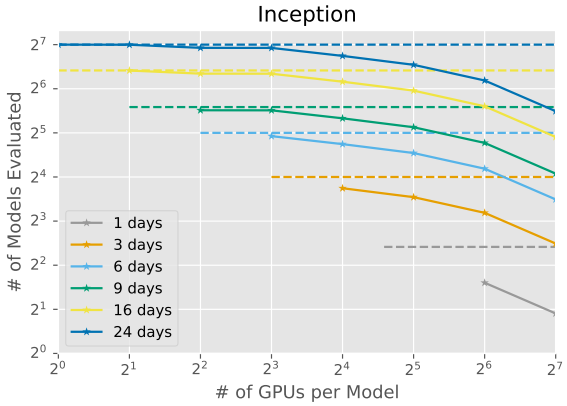


Figure 3.8: **Tradeoffs for parallel training of Imagenet using Inception V3.** Given that each configuration takes 24 days to train on a single Tesla K80 GPU, we chart the estimated number (according to the Paleo performance model) of configurations evaluated by 128 Tesla K80s as a function of the number of GPUs used to train each model for different time budgets. The dashed line for each color represents the number of models evaluated under perfect scaling, i.e. n GPUs train a single model n times as fast, and span the feasible range for number of GPUs per model in order to train within the allocated time budget. As expected, more GPUs per configuration are required for smaller time budgets and the total number of configurations evaluated decreases with number of GPUs per model due to decreasing marginal benefit.

We address these issues with a centralized fair-share scheduler that adaptively allocates resources over the lifetime of each job. Such a scheduler must both (i) determine the appropriate amount of parallelism for each individual job, and (ii) allocate computational resources across all user jobs. In the context of an ASHA workload, the scheduler automatically determines the maximum resource requirement at any given time based on the inputs to ASHA and the parallel scaling profile determined by Paleo. Then, the scheduler allocates cluster resources by considering the resource requirements of all jobs while maintaining fair allocation across users. We describe each of these components in more detail below.

Algorithm level resource allocation. Recall that we propose to use the number of configurations, n, as a stopping criteria for ASHA in production settings. Crucially, this design decision limits the maximum degree of parallelism for an ASHA job. If n is the number of desired configurations for a given ASHA bracket and κ the maximum allowable training parallelism, e.g., as determined by Paleo, then at initialization, the maximum parallelism for the bracket is $n\kappa$. We maintain a stack of training tasks $\mathbf{S}$ that is populated initially with all configurations for the bottom rung n. The top task in $\mathbf{S}$ is popped off whenever a worker requests a task and promotable configurations are added to the top of $\mathbf{S}$ when tasks complete. As ASHA progresses, the maximum parallelism is adaptively defined as $\kappa|\mathbf{S}|$. Hence, an adaptive worker allocation schedule that relies on $\kappa|\mathbf{S}|$ would improve cluster utilization relative to a static allocation scheme, without adversely impacting performance.

Cluster level resource allocation. Given the maximum degree of parallelism for any ASHA job, the scheduler then allocates resources uniformly across all jobs while respecting these maximum parallelism limits. We allow for an optional priority weighting factor so that certain jobs

can receive a larger ratio of the total computational resources. Resource allocation is performed using a water-filling scheme where any allocation above the maximum resource requirements for a job are distributed evenly to remaining jobs.

For concreteness, consider a scenario in which we have a cluster of 32 GPUs shared between a group of users. When a single user is running an ASHA job with 8 configurations in $\mathbf{S_1}$ and a maximum training parallelism of $\kappa_1 = 4$, the scheduler will allocate all 32 GPUs to this ASHA job. When another user submits an ASHA job with a maximum parallelism of $\kappa_2|\mathbf{S_2}| = 64$, the central scheduler will then allocate 16 GPUs to each user. This simple scenario demonstrate how our central scheduler allows jobs to benefit from maximum parallelism when the computing resources are available, while maintaining fair allocation across jobs in the presence of resource contention.

3.5.4 Reproducibility in Distributed Environments

Reproducibility is critical in production settings to instill trust during the model development process; foster collaboration and knowledge transfer across teams of users; and allow for fault tolerance and iterative refinement of models. However, ASHA introduces two primary reproducibility challenges, each of which we describe below.

Pausing and restarting configurations. There are many sources of randomness when training machine learning models; some source can be made deterministic by setting the random seed, while others related to GPU floating-point computations and CPU multi-threading are harder to avoid without performance ramifications. Hence, reproducibility when resuming promoted configurations requires carefully checkpointing all stateful objects pertaining to the model. At a minimum this includes the model weights, model optimizer state, random number generator states, and data generator state. We provide a checkpointing solution that facilitates reproducibility in the presence of stateful variables and seeded random generators. The availability of deterministic GPU floating-point computations is dependent on the deep learning framework, but we allow users to control for all other sources of randomness during training.

Asynchronous promotions. To allow for full reproducibility of ASHA, we track the sequence of all promotions made within a bracket. This sequence fixes the nondeterminism from asynchrony, allowing subsequent replay of the exact promotions as the original run. Consequently, we can reconstruct the full state of a bracket at any point in time, i.e. which configurations are on which rungs and which training tasks are in the stack.

Taken together, reproducible checkpoints and full bracket states allow us to seamlessly resume hyperparameter tuning jobs when crashes happen and allow users to request to evaluate more configurations if desired. For ASHA, refining hyperparameter selection by resuming an existing bracket is highly beneficial, since a wider rung gives better empirical estimates of the top $1/\eta$ configurations.

3.6 Experimental Details

In this section, we present the comparison to Fabolas in the sequential setting and provide additional details for the empirical results shown in Section 3.4.

3.6.1 Comparison with Fabolas in Sequential Setting

Klein et al. [2017a] showed that Fabolas can be over an order of magnitude faster than prior Bayesian optimization methods. Additionally, the empirical studies presented by Klein et al. [2017a] suggest that Fabolas is faster than Hyperband at finding a good configuration. We conducted our own experiments to compare Fabolas with Hyperband on the following tasks:

1. Tuning an SVM using the same search space as that used by Klein et al. [2017a].
2. Tuning a convolutional neural network (CNN) with the same search space as that considered in Chapter 2.4.1 on CIFAR-10 [Krizhevsky, 2009].
3. Tuning a CNN on SVHN [Netzer et al., 2011] with varying number of layers, batch size, and number of filters (see Section 3.6.3 for more details).

In the case of the SVM task, the allocated resource is number of training datapoints, while for the CNN tasks, the allocated resource is the number of training iterations.

We note that Fabolas was specifically designed for data points as the resource, and hence, is not directly applicable to tasks (2) and (3). However, freeze-thaw Bayesian optimization [Swersky et al., 2014], which was specifically designed for models that use iterations as the resource, is known to perform poorly on deep learning tasks [Domhan et al., 2015]. Hence, we believe Fabolas to be a reasonable competitor for tasks (2) and (3) as well, despite the aforementioned shortcoming.

We use the same evaluation framework as Klein et al. [2017a], where the best configuration, also known as the *incumbent*, is recorded through time and the test error is calculated in an offline validation step. Following Klein et al. [2017a], the incumbent for Hyperband is taken to be the configuration with the lowest validation loss and the incumbent for Fabolas is the configuration with the lowest predicted validation loss on the full dataset. Moreover, for these experiments, we set $\eta = 4$ for Hyperband.

Notably, when tracking the best performing configuration for Hyperband, we consider two approaches. We first consider the approach proposed in Chapter 2.3 used by Klein et al. [2017a] in their evaluation of Hyperband. In this variant, which we refer to as "Hyperband (by bracket)," the incumbent is recorded *after the completion of each SHA bracket*. We also consider a second approach where we record the incumbent *after the completion of each rung* of SHA to make use of intermediate validation losses, similar to what we propose for ASHA (see discussion in Section 3.3.4 for details). We will refer to Hyperband using this accounting scheme as "Hyperband (by rung)." Interestingly, by leveraging these intermediate losses, we observe that Hyperband actually outperforms Fabolas.

In Figure 3.9, we show the performance of Hyperband, Fabolas, and random search. Our results show that Hyperband (by rung) is competitive with Fabolas at finding a good configuration and will often find a better configuration than Fabolas with less variance. Note that Hyperband loops through the brackets of SHA, ordered by decreasing early-stopping rate; the first bracket finishes when the test error for Hyperband (by bracket) drops. Hence, most of the progress made by Hyperband comes from the bracket with the most aggressive early-stopping rate, i.e. bracket 0.

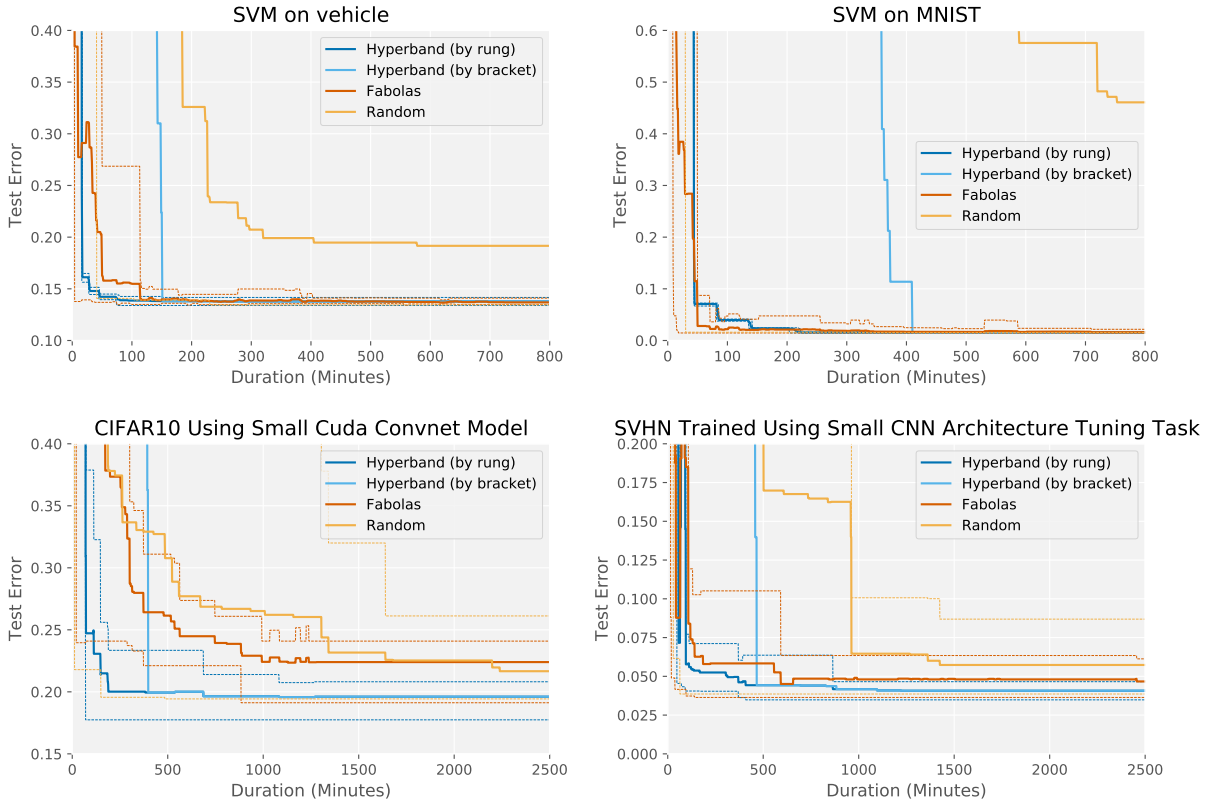


Figure 3.9: **Sequential Experiments** (1 worker) with Hyperband running synchronous SHA. Hyperband (by rung) records the incumbent after the completion of a SHA rung, while Hyperband (by bracket) records the incumbent after the completion of an entire SHA bracket. The average test error across 10 trials of each hyperparameter optimization method is shown in each plot. Dashed lines represent min and max ranges for each tuning method.

3.6.2 Experiments in Section 3.4.1 and Section 3.4.2

We use the usual train/validation/test splits for CIFAR-10, evaluate configurations on the validation set to inform algorithm decisions, and report test error. These experiments were conducted using g2.2xlarge instances on Amazon AWS.

For both benchmark tasks, we run SHA and BOHB with $n = 256$, $\eta = 4$, $s = 0$, and set $r = R/256$, where $R = 30000$ iterations of stochastic gradient descent. Hyperband loops through 5 brackets of SHA, moving from bracket $s = 0, r = R/256$ to bracket $s = 4, r = R$. We run ASHA and asynchronous Hyperband with the same settings as the synchronous versions. We run PBT with a population size of 25, which is between the recommended 20–40 [Jaderberg et al., 2017]. Furthermore, to help PBT evolve from a good set of configurations, we randomly sample configurations until at least half of the population performs above random guessing.

We implement PBT with truncation selection for the exploit phase, where the bottom 20% of configurations are replaced with a uniformly sampled configuration from the top 20% (both weights and hyperparameters are copied over). Then, the inherited hyperparameters pass through an exploration phase where $3/4$ of the time they are either perturbed by a factor of 1.2 or 0.8 (discrete hyperparameters are perturbed to two adjacent choices), and $1/4$ of the time they are randomly resampled. Configurations are considered for exploitation/exploration every 1000 iterations, for a total of 30 rounds of adaptation. For the experiments in Section 3.4.2, to maintain 100% worker efficiently for PBT while enforcing that all configurations are trained for within 2000 iterations of each other, we spawn new populations of 25 whenever a job is not available from existing populations.

Hyperparameter	Type	Values
batch size	choice	$\{2^6, 2^7, 2^8, 2^9\}$
# of layers	choice	$\{2, 3, 4\}$
# of filters	choice	$\{16, 32, 48, 64\}$
weight init std 1	continuous	$\log\,[10^{-4}, 10^{-1}]$
weight init std 2	continuous	$\log\,[10^{-3}, 1]$
weight init std 3	continuous	$\log\,[10^{-3}, 1]$
l_2 penalty 1	continuous	$\log\,[10^{-5}, 1]$
l_2 penalty 2	continuous	$\log\,[10^{-5}, 1]$
l_2 penalty 3	continuous	$\log\,[10^{-3}, 10^{2}]$
learning rate	continuous	$\log\,[10^{-5}, 10^{1}]$

Table 3.1: Hyperparameters for small CNN architecture tuning task.

Vanilla PBT is not compatible with hyperparameters that change the architecture of the neural network, since inherited weights are no longer valid once those hyperparameters are perturbed. To adapt PBT for the architecture tuning task, we fix hyperparameters that affect the architecture in the explore stage. Additionally, we restrict configurations to be trained within 2000 iterations of each other so a fair comparison is made to select configurations to exploit. If we do not impose this restriction, PBT will be biased against configurations that take longer to train, since it will be comparing these configurations with those that have been trained for more iterations.

3.6.3 Experimental Setup for the Small CNN Architecture Tuning Task

This benchmark tunes a multiple layer CNN network with the hyperparameters shown in Table 3.1. This search space was used for the small architecture task on SVHN (Section 3.6.1) and CIFAR-10 (Section 3.4.2). The # of layers hyperparameter indicate the number of convolutional layers before two fully connected layers. The # of filters indicates the # of filters in the CNN layers with the last CNN layer having $2 \times$ # filters. Weights are initialized randomly from a Gaussian distribution with the indicated standard deviation. There are three sets of weight init and l_2 penalty hyperparameters; weight init 1 and l_2 penalty 1 apply to the convolutional layers, weight init 2 and l_2 penalty 2 to the first fully connected layer, and weight init 3 and l_2 penalty 3 to the last fully connected layer. Finally, the learning rate hyperparameter controls the initial learning rate for SGD. All models use a fixed learning rate schedule with the learning rate decreasing by a factor of 10 twice in equally spaced intervals over the training window. This benchmark is run on the SVHN dataset [Netzer et al., 2011] following Sermanet et al. [2012] to create the train, validation, and test splits.

3.6.4 Experimental Setup for Neural Architecture Search Benchmarks

For NAS benchmarks evaluated in Section 3.4.3, we used the same search space as that considered by Liu et al. [2019] for designing CNN and RNN cells. We sample architectures from the associated search space randomly and train them using the same hyperparameter settings as that used by Liu et al. [2019] in the evaluation stage. We refer the reader to the following code repository for more details: https://github.com/liamcli/darts_asha.

3.6.5 Tuning Modern LSTM Architectures

For the experiments in Section 3.4.4, we used $\eta = 4$, $r = 1$ epoch, $R = 256$ epochs, and $s = 0$ for ASHA, SHA, and BOHB. For PBT, we use a population size to 20, a maximum resource of 256 epochs, and perform explore/exploit every 8 epochs using the same settings as the previous experiments. The hyperparameters that we considered along with their associated ranges are shown in Table 3.2.

Hyperparameter	Type	Values
learning rate	continuous	$\log\,[10, 100]$
dropout (rnn)	continuous	$[0.15, 0.35]$
dropout (input)	continuous	$[0.3, 0.5]$
dropout (embedding)	continuous	$[0.05, 0.2]$
dropout (output)	continuous	$[0.3, 0.5]$
dropout (dropconnect)	continuous	$[0.4, 0.6]$
weight decay	continuous	$\log\,[0.5e-6, 2e-6]$
batch size	discrete	$[15, 20, 25]$
time steps	discrete	$[65, 70, 75]$

Table 3.2: Hyperparameters for 16 GPU near state-of-the-art LSTM task.

3.6.6 Experimental Setup for Large-Scale Benchmarks

Hyperparameter	Type	Values
batch size	discrete	$[10, 80]$
# of time steps	discrete	$[10, 80]$
# of hidden nodes	discrete	$[200, 1500]$
learning rate	continuous	$\log [0.01, 100.]$
decay rate	continuous	$[0.01, 0.99]$
decay epochs	discrete	$[1, 10]$
clip gradients	continuous	$[1, 10]$
dropout probability	continuous	$[0.1, 1.]$
weight init range	continuous	$\log [0.001, 1]$

Table 3.3: Hyperparameters for PTB LSTM task.

The hyperparameters for the LSTM tuning task comparing ASHA to Vizier on the Penn Tree Bank (PTB) dataset presented in Section 3.4.5 is shown in Table 3.3. Note that all hyperparameters are tuned on a linear scale and sampled uniform over the specified range. The inputs to the LSTM layer are embeddings of the words in a sequence. The number of hidden nodes hyperparameter refers to the number of nodes in the LSTM. The learning rate is decayed by the decay rate after each interval of decay steps. Finally, the weight initialization range indicates the upper bound of the uniform distribution used to initialize all weights. The other hyperparameters have their standard interpretations for neural networks. The default training (929k words) and test (82k words) splits for PTB are used for training and evaluation [Marcus et al., 1993]. We define resources as the number of training records, which translates into the number of training iterations after accounting for certain hyperparameters.

Chapter 4

Reuse in Pipeline-Aware Hyperparameter Optimization

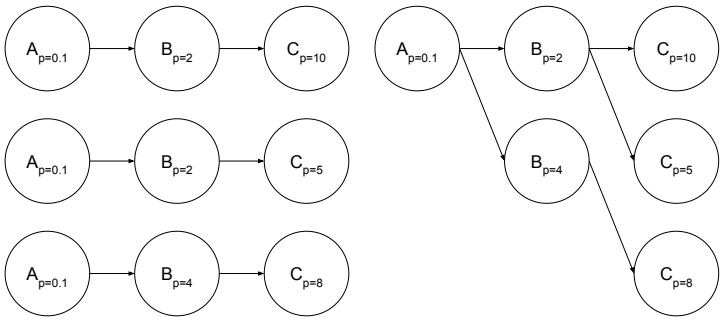
The hyperparameter tuning experiments in Chapter 2.4.2.1 considered a search space for machine learning pipelines that included methods for data preprocessing, featurization, and model selection. Such multi-stage machine learning pipeline search spaces will be the focus of this chapter. In particular, we explore methods to speed up hyperparameter optimization for pipelines by exploiting the structure inherent in the search space to remove redundant computation. This work extends the methodology by Sparks [2016, Chapter 6] with strategies to increase potential for reuse in pipeline search spaces.

4.1 Introduction

Modern machine learning workflows combine multiple stages of data-preprocessing, feature extraction, and supervised and unsupervised learning [Feurer et al., 2015a, Sánchez et al., 2013]. The methods considered in *each* of these stages typically have their own associated hyperparameters, which must be tuned to achieve high predictive accuracy. Although tools have been designed to speed up the development and execution of such complex pipelines [Meng et al., 2016, Pedregosa et al., 2011, Sparks et al., 2017], tuning hyperparameters at various pipeline stages remains a computationally burdensome task.

Some tuning methods are sequential in nature and recommend hyperparameter configurations one at a time for evaluation [e.g. Bergstra et al., 2011, Hutter et al., 2011, Snoek et al., 2012], while other batch mode algorithms like HYPERBAND (Chapter 2) and ASHA (Chapter 3) among others [e.g. Krueger et al., 2015, Sabharwal et al., 2016] evaluate multiple configurations simultaneously. In either case, holistically analyzing the functional structure of the resulting collection of evaluated configurations introduces opportunities to exploit computational reuse in the traditional black-box optimization problem.

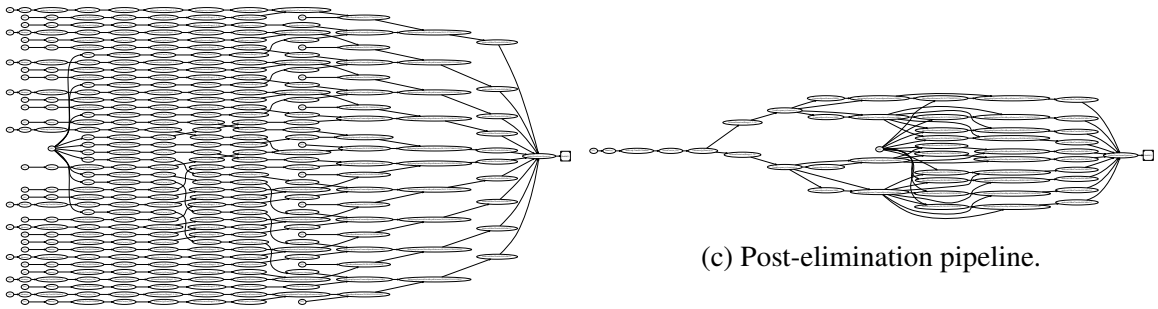
Given a batch of candidate pipeline configurations, it is natural to model the dependencies between pipelines via a directed acyclic graph (DAG). We can eliminate redundant computation in the original DAG by collapsing shared prefixes so that pipeline configurations that depend on the same intermediate computation will share parents, resulting in a *merged DAG*. We define



(a) (b)

Figure 4.1: Three pipelines with overlapping hyperparameter configurations for operations A, B, and C. (a) Three independent pipelines. (b) Merged DAG after eliminating shared prefixes.

(a) Example pipeline for text classification.



(b) Pre-elimination pipeline.

(c) Post-elimination pipeline.

Figure 4.2: Ten sampled hyperparameter configurations for the pipeline in 4.2a before and after common prefix elimination.

pipeline-aware hyperparameter tuning as the task of leveraging this merged DAG to evaluate hyperparameter configurations, with the goal being to exploit reuse for improved efficiency.

Consider the example of the three toy pipelines in Figure 4.1(a). All of them have the identical configuration for operator A, and the first two have the same configuration for operator B, leading to the merged DAG illustrated in Figure 4.1(b). Whereas treating each configuration independently requires computing node $A_{p=0.1}$ three times and node $B_{p=2}$ twice, operating on the merged DAG allows us to compute each node once. Figure 4.2 illustrates the potential for reuse via merged DAGs in a more realistic problem.

In this chapter, we tackle the problem of pipeline-aware hyperparameter tuning, by optimizing both the *design* and *evaluation* of merged DAGs to maximally exploit reuse. In particular, we identify the following three core challenges, and our associated contributions addressing them.

Designing Reusable DAGs. Hyperparameter configurations are typically randomly sampled, often from continuous search spaces. The resulting merged DAGs consequently are extremely limited in terms of their shared prefixes, which fundamentally restricts the amount of possible reuse. To overcome this limitation, we introduce a novel configuration selection method that encourages shared prefixes. Our proposed approach *gridded random search* limits the branching factor of continuous hyperparameters, while performing more exploration than standard grid search.

Designing Balanced DAGs. Modern learning models are computationally expensive to train. Therefore, the leaf nodes of the DAG can be more expensive to evaluate than intermediate nodes, limiting the impact of reuse. To balance backloaded computation, we propose using hyperparameter tuning strategies that exploit early-stopping and partial training. These approaches can be interpreted as converting a single, expensive training node into a sequence of cheaper training nodes, resulting in a more balanced DAG. In particular, we show how the successive halving algorithm presented in Chapter 2 can be used to significantly reduce total training time.

Exploiting Reuse via Caching. Exploiting reuse requires an efficient cache strategy suitable for machine learning pipelines, which can have highly variable computational profiles at different nodes. Although the general problem of caching is well studied in computer science [McGeoch and Sleator, 1991, Sleator and Tarjan, 1985], there is limited work on caching machine learning workloads, where the cost of computing an item can depend on the state of the cache. Surprisingly, our results demonstrate that the least-recently used (LRU) cache eviction heuristic used in many frameworks (e.g., scikit-learn and MLlib) is not well suited for this setting, and we thus propose using WRECIPROCAL [Gunda et al., 2010], a caching strategy that accounts for both the size and the computational cost of an operation.

In Section 4.5, we apply these three complementary approaches to real pipelines and show that pipeline-aware hyperparameter tuning can offer more than an *order-of-magnitude* speedup compared with the standard practice of evaluating each pipeline independently.

4.2 Related Work

Hyperparameter tuning and caching are both areas with extensive prior work. We highlight pertinent results in both areas in the context of reuse.

Hyperparameter Tuning Methods. The classical hyperparameter tuning method grid search is naturally amenable to reuse via prefix sharing. However, random search performs better in practice than grid search [Bergstra and Bengio, 2012]. Our gridded random search approach combines the best of both of these classical methods (reuse from grid search and improved accuracy from random search).

Early-stopping methods like successive halving and others mentioned in Chapter 2 and Chapter 3 reduce average training time per configuration by over an order-of-magnitude. We leverage this observation by deploying these early stopping methods in a pipeline-aware context to generate balanced DAGs amenable for reuse.

Caching with Perfect Information. One straightforward way of reusing intermediate computation is to save results to disk. However, the associated I/O costs are large, thus motivating our exploration of efficient caching in memory as the primary method of realizing reuse in

pipeline-aware hyperparameter tuning.

Hence, we study the benefits from reuse of three online strategies: (1) LRU: Least recently used items are evicted first. LRU is *k-competitive* with Belady's algorithm [Sleator and Tarjan, 1985] for the classic paging problem with uniform size and cost.[1] (2) RECIPROCAL: Items are evicted with probability inversely proportional to the cost. RECIPROCAL is k-competitive with the optimal caching strategy for the weighted paging problem [Motwani and Raghavan, 1995]. (3) WRECIPROCAL: Items are evicted with probability inversely proportional to cost and directly proportional to size. WRECIPROCAL is a weighted variant of RECIPROCAL that has been shown to work well for datacenter management [Gunda et al., 2010].

For the classic paging problem, randomized online caching strategies outperform deterministic ones and are more robust [Fiat et al., 1991]. However, we include LRU in our comparison since it is widely used on account of its simplicity (including in some of the machine learning methods described below). Additionally, there is a large body of work on efficient randomized online caching policies for the weighted paging problem and generalized paging problem [Bansal et al., 2008]. Studying the empirical effectiveness of these more recent and complex caching methods for tuning machine learning pipelines is a direction for future work.

Caching in Machine Learning. Popular frameworks scikit-learn and MLlib offer caching via the least-recently used heuristic (LRU). Some hyperparameter tuning algorithms like TPOT [Gijsbers et al., 2018] and FLASH [Zhang et al., 2016] also offer LRU caching of intermediate computation. However, as we show in Section 4.4.3, LRU is unsuitable for machine learning pipelines.

4.3 DAG Formulation and Analysis

We first formalize the DAG representation of the computation associated with a set of candidate pipeline configurations. Nodes within the DAG represent the computation associated with each stage of a single pipeline with specific hyperparameter settings and edges within the DAG represent the flow of the transformed data to another stage of the pipeline. For a given node, the cost refers to the time needed to compute the output of the node, while the size refers to the memory required to store the output. Finally, to construct the merged DAG, we merge two nodes if (1) they share the same ancestors, indicating they operate on the same data; and (2) they have the same hyperparameter settings, indicating they represent the same computation.

We can use our DAG representation to gain intuition for potential speedups from reuse. For simplicity, we assume the cache is unbounded and we can store as many intermediate results as desired. Additionally, let P be the set of all pipelines considered in the DAG. Next, for a given pipeline p, let V_p represent the nodes associated with each operator within the pipeline and $c(v)$ be the cost in execution time for a given node $v \in V_p$. Then the total time required to evaluate the DAG without reuse is $TP(P) = \sum_{p \in P} \sum_{v \in V_p} c(v)$.

Let $P_{\text{merged}} = \cup_{p \in P} V_p$ represent the nodes in the merged DAG, where $\cup$ is the merge operator following the rules described above. Then the runtime for the merged DAG is $TP(P_{\text{merged}}) = \sum_{v \in V_{P_{\text{merged}}}} c(v)$, and the ratio, $TP(P)/TP(P_{\text{merged}})$ is the speedup achieved from merging two pipelines.

[1] An algorithm is k-competitive if its worst case performance is no more than a factor of k worse than the optimal algorithm, where k denotes the size of the cache.

For the purposes of illustration, if one holds $c(v)$ constant and assumes that each pipeline has the same length, $|V|$, it is simple to see that $TP(P) = |P||V|c(v)$. For the merged pipeline, if all the pipelines in the set are disjoint, then $TP(P) = TP(P_{\text{merged}})$ and the speedup is $1\times$, indicating no benefit from reuse. On the other hand, for the maximally redundant pipeline, where all the pipelines differ only in the *last* node, the total execution time is $TP(P_{\text{merged}}) = (|V|+|P|-1)c(v)$. Hence, the maximum speedup ratio in this setting is ${}^{|V||P|}/_{(|V|+|P|-1)}$. In the limit, when all pipelines are long, speedups tend to $|P|$, while if the number of trajectories dominate, speedups tend to $|V|$.

Next, if we relax the assumption that $c(v)$ is fixed, and assume instead that merged nodes are expensive relative to the *last* stage of the pipeline, the speedups will approach $|P|$, since a pipeline with more expensive intermediate nodes is similar to a longer pipeline with fixed cost. Alternatively, if we assume that the last stage of the pipeline is expensive relative to intermediate stages, we can interpret this as a long pipeline where many stages are disjoint, and the speedups will approach $1\times$. Hence, to maximize speedups from reuse, we need to optimize the design of DAGs to promote prefix sharing and a balanced computational profile with limited time spent on training leaf nodes.

4.4 Increasing Potential for Reuse in ML Pipelines

In this section, we delve into our proposed solutions to each of the three core challenges limiting reuse in machine learning pipelines. Specifically, we introduce gridded random search in Section 4.4.1 to increase prefix sharing; demonstrate how early-stopping can balance training-heavy DAGs by decreasing average training time in Section 4.4.2; and identify suitable caching strategies for machine learning workloads in Section 4.4.3.

4.4.1 Gridded Random Search versus Random Search

As discussed in Section 4.2, grid search discretizes each hyperparameter dimension, while random search samples randomly in each dimension. Gridded random is a hybrid of the two approaches that encourages prefix sharing by controlling the branching factor from each node in a given stage of the pipeline, promoting a narrower tree-like structure. We show a visual comparison between grid search and gridded random search in Figure 4.3. The key difference from grid search is that the children of different parent nodes from the same level are not required to have the same hyperparameter settings. In effect each node performs a separate instance of gridded random search for the remaining downstream hyperparameters. Gridded random search introduces a tradeoff between the amount of potential reuse and coverage of the search space. Generally, the branching factor should be higher for nodes that have more hyperparameters in order to guarantee sufficient coverage. As we will see next, gridded random search achieves performance comparable to random search, despite the former's limited coverage of the search space relative to the latter.

We evaluate the performance of gridded random search compared to standard random search on 20 OpenML classification datasets. The search space we consider includes the following pipeline stages: (1) Preprocessor: choice of none, standardize, normalize, or min/max scaler; (2) Featurizer: choice of PCA, select features with top percentile variance, or ICA; (3) Classifier:

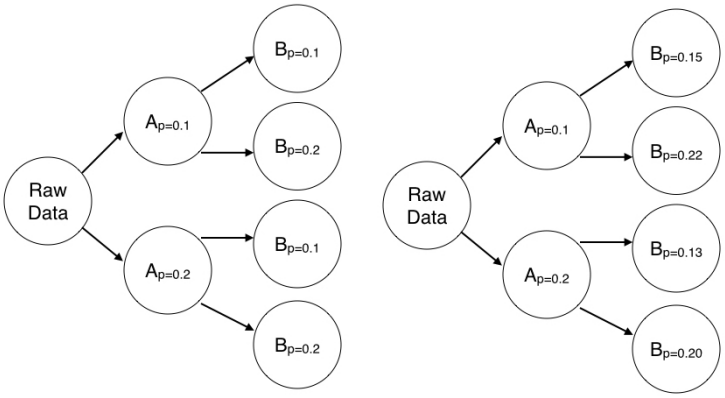


Figure 4.3: Comparison of grid search versus gridded random search. Grid search considers two different values for each hyperparameter dimension. In contrast, gridded random considers two hyperparameter values for operator A and four different values for operator B, while still maintaining a branching factor of two for the DAG. Note that random search would consider four different values for A and four different values for B but allow no potential for reuse.

choice of linear classifier using stochastic gradient descent, random forests, or logistic regression. For each dataset, all four preprocessors are considered, but a single featurizer and a single classifier are randomly selected. For gridded random search, the branching factor in the first stage is 4, one for each preprocessor type; the branching factor is 5 per node for the selected featurizer; and then 5 per featurizer node for the selected classifier. This results in 100 total pipelines for gridded random search.

For each dataset, we evaluated a set of 100 pipelines sampled using either random or gridded random and recorded the best configuration for each sampling method. To reduce the effect of randomness, we averaged results across 10 trials of 100 pipelines for each dataset. Our results in Table 4.1 show that on average gridded random search converges to similar objective values as standard random search on a variety of datasets. With the exception of one task, the differences in performance between random and gridded random are all below 1% accuracy. While by no means exhaustive, these results demonstrate the viability of using gridded random search to increase reuse.

4.4.2 Balancing DAGs via Successive Halving

The benefit from reuse is limited for pipelines with long training times due to Amdahl's Law, i.e., the relative amount of redundant computation is only a small fraction of the total time. For backloaded pipelines, we can increase the potential for reuse by reducing the average training time per configuration, e.g., by employing iterative early-stopping hyperparameter tuning algorithms when evaluating a batch of configurations. We next demonstrate how early-stopping via successive halving (Algorithm 2) can be used to generate more balanced DAGs when training time dominates.

As an example of how SHA can be combined with pipeline-aware hyperparameter tuning, consider the DAG in Figure 4.4 where the time needed for the preprocessing steps in the pipeline is the same as the training time. Assume we run SHA with $n = 16$ pipelines, $\eta = 4$, for a maximum

Dataset ID	Featurizer	Classifier	Mean	20%	80%
OpenML 182	SelectPercentile	LogisticReg	0.1%	0.0%	0.0%
OpenML 300	SelectPercentile	SVC	0.6%	0.6%	0.6%
OpenML 554	PCA	LogisticReg	0.0%	0.1%	0.0%
OpenML 722	PCA	RandForest	-0.1%	-0.1%	-0.1%
OpenML 734	SelectPercentile	LogisticReg	0.0%	0.1%	0.0%
OpenML 752	SelectPercentile	LogisticReg	-0.0%	0.0%	-0.0%
OpenML 761	SelectPercentile	LogisticReg	0.6%	2.1%	0.0%
OpenML 833	SelectPercentile	RandForest	-0.0%	0.1%	-0.1%
OpenML 1116	PCA	RandForest	-0.2%	-0.2%	-0.3%
OpenML 1475	SelectPercentile	SVC	-0.8%	-0.9%	-0.9%
OpenML 1476	PCA	SVC	0.3%	-0.4%	0.0%
OpenML 1477	PCA	SVC	0.4%	-0.8%	0.1%
OpenML 1486	PCA	SVC	0.2%	0.3%	0.3%
OpenML 1496	FastICA	LogisticReg	0.0%	0.1%	-0.0%
OpenML 1497	PCA	RandForest	-0.3%	-0.1%	0.2%
OpenML 1507	SelectPercentile	SVC	0.1%	0.2%	0.0%
OpenML 4538	SelectPercentile	RandForest	-0.0%	-0.4%	0.3%
OpenML 23517	SelectPercentile	RandForest	-0.0%	0.0%	-0.1%
OpenML 40499	PCA	LogisticReg	2.9%	6.1%	1.5%
OpenML 40996	SelectPercentile	RandForest	-0.0%	-0.1%	-0.1%

Table 4.1: For each OpenML dataset, we randomly sample a featurizer and classifier to construct the search space. The right three columns show the difference in aggregate statistics between random search and gridded random search. Positive values indicates the performance of random is better than that of gridded random. For each task, statistics for the best hyperparameter setting found over 100 pipelines are computed over 10 independent runs.

resource of R. SHA begins with the full DAG containing 16 pipelines and traverses the DAG, while training leaf nodes (i.e., a classifier) with the initial minimum resource $r = R/16$ (blue sub-DAG). In the next rung, it increases the resource per leaf node to $R/4$ and traverse the pruned DAG with 4 remaining configurations (yellow sub-DAG). Finally, in the top rung a single pipeline remains, and it is trained for the maximum resource R (red sub-DAG). Hence in total, SHA use $3R$ resources for training compared to the $16R$ that would be needed without early-stopping. If training times scaled linearly, then total training time is reduced by over $5\times$ with SH. Visually, this reduction in training time is evident in Figure 4.4, where the computational profile for the sub-DAGs is much more frontloaded.

In general, for a maximum resource of R per configuration, a total budget of nR is needed to evaluate n configurations without early-stopping. In contrast, SHA with k rungs requires a budget equivalent to $(nR\eta^{-k})k$, where $nR\eta^{-k}$ indicates the total resource allocated in each rung. Hence, SHA can be used to balance backloaded computation by reducing the total training time by a factor of η^k/k relative to no early-stopping. Note that SHA is normally run with larger DAGs and more aggressive early-stopping rates, which as we show in Section 4.5.2, can drastically increase

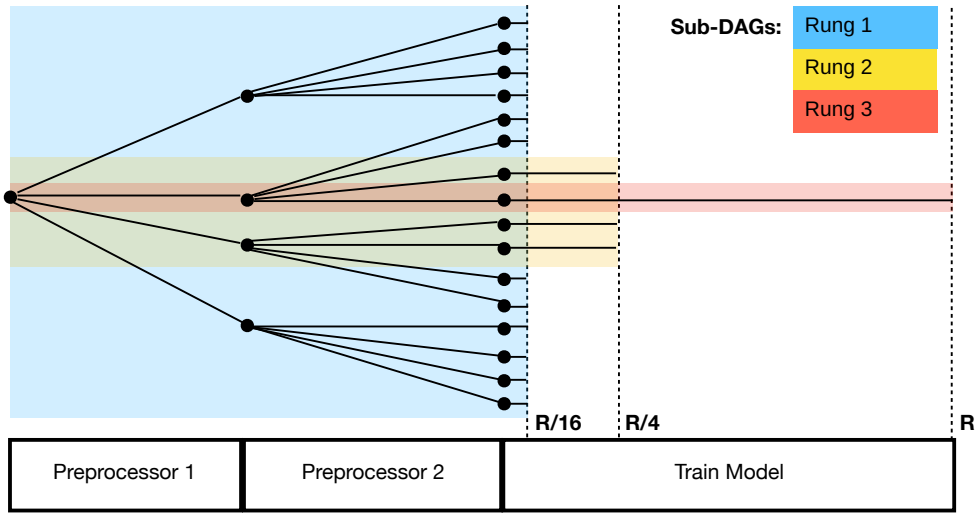


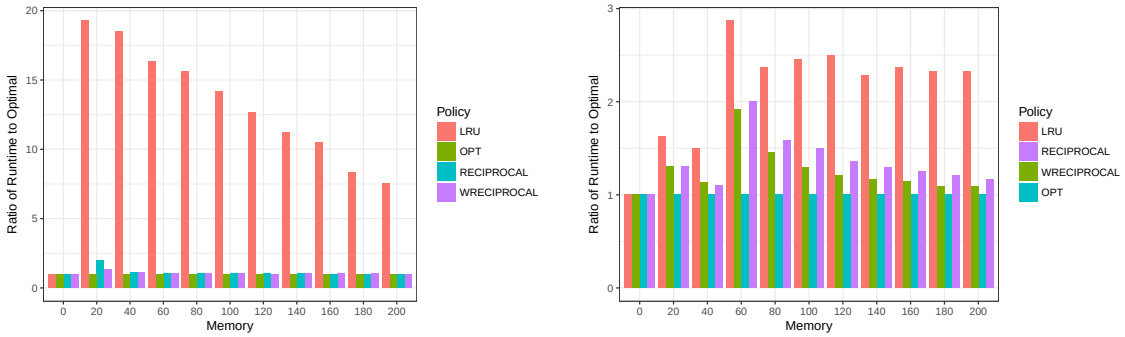
Figure 4.4: In this DAG, all pipelines start with the raw data in the root node and proceed to subsequent steps of the pipeline with a branching factor of 4 for a total of 16 pipelines. The lengths of the edges correspond to the time needed to compute the subsequent node. Hence, for the depicted pipelines, the time needed to train a model on R resource is equal to the time needed to compute the two preprocessing steps. Therefore, without early-stopping, training time accounts for over half of the total time needed to execute the entire DAG. Switching over to Successive Halving (SHA) with elimination rate $\eta = 4$, the shaded blocks indicate the resulting pruned DAGs after each rung. The lengths of the edges in the 'Train Model' phase correspond to resources allocated to those configurations by SHA: 16 configurations are trained for $R/16$ in the first rung, 4 configurations for $R/4$ in the second rung, and one configuration for R in the top rung. Hence, the total training time required for SHA is $3R$, which is over $5\times$ smaller than the $16R$ needed without early-stopping, leading to a DAG with more frontloaded computation. Note that SHA is normally run with larger DAGs (more pipelines) and more aggressive early-stopping rates, which, as we show in Section 4.5.2, further increase speedups from reuse.

the speedups from reuse on a real world pipeline optimization task.

4.4.3 Evaluating Caching Strategies

We compare the performance of online caching strategies introduced in Section 4.2 to the precomputed optimal policy (OPT) on synthetic DAGs designed to look like common hyperparameter tuning pipelines in terms of size and computational cost. We consider DAGs of k-ary trees with varying depth d and branching factor k; this is akin to generated DAGs using gridded random search on pipelines of length d for which there are k choices of hyperparameter values at each node.

First we consider a scenario where the memory size of each result is fixed at 10 and the root node is assigned a computational cost of 100 while all other nodes have a cost of 1. Figure 4.5a shows the results of this experiment. LRU performs poorly for this configuration precisely because



(a) All nodes have size 10; root node has cost 100 and all other nodes have cost 1.

(b) For each node, size is randomized to be 10 or 50 and cost is randomized to be 1 or 100.

Figure 4.5: A comparison of cache management policies on a 3-ary tree with depth 3.

the root node in the tree uses 100 units of computation and LRU does not account for this. The other strategies converge to the optimal strategy relatively quickly.

Next, we allow for variable sizes and costs. Each node is randomly given a size of either 10 or 50 units with equal probability and a corresponding cost of 1 or 100. Figure 4.5b shows the competitiveness of each strategy vs. the optimal strategy. In this setting, LRU is again the least effective policy while WRECIPROCAL is closest to optimal for most cache sizes. These results on synthetic DAGs suggest that LRU, the default caching strategy in scikit-learn and MLlib, is poorly suited for machine learning pipelines. Our experiments in the next section on real-world tasks reinforce this conclusion.

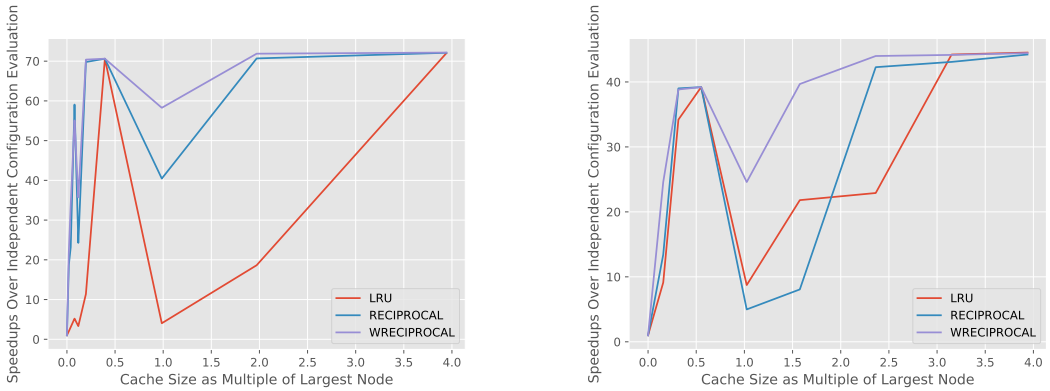
4.5 Experiments

In this section we evaluate the speedups from reuse on real-world hyperparameter tuning tasks gained by integrating our proposed techniques. We consider tuning pipelines for the following three datasets: 20 Newsgroups,[2] Amazon Reviews [McAuley et al., 2015], and the TIMIT [Huang et al., 2014] speech dataset. Our results are simulated on DAGs generated using profiles collected during actual runs of each pipeline. For each task, we examine potential savings from reuse for 100 pipelines trained simultaneously.

In Section 4.5.1, we examine the speedups from the caching component of pipeline-aware tuning on a text classification pipeline for the Newsgroup and Amazon Reviews datasets. The DAGs associated with both datasets are naturally amenable to reuse due to discrete hyperparameters and front-loaded computation. In Section 4.5.2, we then leverage our entire three-pronged approach to pipeline-aware hyperparameter tuning on a speech classification task that at first glance is not amenable to reuse.

We quantify the speedups by comparing the execution time required for our pipeline-aware approach to the execution time when evaluating the pipelines independently. We focus on the computational efficiency of our approach as opposed to accuracy, since we have already explored

[2] http://qwone.com/~jason/20Newsgroups/



(a) 20 Newsgroups

(b) Amazon Reviews

Figure 4.6: Effect of reuse on two real-world text processing hyperparameter tuning tasks. Speedups over independent configuration evaluation when evaluating a DAG with 100 pipelines are shown for each cache policy given a particular cache size.

the impact of gridded random search.

4.5.1 Tuning Text Classification Pipelines

We tune the pipeline in Figure 4.2(a) for the 20 Newsgroups and the Amazon Reviews datasets. As shown in Figure 4.2a, RECIPROCAL and WRECIPROCAL offer $70\times$ speedup over independent configuration evaluation for the 20 Newsgroups dataset. Similarly, for the Amazon Reviews dataset, all the caching strategies offer more than $40\times$ speedup over independent configuration evaluation. In these instances, caching offers significant speedups due to the front-loaded nature of the pipelines. By caching, we save on several shared steps: tokenization, NGrams generation, and feature selection/generation. Additionally, compared to the preprocessing and featurization steps, training is relative inexpensive.

The frontloaded computation also explains why LRU underperforms on small memory sizes; by biasing older nodes, earlier and more expensive pipeline stages are more likely to be evicted. Another drawback of LRU is that it does not take into account the cost of computing each node and it will always try to cache the most recent node. Hence, once the size of the cache is large enough to cache the output of the largest node, LRU adds the node to cache and evicts all previous caches, while RECIPROCAL and WRECIPROCAL are less likely to cache the node. This explains the severe drop in performance for LRU once cache size exceeded 1. These results also show WRECIPROCAL to be more robust than RECIPROCAL because it is less likely to cache large nodes with relatively inexpensive cost. Hence, we propose using WRECIPROCAL as an alternative to LRU in popular machine learning libraries.

4.5.2 Designing DAGs for Reuse on TIMIT

The TIMIT pipeline tuning task performs random feature kernel approximation followed by 20 iterations of LBFGS. The first stage of the pipeline is a continuous hyperparameter, so we first

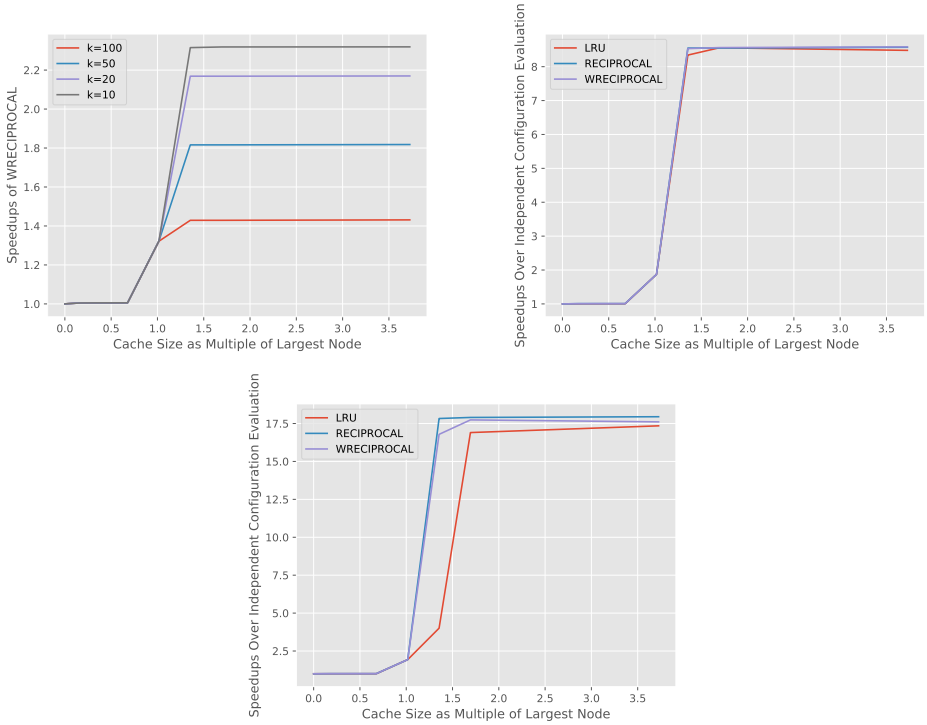


Figure 4.7: Speedups over independent configuration evaluation on TIMIT. Left: Speedups using WRECIPROCAL for a DAG with 100 pipelines generated using gridded random search with different branching factors in the random feature step. As expected, speedups from reuse is higher for DAGs with smaller branching factors. Middle: DAG with 100 pipelines and a maximum downsampling rate of $R/64$ for SHA. Right: DAG with 256 pipelines and a maximum downsampling rate of $R/256$ for SHA.

examine the impact of promoting prefix sharing through gridded random search. Figure 4.7(left) shows the speedups when using WRECIPROCAL over independent configuration evaluation for gridded random search with different branching factors. As expected, speedups increase with lower branching factor, though the speedups are muted for all branching factors due to the long training time for this task. We proceed with a branching factor of 10 for the random feature stage of the pipeline.

Next, we use Successive Halving with training set size as the resource to evaluate the impact on speedups when reducing the portion of time spent on the last stage of the pipeline. We run SHA with the following parameters: $\eta = 4$ and $G = 4$ (which implies $r = R/64$). Figure 4.7 (middle) shows the speedups over independent configuration evaluation increases from $2\times$ for uniform allocation to over $8\times$ when using Successive Halving. Finally, we examine a more realistic setup for SHA where the number of pipelines considered is higher to allow for a more aggressive downsampling rate ($n = 256$, $G = 5$, and $r = R/256$). Figure 4.7 (right) shows that in this case, speedups from reuse via caching reaches over $17\times$.

Part II

Neural Architecture Search

Chapter 5

Random Search Baselines for NAS

In the first part of this thesis, we developed novel algorithms for efficient hyperparameter optimization by exploiting early-stopping to reduce the average computational cost needed to evaluate a hyperparameter configurations. In this second part, we continue the exploration of cost-efficient evaluation methods but focus on the problem of neural architecture search (NAS), a sub-field of hyperparamter optimization that has attracted significant interest recently. We begin, in this chapter, by establishing two random search baselines for NAS to ground results for prior NAS methods and culminate in a detailed discussion of the state of reproducibility in NAS.

5.1 Introduction

Deep learning offers the promise of bypassing the process of manual feature engineering by learning representations in conjunction with statistical models in an end-to-end fashion. However, neural network architectures themselves are typically designed by experts in a painstaking, ad-hoc fashion. Neural architecture search (NAS) presents a promising path for alleviating this pain by automatically identifying architectures that are superior to hand-designed ones. Since the work by Zoph and Le [2017], there has been explosion of research activity on this problem [Bender et al., 2018, Brock et al., 2018, Cai et al., 2019, Elsken et al., 2018a, Jin et al., 2018, Liu et al., 2018a,b, 2019, Negrinho and Gordon, 2017, Pham et al., 2018, Real et al., 2018, Xie et al., 2019, Zhang et al., 2019]. Notably, there has been great industry interest in NAS, as evidenced by the vast computational [Real et al., 2018, Zoph and Le, 2017, Zoph et al., 2018] and marketing resources [Google, 2018b] committed to industry-driven NAS research. However, despite a steady stream of promising empirical results [Cai et al., 2019, Liu et al., 2019, Luo et al., 2018, Real et al., 2018, Zoph and Le, 2017, Zoph et al., 2018], we see three fundamental issues motivating the work in this chapter:

Inadequate Baselines. NAS methods exploit many of the strategies that were initially explored in the context of traditional hyperparameter optimization tasks, e.g., evolutionary search [Jaderberg et al., 2017, Olson and Moore, 2016], Bayesian optimization [Bergstra et al., 2011, Hutter et al., 2011, Snoek et al., 2012], and gradient-based approaches [Bengio, 2000, Maclaurin et al., 2015]. Moreover, the NAS problem is in fact a specialized instance of the broader hyperparameter optimization problem. However, in spite of the close relationship between these

two problems, prior comparisons between novel NAS methods and standard hyperparameter optimization methods are inadequate. In particular, at the time of this work, no state-of-the-art hyperparameter optimization methods have been evaluated on standard NAS benchmarks. *Without benchmarking against leading hyperparameter optimization baselines, it difficult to quantify the performance gains provided by specialized NAS methods.*

Complex Methods. We have witnessed a proliferation of novel NAS methods, with research progressing in many different directions. Some approaches introduce a significant amount of algorithmic complexity in the search process, including complicated training routines [Bender et al., 2018, Cai et al., 2019, Pham et al., 2018, Xie et al., 2019], architecture transformations [Cai et al., 2018, Elsken et al., 2018a, Liu et al., 2018b, Real et al., 2018, Wei et al., 2016], and modeling assumptions [Brock et al., 2018, Jin et al., 2018, Kandasamy et al., 2018, Liu et al., 2018a, Zhang et al., 2019] (see Section 5.2 for more details). While many technically diverse NAS methods demonstrate good empirical performance, they often lack corresponding ablation studies [Cai et al., 2019, Luo et al., 2018, Zhang et al., 2019], and as a result, *it is unclear what NAS component(s) are necessary to achieve a competitive empirical result*.

Lack of Reproducibility. Experimental reproducibility is of paramount importance in the context of NAS research, given the empirical nature of the field, the complexity of new NAS methods, and the steep computational costs associated with empirical evaluation. In particular, there are (at least) two important notions of reproducibility to consider: (1) "exact" reproducibility i.e., whether it is possible to reproduce explicitly reported experimental results; and "broad" reproducibility, i.e., the degree to which the reported experimental results are themselves robust and generalizable. Broad reproducibility is difficult to measure due to the computational burden of NAS methods and the high variance associated with extremal statistics. However, most of the published results in this field do not even satisfy exact reproducibility. *For example, of the 12 papers published since 2018 at NeurIPS, ICML, and ICLR that introduce novel NAS methods (see Table 5.1), none are exactly reproducible.* Indeed, each fails on account of some combination of missing model evaluation code, architecture search code, random seeds used for search and evaluation, and/or undocumented hyperparameter tuning.[1]

While addressing these challenges will require community-wide efforts, in this chapter, we present results that aim to make some initial progress on each of these issues. In particular, our contributions are as follows:

1. We help ground prior NAS results by providing a new perspective on the gap between traditional hyperparameter optimization and leading NAS methods. Specifically, we evaluate the ASHA algorithm presented in Chapter 3 on two standard NAS benchmarks (CIFAR-10 and PTB) and compare our results to DARTS [Liu et al., 2019] and ENAS [Pham et al., 2018], two well-known NAS methods. With approximately the same amount of compute as DARTS, this simple method provides a much more competitive baseline for

[1]It is important to note that these works vary drastically in terms of what materials they provide, and some authors such as Liu et al. [2019], provide a relatively complete codebase for their methods. However, even in the case of DARTS, the code for the CIFAR-10 benchmark is not deterministic and Liu et al. [2019] do not provide random seeds or documentation regarding the post-processing steps in which they perform hyperparameter optimization on final architectures returned by DARTS. We were thus not able to reproduce the results in Liu et al. [2019], but we were able to use the DARTS code repository (https://github.com/quark0/darts) as the launching point for our experimental setup.

both benchmarks: (1) on PTB, random search with early-stopping reaches test perplexity of 56.4 compared to the published result for ENAS of 56.3,[2] and (2) for CIFAR-10, random search with early-stopping achieves a test error of 2.85%, whereas the published result for ENAS is 2.89%. While DARTS still outperforms this baseline, our results demonstrate that the gap is not nearly as large as that suggested by published random search baselines on these tasks [Liu et al., 2019, Pham et al., 2018].

2. We identify a small subset of NAS components that are sufficient for achieving good empirical results. We construct a simple algorithm from the ground up starting from vanilla random search, and demonstrate that properly tuned random search with weight-sharing is competitive with much more complicated methods when using similar computational budgets. In particular, we identify the following meta-hyperparameters that impact the behavior of our algorithm: batch size, number of epochs, network size, and number of evaluated architectures. We evaluate our proposed method using the same search space and evaluation scheme as DARTS. We explore a few modifications of the meta-hyperparameters to improve search quality and make full use of available GPU memory and computational resources, and observe state-of-the-art performance on the PTB benchmark and comparable performance to DARTS on the CIFAR-10 benchmark. We emphasize that we do not perform additional hyperparameter tuning of the final architectures discovered at the end of the search process.
3. We open-source all of the necessary code, random seeds, and documentation necessary to reproduce our experiments. Our single machine results shown in Table 5.2 and Table 5.5 follow a deterministic experimental setup, given a fixed random seed, and satisfy exact reproducibility. For these experiments on the two standard benchmarks, we study the broad reproduciblity of our random search with weight-sharing results by repeating our experiments with different random seeds. We observe non-trivial differences across independent runs and identify potential sources for these differences. Our results highlight the need for more careful reporting of experimental results, increased transparency of intermediate results, and more robust statistics to quantify the performance of NAS methods.

5.2 Related Work

We now provide additional context for the three issues we identified with prior empirical studies in NAS in Section 5.1.

5.2.1 Inadequate Baselines

Prior works in NAS do not provide adequate comparison to random search and other hyperparameter optimization methods. Some works either compare to random search given a budget of

[2]We could not reproduce this result using the initial final architecture and code provided by the authors (https://github.com/melodyguan/enas). They have since released another repository (https://github.com/google-research/google-research/tree/master/enas_lm) that reports reproduced results but we have not verified these figures.

just of few evaluations [Liu et al., 2019, Pham et al., 2018] or Bayesian optimization methods without efficient architecture evaluation schemes [Jin et al., 2018]. While Real et al. [2018] and Cai et al. [2018] provide a thorough comparison to random search, they use random search with full training even though our empirical studies in Chapter 2 and Chapter 3 demonstrate that partial training via early-stopping is often orders-of-magnitude faster than standard random search. Hence, we compare the empirical performance of ASHA (Chapter 3) with that of NAS methods in Section 5.5.

5.2.2 Complex Methods

Much of the complexity of NAS methods is introduced in the process of adapting search methods for NAS-specific search spaces: evolutionary approaches need to define a set of possible mutations to apply to different architectures [Real et al., 2017, 2018]; Bayesian optimization approaches [Jin et al., 2018, Kandasamy et al., 2018] rely on specially designed kernels; gradient-based methods transform the discrete architecture search problem into a continuous optimization problem so that gradients can be applied [Cai et al., 2019, Liu et al., 2019, Luo et al., 2018, Xie et al., 2019]; and reinforcement learning approaches need to train a recurrent neural network controller to generate good architectures [Pham et al., 2018, Zoph and Le, 2017, Zoph et al., 2018]. All of these search approaches add a significant amount of complexity with no clear winner, especially since methods some times use different search spaces and evaluation methods. To simplify the search process and help isolate important components of NAS, we use random search to sample architectures from the search space.

Additional complexity is also introduced by the NAS-specific evaluation methods mentioned previously. Network morphisms require architecture transformations that satisfy certain criteria; hypernetworks and performance prediction methods encode information from previously seen architectures in an auxiliary network; and weight-sharing methods [Bender et al., 2018, Cai et al., 2019, Liu et al., 2019, Pham et al., 2018, Xie et al., 2019] use a single set of weights for all possible architectures and hence, can require careful training routines. Despite their complexity, these more efficient NAS evaluation methods are 1-3 orders-of-magnitude cheaper than full training (see Table 5.5 and Table 5.2), at the expense of decreased fidelity to the true performance. Of these evaluation methods, network morphism still requires on the order of 100 GPU days [Elsken et al., 2018a, Liu et al., 2018a] and, while hypernetworks and prediction performance based methods can be cheaper, weight-sharing is less complex since it does not require training an auxiliary network. In addition to the computational efficiency of weight-sharing methods [Cai et al., 2019, Liu et al., 2019, Pham et al., 2018, Xie et al., 2019], which only require computation on the order of fully training a single architecture, this approach has also achieved the best result on the two standard benchmarks [Cai et al., 2019, Liu et al., 2019]. Hence, we use random search with weight-sharing as our starting point for a simple and efficient NAS method.

Our work is inspired by the result of Bender et al. [2018], which showed that random search, combined with a well-trained set of shared weights can successfully differentiate good architectures from poor performing ones. However, their work required several modifications to stabilize training (e.g., a tunable path dropout schedule over edges of the search DAG and a specialized ghost batch normalization scheme [Hoffer et al., 2017]). Furthermore, they only report experimental results on the CIFAR-10 benchmark, on which they fell slightly short of the

results for ENAS and DARTS. In contrast, our combination of random search with weight-sharing greatly simplifies the training routine and we identify key variables needed to achieve competitive results on both CIFAR-10 and PTB benchmarks.

5.2.3 Lack of Reproducibility

The earliest NAS results lacked exact and broad reproducibility due to the tremendous amount of computation required to achieve the results [Real et al., 2018, Zoph and Le, 2017, Zoph et al., 2018]. Additionally, some of these methods used specialized hardware (i.e., TPUs) that were not easily accessible to researchers at the time [Real et al., 2018]. Although the final architectures were eventually provided [Google, 2018a,c], the code for the search methods used to produce these results has not been released, precluding researchers from reproducing these results even if they had sufficient computational resources.

Table 5.1: **Reproducibility of NAS Publications.** Summary of the reproducibility status of prior NAS publications appearing in top machine learning conferences. For the hyperparameter tuning column, N/A indicates we are not aware that the authors performed additional hyperparameter optimization.
† Published result is not reproducible for the PTB benchmark when training the reported final architecture with provided code.
* Code to reproduce experiments was requested on OpenReview.

Conference	Publication	Architecture Search Code	Model Evaluation Code	Random Seeds	Hyperparameter Tuning
ICLR 2018	Brock et al. [2018]	Yes	Yes	No	N/A
	Liu et al. [2018b]	No	No		
ICML 2018	Pham et al. [2018]†	Yes	Yes	No	Undocumented
	Cai et al. [2018]	Yes	Yes	No	N/A
	Bender et al. [2018]	No	No		
NIPS 2018	Kandasamy et al. [2018]	Yes	Yes	No	N/A
	Luo et al. [2018]	Yes	Yes	No	Grid Search
ICLR 2019	Liu et al. [2019]	Yes	Yes	No	Undocumented
	Cai et al. [2019]	No	Yes	No	N/A
	Zhang et al. [2019]*	No	No		
	Xie et al. [2019]*	No	No		
	Cao et al. [2019]	No	No		

It has become feasible to evaluate the exact and broad reproducibility of many NAS methods due to their reduced computational cost. However, while many authors have released code for their work [e.g., Brock et al., 2018, Cai et al., 2018, Liu et al., 2019, Pham et al., 2018], others have not made their code publicly available [e.g., Xie et al., 2019, Zhang et al., 2019], including the work most closely related to ours by Bender et al. [2018]. We summarize the reproducibility of prior NAS publications at some of the major machine learning conferences in Table 5.1 according to the availability of the following:

1. **Architecture search code.** The output of this code is the final architecture that should be trained on the evaluation task.

2. **Model evaluation code.** The output of this code is the final performance on the evaluation task.
3. **Hyperparameter tuning documentation.** This includes code used to perform hyperparameter tuning of the final architectures, if any.
4. **Random Seeds.** This includes random seeds used for both the search and post-processing (i.e., retraining of final architecture as well as any additional hyperparameter tuning) phases. Most works provide the final architectures but random seeds are required to verify that the search process actually results in those final architectures and the performance of the final architectures matches the published result. Note the random seeds are only useful if the code for search and post-processing phases are deterministic up to a random seed; this was not the case for the DARTS code used for the CIFAR-10 benchmark.

All 4 criteria are necessary for exact reproducibility. Due to the absence of random seeds for all methods with released code, none of the methods in Table 5.1 are exactly reproducible from the search phase to the final architecture evaluation phase.

While only criteria 1–3 are necessary to estimate broad reproducibility, there is minimal discussion of the broad reproducibility of prior methods in published work. With the exception of NASBOT [Kandasamy et al., 2018] and DARTS [Liu et al., 2019], the methods in Table 5.1 only report the performance of the best found architecture, presumably resulting from a single run of the search process. Although this is understandable in light of the computational costs for some of these methods [Cai et al., 2018, Luo et al., 2018], the high variance of extremal statistics makes it difficult to isolate the impact of the novel contributions introduced in each work. DARTS is particularly commendable in acknowledging its dependence on random initialization, prompting the use multiple runs to select the best architecture. In our experiments in Section 5.5, we follow DARTS and report the result of our random weight-sharing method across multiple trials; in fact, we go one step further and evaluate the broad reproducibility of our results with multiple sets of random seeds.

5.3 Background on Weight-Sharing

In this section, we first provide relevant background on weight-sharing to contexualize our novel algorithm combining random search with weight-sharing in the next section.

5.3.1 Search Spaces

Since weight-sharing methods reduce the architecture evaluation cost to that of training a single supernetwork encompassing all architectures, the memory requirement of weight-sharing is significantly higher than that of a standalone architecture. Consequently, weight-sharing is usually used to search smaller cell-based search spaces so that the supernetwork can fit within the memory of a single GPU. We will review the CNN and RNN cell-based search spaces considered by Liu et al. [2019] to ground our discussion.

5.3.1.1 Convolutional Cell

Recall from Chapter 1.3 that NAS search spaces can be represented as directed acyclic graphs (DAGs). The nodes of the DAG represent intermediate features and the directed edges correspond to transformations applied to the features from the source node which are then passed to the destination node. Hence, a search space $\mathcal{A}$ is fully defined by the set of possible DAGs and the set of possible transformations O that can be applied at each edge.

The CNN search space considered by Liu et al. [2019] includes cells represented by DAGs with N total nodes, including two input nodes followed by $N-2$ intermediate nodes. Valid architecture DAGs must further satisfy the following constraints:

1. Edges can only go from lower indexed nodes to higher indexed nodes, where the nodes are indexed according to their topological order in the DAG.
2. Each intermediate node has two incoming edges.

Each selected edge is then associated with an operation from a set of transformations O with the following possible operations: zero, max pooling, average pooling, skip connection, separable convolution (3x3), separable convolution (5x5), dilated convolution (3x3), and dilated convolution (5x5). Given this setup, we compute the resulting feature at node j as

$$\sum_{i<j} o^{(i,j)}(x^{(i)}),$$

where $x^{(i)}$ denotes the latent representation at node i and $o^{(i,j)}$ denotes the selected operation associated with the edge from node i to node j; note there will only be two edges with nonzero operations do the the edge constraints.

The final architecture consists of both a normal cell and a reduction cell from this search space. The normal cell is applied with stride length 1 to maintain the feature dimension and the reduction cell is applied with stride length 2 to reduce the feature dimension by a factor of 2. Hence, this search space contains a total of

$$\left(\prod_{n=2}^{N-1}\binom{n}{2}|O|^2\right)^2$$

possible architectures. For the search space considered by Liu et al. [2019] in their experiments with $N=6$ nodes, there are a total of $9.1\text{e}18$ possible architectures.

5.3.1.2 Recurrent Cell

The recurrent cell search space considered by Liu et al. [2019] includes DAGs with one input node indexed 0 and a variable number of intermediate nodes. Valid architecture DAGs must further satisfy the following constraints:

1. Edges can only go from lower indexed nodes to higher indexed nodes.
2. Each intermediate node has one incoming edge.

Each selected edge is then associated with an operation from a set of transformations $O = \{\text{zero, tanh, ReLU, sigmoid, identity}\}$. Hence, for a search space including DAGs with a total of N nodes, there are a total of $(N-1)!|O|^(N-1)$. For the actual search space considered by Liu et al. [2019] in their experiments with $N=9$ nodes, there are a total of 15.8 billion architectures.

5.3.2 The Weight-Sharing Paradigm

Given a cell-based search space, the supernetwork trained by weight-sharing methods includes all possible edges and operations per edge. More concretely, the supernetwork contains all edges (i, j) with $i < j$ and there is an instantiation of every operation from the set O for each edge. Therefore, for a search space with N nodes, relative to a standalone architecture, the weight-sharing network would have $O(N^2)$ edges instead of $O(N)$ edges and $O(ON^2)$ instead of $O(N)$ total operations.

The supernetwork is further parameterized by a set of architecture weights α that provide an interface for us to configure the supernetwork during the architecture search process. Let $\alpha^{(i,j)} \in [0, 1]^{|O|}$ correspond to a set of weights on the operations at edge (i, j). Then, the feature at node j can be computed as

$$\sum_{i<j} \sum_{k=1}^{|O|} \alpha_k^{(i,j)} o_k^{(i,j)}(x^{(i)}),$$

where k indexes the operation. We can then use the architecture parameters α to configure the supernetwork to individual architectures from the search space $\mathcal{A}$ by setting the weights for active edge-operation pairs to 1 and the rest to 0. As we will see in the Chapter 6, besides random search, weight-sharing based NAS methods also apply different search methods to the architecture parameters of the weight-sharing supernetwork to identify promising architectures.

5.4 Random Search with Weight-Sharing

We now introduce our NAS algorithm that combines random search with weight-sharing. Our algorithm can be applied to any search space that is compatible with weight-sharing. In our experiments in Section 5.5, we use the same search spaces as that considered by DARTS [Liu et al., 2019] for the standard CIFAR-10 and PTB NAS benchmarks.

For concreteness, consider the search space used by DARTS for designing a recurrent cell for the PTB benchmark with $N = 8$ nodes. To sample an architecture from this search space, we apply random search in the following manner:

1. For each node in the DAG, determine what decisions must be made. In the case of the RNN search space, we need to choose a node as input and a corresponding operation to apply to generate the output of the node.
2. For each decision, identify the possible choices for the given node. In the case of the RNN search space, if we number the nodes from 1 to N, node i can take the outputs of nodes 0 to node $i - 1$ as input (the initial input to the cell is index 0 and is also a possible input). Additionally, we can choose an operation to apply to the selected input.
3. Finally, moving from node to node, we sample uniformly from the set of possible choices for each decision that needs to be made.

Figure 5.1 shows an example of an architecture from this search space.

In order to combine random search with weight-sharing, we simply use randomly sampled architectures to train the shared weights. Shared weights are updated by selecting a single architecture for a given minibatch and updating the shared weights by back-propagating through

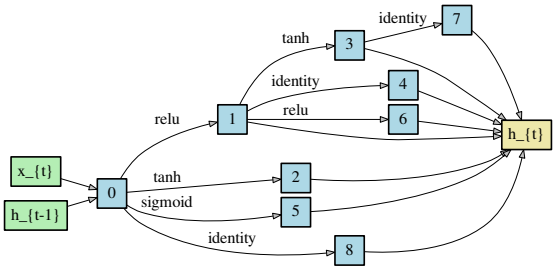


Figure 5.1: **Recurrent Cell on PTB Benchmark.** The best architecture found by random search with weight-sharing in Section 5.5.1 is depicted. Each numbered square is a node of the DAG and each edge represents the flow of data from one node to another after applying the indicated operation along the edge. Nodes with multiple incoming edges (i.e., node 0 and output node $\mathrm{h\_\{t\}}$ concatenate the inputs to form the output of the node.

the network with only the edges and operations as indicated by the architecture activated. Hence, the number of architectures used to update the shared weights is equivalent to the total number of minibatch training iterations.

After training the shared weights for a certain number of epochs, we use these trained shared weights to evaluate the performance of a number of randomly sampled architectures on a separate held out dataset. We select the best performing one as the final architecture, i.e., as the output of our search algorithm.

5.4.1 Relevant Meta-Hyperparameters

There are a few key meta-hyperparameters that impact the behavior of our search algorithm. We describe each of them below, along with a description of how we expect them to impact the search algorithm, both in terms of search quality and computational costs.

1. **Training epochs.** Increasing the number of training epochs while keeping all other parameters the same increases the total number of minibatch updates and hence, the number of architectures used to update the shared weights. Intuitively, training with more architectures should help the shared weights generalize better to what are likely unseen architectures in the evaluation step. Unsurprisingly, more epochs increase the computational time required for architecture search.
2. **Batch size.** Decreasing the batch size while keeping all other parameters the same also increases the number of minibatch updates but at the cost of noisier gradient update. Hence, we expect reducing the batch size to have a similar effect as increasing the number of training epochs but may necessitate adjusting other meta-hyperparameters to account for the noisier gradient update. Intuitively, more minibatch updates increase the computational time required for architecture search.
3. **Network size.** Increasing the search network size increases the dimension of the shared weights. Intuitively, this should boost performance since a larger search network can store

more information about different architectures. Unsurprisingly, larger networks require more GPU memory.

4. **Number of evaluated architectures.** Increasing the number of architectures that we evaluate using the shared weights allows for more exploration in the architecture search space. Intuitively, this should help assuming that there is a high correlation between the performance of an architecture evaluated using shared weights and the ground truth performance of that architecture when trained from scratch [Bender et al., 2018]. Unsurprisingly, evaluating more architectures increases the computational time required for architecture search.

Other learning meta-hyperparameters will likely need to be adjusted accordingly for different settings of the key relevant meta-hyperparameters listed above. In our experiments in Section 5.5, we tune *gradient clipping* as a fifth meta-hyperparameter, though there are other possible meta-hyperparameters that may benefit from additional tuning (e.g., learning rate, momentum).

In Section 5.5, following these intuitions, we incrementally explore the design space of our search method in order to improve search quality and make full use of the available GPU memory and computational resources.

5.4.2 Memory Footprint

Since we train the shared weights using a single architecture at a time, we have the option of only loading the weights associated with the operations and edges that are activated into GPU memory. Hence, the memory footprint of our random search with weight-sharing can be reduced to that of a single model. In this sense, our approach is similar to ProxylessNAS [Cai et al., 2019] and allows us to perform architecture search with weight-sharing on the larger "proxyless" models that are usually used in the final architecture evaluation step instead of the smaller proxy models that are usually used in the search step. We take advantage of this in a subset of our experiments for the PTB benchmark in Section 5.5.1; performing random search with weight-sharing on a proxyless network for the CIFAR-10 benchmark is a direction for future work.

In contrast, Bender et al. [2018] train the shared weights with a path dropout schedule that incrementally prunes edges within the DAG so that the sub-DAGs used to train the shared weights become sparser as training progresses. Under this training routine, since most of the edges in the search DAG are activated in the beginning, the memory footprint cannot be reduced to that of a single model to allow a proxyless network for the shared weights.

5.5 Experiments

In line with prior work [Liu et al., 2019, Pham et al., 2018, Zoph and Le, 2017], we consider the two standard benchmarks for neural architecture search: (1) language modeling on the Penn Treebank (PTB) dataset [Marcus et al., 1993] and (2) image classification on CIFAR-10 [Krizhevsky, 2009]. For each of these benchmarks, we consider the same search space and use much of the same experimental setups as DARTS [Liu et al., 2019], and by association SNAS [Xie et al., 2019], to facilitate a fair comparison of our results to prior work.

To evaluate the performance of random search with weight-sharing on these two benchmarks, we proceed in the same three stages as Liu et al. [2019]:

- **Stage 1**: Perform architecture search for a cell block on a cheaper search task.
- **Stage 2**: Evaluate the best architecture from the first stage by retraining a larger, network formed from multiple cell blocks of the best found architecture from scratch. This stage is used to select the best architecture from multiple trials.
- **Stage 3**: Perform the full evaluation of the best found architecture from the second stage by either training for more epochs (PTB) or training with more seeds (CIFAR-10).

We start with the same meta-hyperparameter settings used by DARTS to train the shared weights. Then, we incrementally modify the meta-hyperparameters identified in Section 5.4.1 to improve performance until we either reach state-of-the-art performance (for PTB) or match the performance of DARTS and SNAS (for CIFAR-10).

For our evaluation of random search with early-stopping (i.e., ASHA) on these two benchmarks, we perform architecture search using partial training of the stage (2) evaluation network and then select the best architecture for stage (3) evaluation. For both benchmarks, we run ASHA with a starting resource per architecture of $r = 1$ epoch, a maximum resource of 300 epochs, and a promotion rate of $\eta = 4$, indicating the top $1/4$ of architectures will be promoted in each round and trained for $4\times$ more resource.

5.5.1 PTB Benchmark

We now present results for the PTB benchmark. We use the DARTS search space for the recurrent cell, which is described in Section 5.4. For this benchmark, due to higher memory requirements for their mixture operation, DARTS used a small recurrent network with embedding and hidden dimension of 300 to perform the architecture search followed by a larger network with embedding and hidden dimension of 850 to perform the evaluation. For the PTB benchmark, we refer to the network used in the first stage as the *proxy* network and the network in the later stages as the *proxyless* network. We next present the final search results. We subsequently explore the impact of various meta-hyperparameters on random search with weight-sharing, and finally evaluate the reproducibility of various methods on this benchmark.

5.5.1.1 Final Search Results

We now present our final evaluation results in Table 5.2. Specifically, we report the output of stage (3), in which we train the proxyless network configured according to the best architectures found by different methods for 3600 epochs. This setup matches the evaluation scheme used for the reported results in Table 2 of Liu et al. [2019] (see Section 5.7.2 for more details). We discuss various aspects of these results in the context of the three issues—baselines, complex methods, reproducibility—introduced in Section 5.1.

First, we evaluate the ASHA baseline using 2 GPU days, which is equivalent to the total cost of DARTS (second order). In contrast to the one random architecture evaluated by Pham et al. [2018] and the 8 evaluated by Liu et al. [2019] for their random search baselines, ASHA evaluated over 300 architectures with the allotted computation time. The best architecture found by ASHA achieves a test perplexity of 56.4, which is comparable to the published result for ENAS and significantly better than the random search baseline provided by Liu et al. [2019], DARTS (first

Table 5.2: **PTB Benchmark: Comparison with NAS methods and manually designed networks.** Lower test perplexity is better on this benchmark. The results are grouped by those for manually designed networks, published NAS methods, and the methods that we evaluated. Table entries denoted by "-" indicate that the field does not apply, while entries denoted by "N/A" indicate unknown entries. The search cost, unless otherwise noted, is measured in GPU days. Note that the search cost is hardware dependent and the search cost shown for our results are calculated for Tesla P100 GPUs; all other numbers are those reported by Liu et al. [2019].
[#] Search cost is in CPU-days.
[*] We could not reproduce this result using the code released by the authors at https://github.com/melodyguan/enas.
[†] The stage (1) cost shown is that for 1 trial as opposed to the cost for 4 trials shown for DARTS and Random search WS. It is unclear whether ENAS requires multiple trials followed by stage (2) evaluation in order to find a good architecture.

		Test Perplexity		Params	Search Cost			Comparable	Search
Architecture	**Source**	Valid	Test	(M)	Stage 1	Stage 2	Total	**Search Space?**	**Method**
LSTM + DropConnect	[Merity et al., 2018]	60.0	57.3	24	-	-	-	-	manual
ASHA + LSTM + DropConnect	Chapter 3.4.4	58.1	56.3	24	-	-	13	N	HP-tuned
LSTM + MoS	[Yang et al., 2018]	56.5	54.4	22	-	-	-	-	manual
NAS[#]	[Zoph and Le, 2017]	N/A	64.0	25	-	-	1e4	N	RL
ENAS[*†]	[Pham et al., 2018]	N/A	56.3	24	0.5	N/A	N/A	Y	RL
ENAS[†]	[Liu et al., 2019]	60.8	58.6	24	0.5	N/A	N/A	Y	random
Random search baseline	[Liu et al., 2019]	61.8	59.4	23	-	-	2	Y	random
DARTS (first order)	[Liu et al., 2019]	60.2	57.6	23	0.5	1	1.5	Y	gradient-based
DARTS (second order)	[Liu et al., 2019]	58.1	55.7	23	1	1	2	Y	gradient-based
DARTS (second order)	Ours	58.2	55.9	23	1	1	2	Y	gradient-based
ASHA baseline	Ours	58.6	56.4	23	-	-	2	Y	random
Random search WS	Ours	57.8	55.5	23	0.25	1	1.25	Y	random

order), and the reproduced result for ENAS [Liu et al., 2019]. Our result demonstrates that the gap between competitive NAS methods and standard hyperparameter optimization approaches on the PTB benchmark is significantly smaller than that suggested by the prior comparisons to random search [Liu et al., 2019, Pham et al., 2018].

Next, we evaluate random search with weight-sharing with tuned meta-hyperparameters (see Section 5.5.1.2 for details). With slightly lower search cost than DARTS, this method finds an architecture that reaches test perplexity 55.5, achieving state-of-the-art (SOTA) perplexity compared to previous NAS approaches. We note that manually designed architectures are competitive with RNN cells designed by NAS methods on this benchmark. In fact, the work by Yang et al. [2018] using LSTM with mixture of experts in the softmax layer (MoS) outperforms automatically designed cells. Our architecture would likely also improve significantly with MoS, but we train without MoS to provide a fair comparison to ENAS and DARTS.

Finally, we examine the reproducibility of the NAS methods with available code for both architecture search and evaluation. For DARTS, exact reproducibility was not feasible since Liu et al. [2019] do not provide random seeds for the search process; however, we were able to reproduce the performance of their reported best architecture. We also evaluated the broad reproducibility of DARTS through an independent run, which reached a test perplexity of 55.9, compared to the published value of 55.7. For ENAS, end-to-end exact reproducibility was infeasible due to non-deterministic code and missing random seeds for both the search and

evaluation steps. Additionally, when we tried to reproduce their result using the provided final architecture, we could not match the reported test perplexity of 56.3 in our rerun. Consequently, in Table 5.2 we show the test perplexity for the final architecture found by ENAS trained using the DARTS code base, which Liu et al. [2019] observed to give a better test perplexity than using the architecture evaluation code provided by ENAS. We next considered the reproducibility of random search with weight-sharing. We verified the exact reproducibility of our reported results, and then investigated their broad reproducibility by running another experiment with different random seeds. In this second experiment, we observed a final text perplexity of 56.5, compared with a final test perplexity of 55.5 in the first experiment. Our detailed investigation in Section 5.5.1.3 shows that the discrepancies across both DARTS and random search with weight-sharing are unsurprising in light of the differing convergence rates among architectures on this benchmark.

5.5.1.2 Impact of Meta-Hyperparameters

We now detail the meta-hyperparameter settings that we tried for random search with weight-sharing in order to achieve SOTA performance on the PTB benchmark. Similar to DARTS, in these preliminary experiments we performed 4 separate trials of each version of random search with weight-sharing, where each trial consists of executing stage (1) followed by stage (2). In stage (1), we train the shared weights and then use them to evaluate 2000 randomly sampled architectures. In stage (2), we select the best architecture out of 2000, according to the shared weights, to train from scratch using the proxyless network for 300 epochs.

We incrementally tune random search with weight-sharing by adjusting the following meta-hyperparameters associated with training the shared weights in stage (1): (1) gradient clipping, (2) batch size, and (3) network size. The settings we consider proceed as follows:

- **Random (1):** We train the shared weights of the proxy network using the same setup as DARTS with the same values for number of epochs, batch size, and gradient clipping; all other meta-hyperparameters are the same.
- **Random (2):** We decrease the maximum gradient norm to account for discrete architectures, as opposed to the weighted combination used by DARTS, so that gradient updates are not as large in each direction.
- **Random (3):** We decrease batch size from 256 to 64 in order to increase the number of architectures used to train the shared weights.
- **Random (4):** We train the larger proxyless network architecture with shared weights instead of the proxy network, thereby significantly increasing the number of parameters in the model.

The stage (2) performance of the final architecture after retraining from scratch for each of these settings is shown in Table 5.3. With the extra capacity in the larger network used in Random (4), random search with weight-sharing achieves average validation perplexity of 64.7 across 4 trials, with the best architecture (shown in Figure 5.1 in Section 5.4) reaching 63.8. In light of these stage (2) results, we focused in stage (3) on the best architecture found by Random (4) Run 1, and achieved test perplexity of 55.5 after training for 3600 epochs as reported in Table 5.2.

Table 5.3: **PTB Benchmark: Comparison of Stage (2) Intermediate Search Results for Weight-Sharing Methods.** In stage (1), random search is run with different settings to train the shared weights. The resulting networks are used to evaluate 2000 randomly sampled architectures. In stage (2), the best of these architectures for each trial is then trained from scratch for 300 epochs. We report the performance of the best architecture after stage (2) across 4 trials for each search method.

	Setting				**Trial**					
Method	Network Config	Epochs	Batch Size	Gradient Clipping	1	2	3	4	Best	Average
DARTS [Liu et al., 2019]	proxy	50	256	0.25	67.3	66.3	63.4	63.4	**63.4**	**65.1**
Reproduced DARTS	proxy	50	256	0.25	64.5	67.7	64.0	67.7	64.0	66.0
Random (1)	proxy	50	256	0.25	65.6	66.3	66.0	65.6	65.6	65.9
Random (2)	proxy	50	256	0.1	65.8	67.7	65.3	64.9	64.9	65.9
Random (3)	proxy	50	64	0.1	66.1	65.0	64.9	64.5	64.5	65.1
Random (4) Run 1	proxyless	50	64	0.1	66.3	64.6	64.1	63.8	**63.8**	**64.7**
Random (4) Run 2	proxyless	50	64	0.1	63.9	64.8	66.3	66.7	63.9	65.4

Table 5.4: **PTB Benchmark: Ranking of Intermediate Validation Perplexity.** Architectures are retrained from scratch using the proxyless network and the validation perplexity is reported after training for the indicated number of epochs. The final test perplexity after training for 3600 epochs is also shown for reference.

	Validation Perplexity by Epoch										**Test Perplexity**	
	300		500		1600		2600		3600			
Search Method	Value	Rank	Value	Rank	Value	Rank	Value	Rank	Value	Rank	Value	Rank
DARTS	64.0	4	61.9	2	59.5	2	58.5	2	58.2	2	55.9	2
ASHA	63.9	2	62.0	3	59.8	4	59.0	3	58.6	3	56.4	3
Random (4) Run 1	63.8	1	61.7	1	59.3	1	58.4	1	57.8	1	55.5	1
Random (4) Run 2	63.9	2	62.1	4	59.6	3	59.0	3	58.8	4	56.5	4

5.5.1.3 Investigating Reproducibility

We next examine the stage (2) intermediate results in Table 5.3 in the context of reproducibility. The first two rows of Table 5.3 show a comparison of the published stage (2) results for DARTS and our independent runs of DARTS. Both the best and average across 4 trials are worse in our reproduction of their results. Additionally, as previously mentioned, we perform an additional run of Random (4) with 4 different random seeds to test the broad reproducibility our result. The minimum stage (2) validation perplexity over these 4 trials is 63.9, compared to a minimum validation perplexity of 63.8 for the first set of seeds.

Next, in Table 5.4 we compare the validation perplexities of the best architectures from ASHA, Random (4) Run 1, Random (4) Run 2, and our independent run of DARTS after training each from scratch for up to 3600 epochs. The swap in relative ranking across epochs demonstrates the risk of using noisy signals for the reward. In this case, we see that even partial training for 300 epochs does not recover the correct ranking; training using shared weights further obscures the signal. The differing convergence rates explain the difference in final test perplexity of the best architecture from Random (4) Run 2 and those from DARTS and Random (4) Run 1, despite Random (4) Run 2 reaching a comparable perplexity after 300 epochs.

Overall, the results of Tables 5.3 and 5.4 demonstrate a high variance in the stage (2) inter-

mediate results across trials, along with issues related to differing convergence rates for different architectures. These two issues help explain the differences between the independent runs of DARTS and random search with weight-sharing. A third potential source of variation, which could in particular adversely impact our random search with weight-sharing results, stems from the fact that we did not perform any additional hyperparameter tuning in stage (3); instead we used the same training hyperparameters that were tuned by Liu et al. [2019] for the final architecture found by DARTS.

5.5.2 CIFAR-10 Benchmark

We next present results for the CIFAR-10 benchmark. The DAG considered for the convolutional cell has $N = 4$ search nodes and the operations considered include 3×3 and 5×5 separable convolutions, 3×3 and 5×5 dilated separable convolutions, 3×3 max pooling, and 3×3 average pooling, and zero [Liu et al., 2019]. To sample an architecture from this search space, we have to choose, for each node, 2 input nodes from previous nodes and associated operations to perform on each input (there are two initial inputs to the cell that are also possible input); we sample in this fashion twice, once for the normal convolution cell and one for the reduction cell (e.g., see Figure 5.2). Note that in contrast to DARTS, we include the zero operation when choosing the final architecture for each trial for further evaluation in stages (2) and (3). We hypothesize that our results may improve if we impose a higher complexity on the final architectures by excluding the zero op.

Due to higher memory requirements for weight-sharing, Liu et al. [2019] uses a smaller network with 8 stacked cells and 16 initial channels to perform the convolutional cell search, followed by a larger network with 20 stacked cells and 36 initial channels to perform the evaluation. Again, we will refer to the network used in the first stage as the proxy network and the network in the second stage the proxyless network.

Similar to the PTB results in Section 5.5.1, we will next present the final search results for the CIFAR-10 benchmark, and then dive deeper into these results to explore the impact of meta-hyperparameters on stage (2) intermediate results, and finally evaluate associated reproducibility ramifications.

5.5.2.1 Final Search Results

We now present our results after performing the final evaluation in stage (3). We use the same evaluation scheme used to produce the results in Table 1 of Liu et al. [2019]. In particular, we train the proxyless network configured according to the best architectures found by different methods with 10 different seeds and report the average and standard deviation. Again, we discuss our results in the context of the three issues introduced in Section 5.1.

First, we evaluate the ASHA baseline using 9 GPU days, which is comparable to the 10 GPU days we allotted to our independent run of DARTS. In contrast to the one random architecture evaluated by Pham et al. [2018] and the 24 evaluated by Liu et al. [2019] for their random search baselines, ASHA evaluated over 700 architectures in the allotted computation time. The best architecture found by ASHA achieves an average error of 3.03 ± 0.13, which is significantly better than the random search baseline provided by Liu et al. [2019] and comparable to DARTS (first

Table 5.5: **CIFAR-10 Benchmark: Comparison with NAS methods and manually designed networks.** The results are grouped by those for manually designed networks, published NAS methods, and the methods that we evaluated. Models for all methods are trained with auxiliary towers and cutout. Test error for our contributions are averaged over 10 random seeds. Table entries denoted by "-" indicate that the field does not apply, while entries denoted by "N/A" indicate unknown entries. The search cost is measured in GPU days. Note that the search cost is hardware dependent and the search cost shown for our results are calculated for Tesla P100 GPUs; all other numbers follow those reported by Liu et al. [2019].
* We show results for the variants of these networks with comparable number of parameters. Larger versions of these networks achieve lower errors.
# Reported test error averaged over 5 seeds.
† The stage (1) cost shown is that for 1 trial as opposed to the cost for 4 trials shown for DARTS and Random search WS. It is unclear whether the method requires multiple trials followed by stage (2) evaluation in order to find a good architecture.
‡ Due to the longer evaluation we employ in stage (2) to account for unstable rankings, the cost for stage (2) is 1 GPU day for results reported by Liu et al. [2019] and 6 GPU days for our results.

		Test Error		Params	Search Cost			Comparable	Search
Architecture	**Source**	Best	Average	(M)	Stage 1	Stage 2	Total	**Search Space?**	**Method**
Shake-Shake#	[Devries and Taylor, 2017]	N/A	2.56	26.2	-	-	-	-	manual
PyramidNet	[Yamada et al., 2018]	2.31	N/A	26	-	-	-	-	manual
NASNet-A#*	[Zoph et al., 2018]	N/A	2.65	3.3	-	-	2000	N	RL
AmoebaNet-B*	[Real et al., 2018]	N/A	2.55 ± 0.05	2.8	-	-	3150	N	evolution
ProxylessNAS†	[Cai et al., 2019]	2.08	N/A	5.7	4	N/A	N/A	N	gradient-based
GHN#†	[Zhang et al., 2019]	N/A	2.84 ± 0.07	5.7	0.84	N/A	N/A	N	hypernetwork
SNAS†	[Xie et al., 2019]	N/A	2.85 ± 0.02	2.8	1.5	N/A	N/A	Y	gradient-based
ENAS†	[Pham et al., 2018]	2.89	N/A	4.6	0.5	N/A	N/A	Y	RL
ENAS	[Liu et al., 2019]	2.91	N/A	4.2	4	2	6	Y	RL
Random search baseline	[Liu et al., 2019]	N/A	3.29 ± 0.15	3.2	-	-	4	Y	random
DARTS (first order)	[Liu et al., 2019]	N/A	3.00 ± 0.14	3.3	1.5	1	2.5	Y	gradient-based
DARTS (second order)	[Liu et al., 2019]	N/A	2.76 ± 0.09	3.3	4	1	5	Y	gradient-based
DARTS (second order)‡	Ours	2.62	2.78 ± 0.12	3.3	4	6	10	Y	gradient-based
ASHA baseline	Ours	2.85	3.03 ± 0.13	2.2	-	-	9	Y	random
Random search WS‡	Ours	2.71	2.85 ± 0.08	4.3	2.7	6	8.7	Y	random

order). Additionally, the best performing seed reached a test error of 2.85, which is lower than the published result for ENAS. Similar to the PTB benchmark, these results suggest that the gap between SOTA NAS methods and standard hyperparameter optimization is much smaller than previously reported [Liu et al., 2019, Pham et al., 2018].

Next, we evaluate random search with weight-sharing with tuned meta-hyperparameters (see Section 5.5.2.2 for details). This method finds an architecture that achieves an average test error of 2.85 ± 0.08, which is comparable to the reported results for SNAS and DARTS, the top 2 weight-sharing algorithms that use a comparable search space, as well as GHN [Zhang et al., 2019]. Note that while the two manually tuned architectures we show in Table 5.5 outperform the best architecture discovered by random search with weight-sharing, they have over $7\times$ more parameters. Additionally, the best-performing efficient NAS method, ProxylessNAS, uses a larger proxyless network and a significantly different search space than the one we consider. As mentioned in Section 5.4, random search with weight-sharing can also directly search over larger proxyless networks since it trains using discrete architectures. We hypothesize that using a

proxyless network and applying random search with weight-sharing to the same search space as ProxylessNAS would further improve our results; we leave this as a direction for future work.

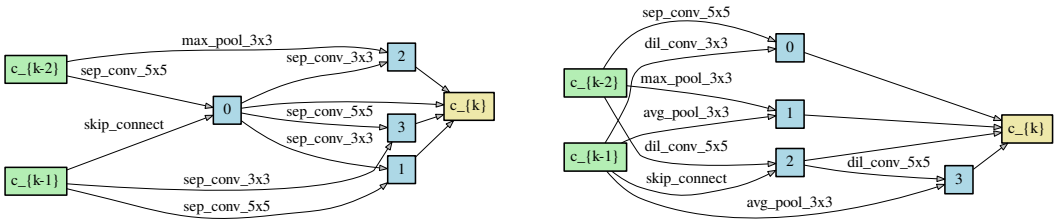
Finally, we examine the reproducibility of the NAS methods using a comparable search space with available code for both architecture search and evaluation (i.e., DARTS and ENAS; to our knowledge, code is not currently available for SNAS). For DARTS, exact reproducibility was not feasible since the code is non-deterministic and Liu et al. [2019] do not provide random seeds for the search process; hence, we focus on broad reproducibility of the results. In our independent run, DARTS reached an average test error of 2.78 ± 0.12 compared to the published result of 2.76 ± 0.09. Notably, we observed that the process of selecting the best architecture in stage (2) is unstable when training stage (2) models for only 100 epochs; see Section 5.5.2.3 for details. Hence, we use 600 epochs in all of our CIFAR experiments, including our independent DARTS run, which explains the discrepancy in stage (2) costs between original DARTS and our independent run. For ENAS, the published results do not satisfy exact reproducibility due to the same issues as those for DARTS. We show in Table 5.5 the broad reproducibility experiment conducted by Liu et al. [2019] for ENAS; here, ENAS found an architecture that achieved a comparable test error of 2.91 in $8\times$ the reported stage (1) search cost. As with the PTB benchmark, we then investigated the reproducibility of random search with weight-sharing. We verified exact reproducibility and then examined broad reproducibility by evaluating 5 additional independent runs of our method. We observe performance below 2.90 test error in 2 of the 5 runs and an average of 2.92 across all 6 runs. We investigate various sources for these discrepancies in Section 5.5.2.3.

Table 5.6: **CIFAR-10 Benchmark: Comparison of Stage (2) Intermediate Search Results for Weight-Sharing Methods.** In stage (1), random search is run with different settings to train the shared weights. The shared weights are then used to evaluate the indicated number of randomly sampled architectures. In stage (2), the best of these architectures for each trial is then trained from scratch for 600 epochs. We report the performance of the best architecture after stage (2) for each trial for each search method.
† This run was performed using the DARTS code before we corrected for non-determinism (see Section 5.7.2).

	Setting				**Trial**					
Method	Epochs	Gradient Clipping	Initial Channels	# Archs Evaluated	1	2	3	4	Best	Average
Reproduced DARTS†	50	5	16	-	2.92	2.77	3.00	3.05	**2.77**	**2.94**
Random (1)	50	5	16	1000	3.25	4.00	2.98	3.58	2.98	3.45
Random (2)	150	5	16	5000	2.93	3.80	3.19	2.96	2.93	3.22
Random (3)	150	1	16	5000	3.50	3.42	2.97	2.95	2.97	3.21
Random (4)	300	1	16	11000	3.04	2.90	3.14	3.09	2.90	3.04
Random (5) Run 1	150	1	24	5000	2.96	3.33	2.83	3.00	**2.83**	**3.03**

5.5.2.2 Impact of Meta-Hyperparameters

We next detail the meta-hyperparameter settings that we tried in order to reach competitive performance on the CIFAR-10 benchmark via random search with weight-sharing. Similar to DARTS, and as with the PTB benchmark, in these preliminary experiments we performed 4 separate trials of each version of random search with weight-sharing, where each trial consists



(a) Normal Cell

(b) Reduction Cell

Figure 5.2: **Convolutional Cells on CIFAR-10 Benchmark:** Best architecture found by random search with weight-sharing.

of executing stage (1) followed by stage (2). In stage (1), we train the shared weights and use them to evaluate a given number of randomly sampled architectures on the test set. In stage (2), we select the best architecture, according to the shared weights, to train from scratch using the proxyless network for 600 epochs.

We incrementally tune random search with weight-sharing by adjusting the following meta-hyperparameters that impact both the training of shared weights and the evaluation of architectures using these trained weights: number of training epochs, gradient clipping, number of architectures evaluated using shared weights, and network size. The settings we consider for random search proceed as follows:

- **Random (1):** We start by training the shared weights with the proxy network used by DARTS and default values for number of epochs, gradient clipping, and number of initial filters; all other meta-hyperparameters are the same.
- **Random (2):** We increase the number of training epochs from 50 to 150, which concurrently increases the number of architectures used to update the shared weights.
- **Random (3):** We reduce the maximum gradient norm from 5 to 1 to adjust for discrete architectures instead of the weighted combination used by DARTS.
- **Random (4):** We further increase the number of epochs for training the proxy network with shared weights to 300 and increase the number of architectures evaluated using the shared weights to 11k.
- **Random (5):** We separately increase the proxy network size to be as large as possible given the available memory on a Nvidia Tesla P100 GPU (i.e. by $\approx 50\%$ due to increasing the number of initial channels from 16 to 24).

The performance of the final architecture after retraining from scratch for each of these settings is shown in Table 5.6. Similar to the PTB benchmark, the best setting for random search was Random (5), which has a larger network size. The best trial for this setting reached a test error of 2.83 when retraining from scratch; we show the normal and reduction cells found by this trial in Figure 5.2. In light of these stage (2) results, we focus in stage (3) on the best architecture found by Random (5) Run 1, and achieve an average test error of 2.85 ± 0.08 over 10 random seeds as shown in Table 5.5.

5.5.2.3 Investigating Reproducibility

Our results in this section show that although DARTS appears broadly reproducible, this result is surprising given the unstable ranking in architectures observed between 100 and 600 epochs for stage (2) evaluation. To begin, the first row of Table 5.6 shows our reproduced results for DARTS after training the best architecture for each trial from scratch for 600 epochs. In our reproduced run, DARTS reaches an average test error of 2.94 and a minimum of 2.77 across 4 trials (see Table 5.6). Note that this is not a direct comparison to the published result for DARTS since there, the stage (2) evaluation was performed after training for only 100 epochs.

Table 5.7: **CIFAR-10 Benchmark: Ranking of Intermediate Test Error for DARTS.** Architectures are retrained from scratch using the proxyless network and the error on the test set is reported after training for the indicated number of epochs. Rank is calculated across the 4 trials. We also show the average over 10 seeds for the best architecture from the top trial for reference.
† These results were run before we fixed the non-determinism in DARTS code (see Section 5.7.2).

		Epochs				Across 10 Seeds	
Search		100		600			
Method	Trial	Value	Rank	Value	Rank	Min	Avg
Reproduced	1	7.63	2	2.92	2		
Darts†	2	7.67	3	2.77	1	2.62	2.78 ± 0.12
	3	8.38	4	3.00	3		
	4	7.51	1	3.05	4		

Table 5.8: **CIFAR-10 Benchmark: Broad Reproducibility of Random Search WS** We report the stage 3 performance of the final architecture from 6 independent runs of random search with weight-sharing.
† This run was performed using the DARTS code before we corrected for non-determinism (see Section 5.7.2).

Test Error Across 10 Seeds						
Run 1	Run 2†	Run 3	Run 4	Run 5	Run 6	**Average**
2.85 ± 0.08	2.86 ± 0.09	2.88 ± 0.10	2.95 ± 0.09	2.98 ± 0.12	3.00 ± 0.19	2.92

Delving into the intermediate results, we compare the performance of the best architectures across trials from our independent run of DARTS after training each from scratch for 100 epochs and 600 epochs (see Table 5.7). We see that the ranking is unstable between 100 epochs and 600 epochs, which motivated our strategy of training the final architectures across trials to 600 epochs in order to select the best architecture for final evaluation across 10 seeds. This suggests we should be cautious when using noisy signals for the performance of different architectures, especially since architecture search is conducted for DARTS and Random (5) for only 50 and 150 epochs respectively.

Finally, we investigate the variance of random search with weight-sharing with 5 additional runs as shown in Table 5.8. The stage (3) evaluation of the best architecture for these 5 additional runs reveal that 2 out of 5 achieve similar performance as Run 1, while the 3 remainder underperform but still reach a better test error than ASHA. These broad reproducibility results show that random search with weight-sharing has high variance between runs, which is not surprising given the change in intermediate rankings that we observed for DARTS.

5.5.3 Computational Cost

As mentioned in Section 5.2, there is a trade off between computational cost and the quality of the signal that we get per architecture that we evaluate. To get a better sense of the tradeoff, we estimate the per architecture computational cost of different methods considered in our experiments, noting that these methods differ in per architecture cost by at least an order-of-magnitude as we move from expensive to cheaper evaluation methods:

1. **Full training:** The random search baselines considered by Liu et al. [2019] cost 0.5 GPU days per architecture for the PTB benchmark and 4 GPU hours per architecture for the CIFAR-10 benchmark.
2. **Partial training:** The amortized cost of ASHA is 9 minutes per architecture for the PTB benchmark and 19 minutes per architecture for the CIFAR-10 benchmark.
3. **Weight-sharing:** It is difficult to quantify the equivalent number of architectures evaluated by DARTS and random search with weight-sharing. For DARTS, the final architecture is taken to be the highest weighted operation for each architectural decision, but it is unclear how much information is provided by the gradient updates to the architecture mixture weights during architecture search. For random search with weight sharing, although we evaluate a given number of architectures using the shared weights, as stated in Section 5.4, this is a tunable meta-hyperparameter and the quality of the performance estimates we receive can be noisy.[3] Nonetheless, to provide a rough estimate of the cost per architecture for random search with weight-sharing, we calculate the amortized cost by dividing the total search cost by the number of architectures evaluated using the shared weights. Hence, the amortized cost for random search with weight-sharing is 0.2 minutes per architecture for the PTB benchmark and 0.8 minutes per architecture for the CIFAR-10 benchmark.

Despite the apparent computational savings from weight-sharing methods, without more robustness and transparency, it is difficult to ascertain whether the total cost of applying weight-sharing methods to NAS problems warrants their broad application. In particular, the total costs of ProxylessNAS, ENAS, and SNAS are likely much higher than that reported in Table 5.2 and Table 5.5 since, as we saw with DARTS and random search with weight-sharing, multiple trials are needed due to sensitivity to initialization. Additionally, while we lightly tuned the meta-hyperparameter settings, we used DARTS' settings as our starting point and it is unclear whether the settings they use required considerable tuning. In contrast, we were able to achieve nearly competitive performance with the default settings of ASHA using roughly the same total computation as that needed by DARTS and random search with weight-sharing.

5.5.4 Available Code

Unless otherwise noted, our results are exactly reproducible from architecture search to final evaluation using the code available at https://github.com/liamcli/randomNAS_release. The code we use for random search with weight-sharing on both benchmarks is deterministic con-

[3] In principle, the equivalent number of architectures evaluated can be calculated by applying an oracle CDF of ground truth performance over randomly sampled architectures to the performance of the architectures found by the shared weights.

ditioned on a fixed random seed. We provide the final architectures used for each of the trials shown in the tables above, as well as the random seeds used to find those architectures. In addition, we perform no additional hyperparameter tuning for final architectures and only tune the meta-hyperparameters according to the discussion in the text itself. We also provide code, final architectures, and random seeds used for our experiments using ASHA. However, we note that there is one uncontrolled source of randomness in our ASHA experiments—in the distributed setting, the asynchronous nature of the algorithm means that the results depend on the order in which different architectures finish (partially) training. Lastly, our experiments were conducted using Tesla P100 and V100 GPUs on Google Cloud. We convert GPU time on V100 to equivalent time on P100 by applying a multiple of 1.5.

5.6 Discussion on Reproducibility

We conclude by summarizing our results and proposing suggestions to push the field forward and foster broader adoption of NAS methods. Since this work, we have seen recent papers adopt some of our suggestions presented below [Carlucci et al., 2019, Cho et al., 2019, Dong and Yang, 2020, Xu et al., 2020, Yu et al., 2020, Zela et al., 2020b].

1. **Better baselines that accurately quantify the performance gains of NAS methods.** The performance of random search with early-stopping evaluated in Section 5.5 reveals a surprisingly small performance gap between leading general-purpose hyperparameter optimization methods and specialized methods tailored for NAS. As shown in Chapter 2.4, random search is a difficult baseline to beat for traditional hyperparameter tuning tasks as well. For these benchmarks, an informative measure of the performance of a novel algorithm is its 'multiple of random search,' i.e., how much more compute would random search need to achieve similar performance. An analogous baseline could be useful for NAS, where the impact of a novel NAS method can be quantified in terms of a multiplicative speedup relative to a standard hyperparameter optimization method such as random search with early-stopping.
2. **Ablation studies that isolate the impact of individual NAS components.** Our head-to-head experimental evaluation of two variants of random search (with early stopping and with weight-sharing) allows us to pinpoint the performance gains associated with the cheaper weight-sharing evaluation scheme. In contrast, the fact that random search with weight-sharing is comparable in performance to competitive NAS methods calls into question the necessity of the auxiliary network used by GHN and the complicated algorithmic components employed by ENAS, SNAS, and DARTS. Relatedly, while ProxylessNAS achieves better average test error on CIFAR-10 than random search with weight-sharing, it is unclear to what degree these performance gains are attributable to the search space, search method, and/or proxyless shared-weights evaluation method. To promote scientific progress, we believe that ablation studies should be conducted to answer these questions in isolation.
3. **Reproducible results that engender confidence and foster scientific progress.** Reproducibility is a core tenet of scientific progress and crucial to promoting wider adoption

of NAS methods. Following standard practice in traditional hyperparameter optimization [Feurer et al., 2015a, Kandasamy et al., 2017, Klein et al., 2017a], empirical results in this thesis are reported over over multiple independent experimental runs. In contrast, as we discuss Section 5.2, results for NAS methods are often reported over a single experimental run [Cai et al., 2019, Pham et al., 2018, Xie et al., 2019, Zhang et al., 2019], without exact reproducibility. This is a consequence of the steep time and computational cost required to perform NAS experiments, e.g., generating the experiments reported in this chapter alone required several months of wall-clock time and tens of thousands of dollars. However, in order to adequately differentiate between various methods, results need to be reported over several independent experimental runs, especially given the nature of the extremal statistics that are being reported. Consequently, we conclude that either significantly more computational resources need to be devoted to evaluating NAS methods and/or more computationally tractable benchmarks need to be developed to lower the barrier for performing adequate empirical evaluations. Relatedly, we feel it is imperative to evaluate the merits of NAS methods not only on their accuracy, but also on their robustness across these independent runs.

5.7 Experimental Details

In this section, we provide additional detail for the experiments in Section 5.5.1 and Section 5.5.2.

5.7.1 PTB Benchmark

Architecture Operations. In stage (1), DARTS trained the shared weights network with the zero operation included in the list of considered operations but removed the zero operation when selecting the final architecture to evaluate in stages (2) and (3). For our random search with weight-sharing, we decided to exclude the zero operation for both search and evaluation.

Stage 3 Procedure. For stage (3) evaluation, we follow the ArXiv version of DARTS [Liu et al., 2018c], which reported two sets of results, one after training for 1600 epochs and another fine tuned result after training for an additional 1000 epochs. In the ICLR version, Liu et al. [2019] simply say they trained the final network to convergence. We trained for another 1000 epochs for a total of 3600 epochs to approximate training to convergence.

5.7.2 CIFAR-10 Benchmark

Architecture Operations. In stage (1), DARTS trained the shared weights network with the zero operation included in the list of considered operations but removed the zero operation when selecting the final architecture to evaluate in stages (2) and (3). For our random search with weight-sharing, we decided to *include* the zero operation for both search and evaluation.

Stage 1 Procedure. For random search with weight-sharing, after the shared weights are fully trained, we evaluate randomly sampled architectures using the shared weights and select the best one for stage (2) evaluation. Due to the higher cost of evaluating on the full validation set, we evaluate each architecture using 10 minibatches instead. We split the total number of

architectures to be evaluated into sets of 1000. For each 1000, we select the best 10 according the cheap evaluation on part of the validation set and evaluate on the full validation set. Then we select the top architecture across all sets of 1000 for stage (2) evaluation.

Reproducibility. The code released by Liu et al. [2019] did not produce deterministic results for the CNN benchmark due to non-determinism in CuDNN and in data loading. We removed the non-deterministic behavior in CuDNN by setting

$\text{cudnn.benchmark} = \text{False}$
$\text{cudnn.deterministic} = \text{True}$
$\text{cudnn.enabled}=\text{True}$

Note that this only disables the non-deterministic functions in CuDNN and does not adversely affect training time as much as turning off CuDNN completely. We fix additional non-determinism from data loading by setting the seed for the random package in addition to numpy.random and pytorch seed and turning off multiple threads for data loading.

We ran ASHA and one set of trials for Random Search (5) with weight-sharing using the non-deterministic code before fixing the seeding to get deterministic results; all other settings for random search with weight-sharing are deterministic. Hence, the result for ASHA does not satisfy exact reproduciblity due to non-deterministic training and asynchronous updates.

Chapter 6

Geometry-Aware Optimization for Gradient-Based NAS

In the previous chapter, we established random search with weight-sharing (RSWS) as a strong baselines for neural architecture search that achieves competitive performance relative to DARTS [Liu et al., 2019] although with higher variance. More recent gradient-based weight-sharing NAS approaches have improved significantly upon DARTS [Chen et al., 2019, Nayman et al., 2019, Xu et al., 2020], widening the lead over RSWS. In this chapter, we study the optimization problem associated with gradient-based weight-sharing methods. Our efforts culminate in a framework for analyzing the convergence rates of gradient-based methods. We use our framework to design geometry-aware gradient-based NAS methods with faster convergence and better empirical performance.

6.1 Introduction

As is the case for the weight-sharing methods we compared to in the previous Chapter [Liu et al., 2019, Pham et al., 2018], weight-sharing is often combined with a continuous relaxation of the discrete search space to allow cheap gradient updates to architecture hyperparameters [Akimoto et al., 2019, Cai et al., 2019, Dong and Yang, 2019, Laube and Zell, 2019, Liu et al., 2018a, Nayman et al., 2019, Noy et al., 2019, Xie et al., 2019, Xu et al., 2020, Yang et al., 2019, Zheng et al., 2019], enabling the use of popular optimizers such as stochastic gradient descent (SGD) or Adam [Kingma and Ba, 2015]. However, despite some empirical success, weight-sharing remains poorly understood and subsequent improvements to the search process continue to be largely heuristic [Carlucci et al., 2019, Li et al., 2019b, Liang et al., 2019, Nayman et al., 2019, Noy et al., 2019, Xu et al., 2020, Zela et al., 2020a]. Furthermore, on recent computer vision benchmarks weight-sharing methods can still be outperformed by traditional hyperparameter optimization algorithms given a similar time-budget [Dong and Yang, 2020]. Thus, motivated by the challenge of developing simple, principled, and efficient NAS methods that achieve state-of-the-art performance in computer vision and beyond, we make progress on the following question: *how can we design and analyze fast and accurate methods for NAS with weight-sharing?*

We start from the fact that methods that use weight-sharing subsume architecture hyperpa-

rameters as another set of learned parameters of the shared-weights network, in effect extending the class of functions being learned. Our motivating question thus reduces to the following: *how can we design and analyze fast and accurate methods for empirical risk minimization (ERM) over this extended class?* While this is a well-studied problem in ML and procedures such as SGD have been remarkably effective for unconstrained optimization of standard neural networks [Wilson et al., 2017], it is far from clear that this will naturally extend to optimizing architecture parameters, which can lie in constrained, non-Euclidean domains.

In this chapter, we develop a more principled, *geometry-aware* framework for designing and analyzing both continuous relaxations and optimizers for NAS with weight-sharing. Drawing upon the mirror descent meta-algorithm [Beck and Teboulle, 2003, Nemirovski and Yudin, 1983], we give polynomial-time convergence guarantees when optimizing the weight-sharing objective for a broad class of gradient-based methods that connect to the underlying problem structure. Notably, this allows us to reason about (1) the speed of both new and existing NAS methods, and (2) how to construct continuous relaxations of discrete search spaces. We use this framework to design state-of-the-art NAS methods, as detailed below in our contributions:

1. We study the optimization of NAS with weight-sharing as ERM over a structured function class in which architectural decisions are treated as learned parameters rather than hyperparameters. We prove polynomial-time stationary-point convergence guarantees for block-stochastic mirror descent over a continuous relaxation of this objective and show that gradient-based methods such as ENAS [Pham et al., 2018] and DARTS [Liu et al., 2019] can be viewed as variants of this meta-algorithm. To the best of our knowledge these are the first finite-time convergence guarantees for any gradient-based NAS method.
2. Crucially, these convergence rates depend on the underlying architecture parameterization. We thus aim to improve upon existing algorithms by (1) using different continuous relaxations of the search space and (2) applying mirror descent procedures adapted to these new geometries. Our first instantiation is a simple modification, applicable to many methods including first-order DARTS, that constrains architecture parameters to the simplex and updates them using exponentiated gradient, a variant of mirror descent that converges quickly over the simplex and enjoys favorable sparsity properties.
3. We use this **Geometry-Aware Exponentiated Algorithm (GAEA)** to improve state-of-the-art methods on three NAS benchmarks for computer vision:
   (a) On the DARTS search space [Liu et al., 2019], GAEA exceeds the best published results by finding an architecture achieving 2.50% test error on CIFAR-10 and, separately, an architecture achieving 24.0% test error on ImageNet.
   (b) On the three search spaces in the NASBench-1Shot1 benchmark [Zela et al., 2020b], GAEA matches or slightly exceeds the performance of the best reported method for each search space.
   (c) On the three datasets within the NAS-Bench-201 benchmark [Dong and Yang, 2020], GAEA exceeds the performance of both weight-sharing and traditional hyperparameter approaches and identifies architectures with near optimal performance on each dataset.

6.1.1 Related Work

Most analyses of optimization problems associated with NAS show monotonic improvement [Akimoto et al., 2019], asymptotic guarantees [Yao et al., 2019], or bounds on auxiliary quantities, e.g. regret, that are not well-connected to the objective [Carlucci et al., 2019, Nayman et al., 2019, Noy et al., 2019]. In contrast, we prove polynomial-time stationary-point convergence on the single-level (ERM) objective for NAS with weight-sharing, which is effective in practice. Furthermore, due to the generality of mirror descent, our results can applied to un-analyzed methods such as first-order DARTS [Liu et al., 2019] and ENAS [Pham et al., 2018]. Finally, we note that a variant of GAEA that modifies first-order DARTS is related to XNAS [Nayman et al., 2019], whose update also involves an exponentiated gradient step; however, GAEA is simpler, easier to implement, and reproducible.[1] Furthermore, the regret guarantees for XNAS do not relate to any meaningful performance measure for NAS such as speed or accuracy, whereas we guarantee convergence on the ERM objective.

Our results draw upon the extensive literature on first-order optimization [Beck, 2017] in particular on the mirror descent meta-algorithm [Beck and Teboulle, 2003, Nemirovski and Yudin, 1983]. Such methods have been successfully used in a wide array of applications in machine learning [Abernethy et al., 2008, Daskalakis et al., 2018, Mahadevan and Liu, 2012], computer vision [Ajanthan et al., 2019, Kunapuli and Shavlik, 2012, Luong et al., 2012], and theoretical computer science [Arora et al., 2012, Bubeck et al., 2018, Rakhlin and Sridharan, 2013]. We extend recent nonconvex convergence guarantees for stochastic mirror descent [Zhang and He, 2018] to handle alternating descent schemes; while there exist related results in this setting [Dang and Lan, 2015] the associated guarantees do not hold for the algorithms we propose.

6.2 The Weight-Sharing Optimization Problem

In standard supervised ML we are given a dataset T of labeled pairs (x, y) drawn from a distribution $\mathcal{D}$ over input space X and output space Y. The goal is to use this data to search some parameterized function class H for a function $h_{\mathbf{w}} : X \mapsto Y$ parameterized by $\mathbf{w} \in \mathbb{R}^d$ that has low expected test loss $\ell(h_{\mathbf{w}}(x), y)$ when using x to predict the associated y over unseen sample drawn from the same distribution, as measured by some loss function $\ell : Y \times Y \mapsto [0, \infty)$. A common way to this is (regularized) ERM, which outputs $\mathbf{w} \in \mathbb{R}^d$ such that $h_{\mathbf{w}} \in H$ has the smallest average (regularized) loss over the training sample T.

6.2.1 Single-Level Neural Architecture Search

As a subset of hyperparameter optimization, NAS can be viewed as hyperparameter optimization on top of ERM, with each architecture $a \in \mathcal{A}$ corresponding to a function class H_a to be selected via validation data. However, unlike hyperparameters such as regularization and step-size, these

[1] Code provided for XNAS (https://github.com/NivNayman/XNAS) does not implement the search algorithm and, as with previous efforts [Li et al., 2019b, OpenReview Forum], we were unable to reproduce the reported search results after correspondence with the authors. The best architecture reported by XNAS achieves an average test error of 2.70% under the DARTS evaluation protocol, while GAEA achieves 2.50%. We provide further discussion and experimental comparison with XNAS in Section 6.5.5.

architectural hyperparameters directly affect the mapping from X to Y, so one can forgo holding out validation data and find the best function in $H_{\mathcal{A}} = \bigcup_{a \in \mathcal{A}} H_a = \{h_{\mathbf{w},a} : X \mapsto Y, \mathbf{w} \in \mathbb{R}^d, a \in \mathcal{A}\}$, or more formally (for some regularization parameter $\lambda \geq 0$) solve

$$\min_{\mathbf{w} \in \mathbb{R}^d, a \in \mathcal{A}} \quad \lambda \|\mathbf{w}\|_2^2 \quad + \quad \frac{1}{|T|} \sum_{(x,y) \in T} \ell(h_{\mathbf{w},a}(x), y) \tag{6.1}$$

Under this formulation, the use of weight-sharing is perhaps not surprising: both the network parameters $\mathbf{w}$ and architecture decisions a are part of the same optimization domain. This is made even more clear in the common approach where the discrete decision space is relaxed into a continuous one and the optimization becomes one over some subset Θ of a finite-dimensional real vector space:

$$\min_{\mathbf{w} \in \mathbb{R}^d, \theta \in \Theta} \quad \lambda \|\mathbf{w}\|_2^2 \quad + \quad \frac{1}{|T|} \sum_{(x,y) \in T} \ell(h_{\mathbf{w},\theta}(x), y) \tag{6.2}$$

Here $h_{\mathbf{w},\theta} : X \mapsto Y$ is determined by some superposition of architectures, such as a θ-weighted mixture on each edge used by DARTS [Liu et al., 2019] or a randomized function with θ-parameterized distribution employed by GDAS [Dong and Yang, 2019]; this is detailed in Section 6.2.2. Note that in order for the relaxation Eq. (6.2) to be a useful surrogate for Eq. (6.1), we need a (possibly randomized) mapping from a feasible point $\theta \in \Theta$ to $a \in \mathcal{A}$; the associated error or variance of this mapping can be catastrophic, as witnessed by the performance of DARTS [Liu et al., 2019] on NAS-Bench-201 [Dong and Yang, 2020].

This analysis reduces our motivating question from the introduction to the following: *how do we design a relaxation Θ of architecture space $\mathcal{A}$ that admits an optimizer that is simultaneously fast at solving Eq. (6.2) and leads to architecture parameters $\theta \in \Theta$ that can be mapped to some $a \in \mathcal{A}$ with low test error?* In the sequel, we provide an answer via the mirror descent framework and demonstrate its theoretical and empirical success.

Before proceeding, we note that many weight-sharing approaches [Dong and Yang, 2019, Liu et al., 2019, Pham et al., 2018] attempt to solve a *bilevel* problem rather than ERM by using a validation set V:

$$\begin{aligned} &\min_{\mathbf{w} \in \mathbb{R}^d, \theta \in \Theta} \quad \frac{1}{|V|} \sum_{(x,y) \in V} \ell(h_{\mathbf{w},\theta}(x), y) \\ \text{s.t.} \quad &\mathbf{w} \in \arg\min_{\mathbf{u} \in \mathbb{R}^d} \quad \lambda \|\mathbf{u}\|_2^2 \quad + \quad \frac{1}{|T|} \sum_{(x,y) \in T} \ell(h_{\mathbf{u},\theta}(x), y) \end{aligned} \tag{6.3}$$

Here our theory does not hold,[2] but we believe that our guarantees for block-stochastic methods–the heuristic of choice for modern bilevel problems [Franceschi et al., 2018, Liu et al., 2019]–are still useful for algorithm design. Furthermore, past work has sometimes used the single-level approach [Li et al., 2019b, Xie et al., 2019] and we find it very effective on some benchmarks.

[2]Known guarantees for gradient-based bilevel optimization are asymptotic and rely on strong assumptions such as uniqueness of the minimum of the inner optimization [Franceschi et al., 2018].

6.2.2 Neural Architecture Search with Weight-Sharing

Recall from Chapter 5.3 the cell-based search spaces considered for NAS, where architectures can be represented as DAGs. Let E be the set of possible edges and O the set of possible operations. Then, $\mathcal{A} \subset \{0,1\}^{|E|\times|O|}$ is the set of all valid architectures for encoded by edge and operation decisions. Treating both the *shared weights* $\mathbf{w} \in \mathbb{R}^d$ and architecture decisions $a \in \mathcal{A}$ as parameters, weight-sharing methods train a single network subsuming all possible functions within the search space.

Gradient-based weight-sharing methods apply continuous relaxations to the architecture space $\mathcal{A}$ in order to compute gradients in a continuous space Θ. Methods like DARTS [Liu et al., 2019] and its variants [Chen et al., 2019, Hundt et al., 2019, Laube and Zell, 2019, Liang et al., 2019, Nayman et al., 2019, Noy et al., 2019] relax the search space by considering a *mixture* of operations per edge. For example, we will consider a relaxation where the architecture space $\{0,1\}^{|E|\times|O|}$ is relaxed into $\Theta = [0,1]^{|E|\times|O|}$ with the constraint that $\sum_{o\in O} \theta_{i,j,o} = 1$, i.e., the operation weights on each edge sum to 1. The feature at node i is then $x^{(i)} = \sum_{j<i} \sum_{o\in O} \theta_{i,j,o} o(x^{(j)})$. To get a valid architecture $a \in \mathcal{A}$ from a mixture θ, rounding and pruning are typically employed after the search phase.

A different, *stochastic* approach, e.g., GDAS [Dong and Yang, 2019], instead uses Θ-parameterized distributions p_θ over $\mathcal{A}$ to sample architectures [Akimoto et al., 2019, Cai et al., 2019, Pham et al., 2018, Xie et al., 2019]; unbiased gradients w.r.t. $\theta \in \Theta$ can be computed using Monte Carlo sampling. The goal of all these relaxations is to use simple gradient-based approaches to approximately optimize Eq. (6.1) over $a \in \mathcal{A}$ by optimizing Eq. (6.2) over $\theta \in \Theta$ instead. However, both the relaxation and the optimizer critically affect the convergence speed and solution quality. We next present a principled approach for understanding both mixture and stochastic methods.

6.3 Geometry-Aware Gradient Algorithms

Recall that we want a fast algorithm that minimizes the regularized empirical risk $f(\mathbf{w},\theta) = \lambda\|\mathbf{w}\|_2^2 + \frac{1}{|T|}\sum_{(x,y)\in T} \ell(h_{\mathbf{w},\theta}(x), y)$ over shared-weights $\mathbf{w} \in \mathbb{R}^d$ and architecture parameters $\theta \in \Theta$. We assume that we can take noisy derivatives of f separately w.r.t. either $\mathbf{w}$ and θ at any point $(\mathbf{w},\theta) \in \mathbb{R}^d \times \Theta$, i.e. stochastic gradients $\tilde{\nabla}_{\mathbf{w}} f(\mathbf{w},\theta)$ or $\tilde{\nabla}_\theta f(\mathbf{w},\theta)$ satisfying $\mathbb{E}\tilde{\nabla}_{\mathbf{w}} f(\mathbf{w},\theta) = \nabla_{\mathbf{w}} f(\mathbf{w},\theta)$ or $\mathbb{E}\tilde{\nabla}_\theta f(\mathbf{w},\theta) = \nabla_\theta f(\mathbf{w},\theta)$, respectively. In practice this is done by minibatch sampling of datapoints from T. Our goal will be to make the function f small while taking as few stochastic gradients as possible, although in the nonconvex case we must settle for finding a point $(\mathbf{w},\theta) \in \mathbb{R}^d \times \Theta$ with a small gradient.

6.3.1 Background on Mirror Descent

While there are many ways of motivating, formulating, and describing mirror descent [Beck and Teboulle, 2003, Nemirovski and Yudin, 1983, Shalev-Shwartz, 2011], for simplicity we consider the proximal formulation. The starting point here is the observation that, in the unconstrained case, performing an SGD update of a point $\theta \in \Theta = \mathbb{R}^k$ by a stochastic gradient $\tilde{\nabla} f(\theta)$ with

learning rate $\eta > 0$ is equivalent to solving a regularized optimization problem:

$$\theta - \eta\tilde{\nabla} f(\theta) \quad = \quad \arg\min_{\mathbf{u}\in\mathbb{R}^d} \quad \eta\tilde{\nabla} f(\theta)\cdot\mathbf{u} \quad + \quad \frac{1}{2}\|\mathbf{u}-\theta\|_2^2 \tag{6.4}$$

Thus the update aligns the output with the gradient while regularizing for closeness to the previous point in the Euclidean norm. This works for unconstrained high-dimensional optimization, e.g. deep net training, but the choice of regularizer may be sub-optimal over a constrained space with sparse solutions.

The canonical example of when a geometry-aware algorithm gives a concrete benefit is when we are optimizing a function over the unit simplex, i.e. when $\Theta = \{\theta \in [0,1]^k : \|\theta\|_1 = 1\}$. Here, if we replace the ℓ_2-regularizer in Equation 6.4 by the relative entropy $\mathbf{u}\cdot(\log\mathbf{u} - \log\theta)$, i.e. the Kullback-Liebler (KL) divergence between $\mathbf{u}$ and θ, then we obtain the exponentiated gradient (EG) update:

$$\theta \odot \exp(-\eta\tilde{\nabla} f(\theta)) \quad \propto \quad \arg\min_{\mathbf{u}\in\Theta} \quad \eta\tilde{\nabla} f(\theta)\cdot\mathbf{u} \quad + \quad \mathbf{u}\cdot(\log\mathbf{u} - \log\theta) \tag{6.5}$$

A classical result states that EG over the k-dimensional simplex requires only $\mathcal{O}(\log k/\varepsilon^2)$ iterations to achieve a function value ε-away from optimal [Beck and Teboulle, 2003, Theorem 5.1], compared to the $\mathcal{O}(k/\varepsilon^2)$ guarantee for SGD. This nearly dimension-independent iteration complexity is achieved by choosing a regularizer–the KL divergence–well suited to the underlying geometry–the simplex.

In full generality, mirror descent algorithms are specified by a distance-generating function (DGF) ϕ on Θ, which should be strongly-convex w.r.t. some suitable norm $\|\cdot\|$ in order to provide geometry-aware regularization. The DGF ϕ induces a *Bregman divergence* $D_\phi(\mathbf{u}||\mathbf{v}) = \phi(\mathbf{u}) - \phi(\mathbf{v}) - \nabla\phi(\mathbf{v})\cdot(\mathbf{u}-\mathbf{v})$ [Bregman, 1967], a notion of distance on Θ that acts as a regularizer in the mirror descent update:

$$\arg\min_{u\in\Theta} \quad \eta\tilde{\nabla} f(\theta)\cdot\mathbf{u} \quad + \quad D_\phi(\mathbf{u}||\theta) \tag{6.6}$$

For example, to recover the SGD update Eq. (6.4) set $\phi(\mathbf{u}) = \frac{1}{2}\|\mathbf{u}\|_2^2$, which is strongly-convex w.r.t. the Euclidean norm, and to get the EG update Eq. (6.5) set $\phi(\mathbf{u}) = \mathbf{u}\cdot\log\mathbf{u}$, which is strongly-convex w.r.t. the ℓ_1-norm. As we will see, regularization w.r.t. the latter norm is useful for obtaining sparse solutions with few iterations.

6.3.2 Block-Stochastic Mirror Descent

In the previous section we saw how mirror descent can yield fast convergence over certain geometries such as the simplex. However, in NAS with weight-sharing we are often interested in optimizing over a hybrid geometry, e.g. one containing both the shared weights in an unconstrained Euclidean space and the architecture parameters in a potentially non-Euclidean domain. Thus we focus on the optimization of a non-convex function over two blocks: shared weights $\mathbf{w} \in \mathbb{R}^d$ and architecture parameters $\theta \in \Theta$, where Θ is associated with a DGF ϕ that is strongly-convex w.r.t. some norm $\|\cdot\|$. In NAS a common approach is to perform alternating gradient steps on each

Algorithm 4: Block-stochastic mirror descent optimization of a function $f : \mathbb{R}^d \times \Theta \mapsto \mathbb{R}$, where Θ has an associated DGF ϕ.

Input: initialization $(\mathbf{w}^{(1)}, \theta^{(1)}) \in \mathbb{R}^d \times \Theta$, no. of steps $T \geq 1$, step-size $\eta > 0$

for iteration $t = 1, \dots, T$ **do**
  sample $b_t \sim \text{Unif}\{\mathbf{w}, \theta\}$ // randomly select update block
  if block $b_t = \mathbf{w}$ **then**
    $\mathbf{w}^{(t+1)} \leftarrow \mathbf{w}^{(t)} - \eta \tilde{\nabla}_{\mathbf{w}} f(\mathbf{w}^{(t)}, \theta^{(t)})$ // SGD update to shared weights
    $\theta^{(t+1)} \leftarrow \theta^{(t)}$ // no update to architecture parameters
  else
    $\mathbf{w}^{(t+1)} \leftarrow \mathbf{w}^{(t)}$ // no update to shared weights
    $\theta^{(t+1)} \leftarrow \arg\min_{\mathbf{u} \in \Theta} \eta \tilde{\nabla}_{\theta} f(\mathbf{w}^{(t)}, \theta^{(t)}) \cdot \mathbf{u} + D_{\phi}(\mathbf{u} || \theta^{(t)})$
    // mirror descent update to architecture parameters
  end
end

Output: $(\mathbf{w}^{(r)}, \theta^{(r)})$ for $r \sim \text{Unif}\{1, \dots, T\}$ // return random iterate

domain; for example, both ENAS [Pham et al., 2018] and first-order DARTS [Liu et al., 2019] alternate between SGD updates to the shared weights and Adam [Kingma and Ba, 2015] updates to the architecture parameters. This type of approach is encapsulated in the block-stochastic algorithm described in Algorithm 4, which at each iteration chooses one block at random to update using mirror descent (recall that SGD is also a variant of mirror descent) and after T such steps returns a random iterate. Note that Algorithm 4 subsumes both ENAS and first-order DARTS if SGD is used over the architecture parameters instead of Adam, with the following mild caveats: in practice blocks are picked cyclically rather than randomly and the algorithm returns the last iterate rather than a random one.

In Section 6.4 we prove that, under some assumptions on the objective f, the output $(\mathbf{w}^{(r)}, \theta^{(r)})$ of Algorithm 4 is an ε-stationary-point of f if we run at least $T = \mathcal{O}(G_{\mathbf{w}}^2 + G_{\theta}^2)/\varepsilon^2$ iterations; here $\varepsilon > 0$ measures the expected non-stationarity, $G_{\mathbf{w}}^2$ upper-bounds the expected squared ℓ_2-norm of the stochastic gradient $\tilde{\nabla}_{\mathbf{w}}$ w.r.t. the shared weights, and G_{θ}^2 upper-bounds the expected squared magnitude of the stochastic gradient $\tilde{\nabla}_{\theta}$ w.r.t. the architecture parameters, as measured by the *dual norm* $\|\cdot\|_*$ of $\|\cdot\|$.

This last term G_{θ} is key to understanding the usefulness of mirror descent, as it is the only term that is affected by our choice of DGF ϕ, which is strongly-convex w.r.t. $\|\cdot\|$ (recall from Section 6.3.1 that a variant of mirror descent is defined by its DGF). In the case of SGD, $\phi(\mathbf{u}) = \frac{1}{2}\|\mathbf{u}\|_2^2$ is strongly-convex w.r.t. the ℓ_2-norm, which is its own dual norm; this is why $G_{\mathbf{w}}^2$ is defined via the ℓ_2-norm of the shared weights gradient. However, in the case of exponentiated gradient $\phi(\mathbf{u}) = \mathbf{u} \cdot \log \mathbf{u}$ is strongly-convex w.r.t. the ℓ_1-norm, whose dual norm is the ℓ_∞-norm. Since the ℓ_2-norm of a k-dimensional vector can be a factor $\sqrt{k}$ larger than its ℓ_∞-norm, and is never smaller, picking a DGF that is strongly-convex w.r.t. the ℓ_1-norm can lead to much better bound on G_{θ} and thus in-turn on the number of iterations needed to reach a stationary point.

6.3.3 A Single-Level Analysis of ENAS and DARTS

Before using this analysis for new NAS algorithms, we first apply it to existing methods, specifically ENAS [Pham et al., 2018] and DARTS [Liu et al., 2019]. For simplicity, we assume all architectures in the relaxed search space are γ-smooth, which excludes components such as ReLU. However, the objective are still smooth when convolved with a Gaussian, which is equivalent to adding noise to every gradient; thus given the noisiness of SGD training we believe the following analysis is still informative [Kleinberg et al., 2018].

ENAS continuously relaxes $\mathcal{A}$ via a neural controller that samples architectures $a \in \mathcal{A}$, so $\Theta = \mathbb{R}^{\mathcal{O}(h^2)}$, where h is the number of hidden units. The controller is trained with Monte Carlo gradients. On the other hand, first-order DARTS uses a mixture relaxation similar to the one in Section 6.2.2 but using a softmax instead of constraining parameters to the simplex. Thus $\Theta = \mathbb{R}^{|E| \times |O|}$ for E the set of learnable edges and O the set of possible operations. If we assume that both algorithms use SGD for the architecture parameters then to compare them we are interested in their respective values of G_θ, which we will refer to as G_{ENAS} and G_{DARTS}. Before proceeding, we note again that our theory holds only for the single-level objective and when using SGD as the architecture optimizer, whereas both algorithms specify the bilevel objective and Adam [Kingma and Ba, 2015], respectively.

At a very high level, the Monte Carlo gradients used by ENAS are known to be high-variance, so G_{ENAS} may be much larger than G_{DARTS}, yielding faster convergence for DARTS, which is reflected in practice [Liu et al., 2019]. We can also do a simple low-level analysis under the assumption that all architecture gradients are bounded entry-wise, i.e. in ℓ_∞-norm, by some constant; then since the squared ℓ_2-norm is bounded by the product of the dimension and the squared ℓ_∞-norm we have $G^2_{\text{ENAS}} = \mathcal{O}(h^2)$ while $G^2_{\text{DARTS}} = \mathcal{O}(|E||O|)$. Since ENAS uses a hidden state size of $h = 100$ and the DARTS search space has $|E| = 14$ edges and $|O| = 7$ operations, this also points to DARTS needing fewer iterations to converge.

6.3.4 GAEA: A Geometry-Aware Exponentiated Algorithm

Equipped with a general understanding block-stochastic optimizers for NAS, we now turn to designing new methods. Our goal is to pick architecture parameterizations and algorithms that (a) have fast convergence in appropriate geometries and (b) encourage favorable properties in the learned parameters. To do so, we return to the exponentiated gradient (EG) update discussed in Section 6.3.1 and propose **GAEA**, a Geometry-Aware Exponentiated Algorithm in which operation weights on each edge are constrained to the simplex and trained using EG, i.e., mirror descent with entropic DGF $\phi(\mathbf{u}) = \mathbf{u} \cdot \log \mathbf{u}$; as in DARTS [Liu et al., 2019], the shared weights are trained using SGD. A simple but principled modification, GAEA obtains state-of-the-art performance on several NAS benchmarks in computer vision.

Here we describe the EG update for the mixture relaxation, although as shown in Section 6.6, it is straightforward to use GAEA to modify a stochastic approach such as GDAS [Dong and Yang, 2019]. As in Section 6.2.2 let Θ be the product set of $|E|$ simplices, each corresponding to an edge $(i, j) \in E$; thus $\theta_{i,j,o} \in \Theta$ corresponds to the weight placed on operation $o \in O$ for edge (i, j). Given a stochastic gradient $\tilde{\nabla}_\theta f(\mathbf{w}^{(t)}, \theta^{(t)})$ and step-size $\eta > 0$, the GAEA update has two

steps:

$$\begin{aligned}\tilde{\theta}^{(t+1)} &\leftarrow \theta^{(t)} \odot \exp\left(-\eta \tilde{\nabla}_\theta f(\mathbf{w}^{(t)}, \theta^{(t)})\right) && \text{(multiplicative update)} \\ \theta_{i,j,o}^{(t+1)} &\leftarrow \frac{\tilde{\theta}_{i,j,o}^{(t+1)}}{\sum_{o' \in O} \tilde{\theta}_{i,j,o'}^{(t+1)}} \;\; \forall\, o \in O,\; \forall\, (i,j) \in E && \text{(simplex projection)}\end{aligned} \tag{6.7}$$

This is equivalent to mirror descent Eq. (6.6) with $\phi(\theta) = \sum_{(i,j)\in E} \sum_{o \in O} \theta_{i,j,o} \log \theta_{i,j,o}$, which is strongly-convex w.r.t. the norm $\|\cdot\|_1/\sqrt{|E|}$ over the product of $|E|$ simplices of size $|O|$. The dual norm is $\sqrt{|E|}\|\cdot\|_\infty$, so if as before $G_{\mathbf{w}}$ bounds the gradient w.r.t. the shared weights and we have an entry-wise bound on the architecture gradient then GAEA needs $\mathcal{O}(G_{\mathbf{w}}^2 + |E|)/\varepsilon^2$ iterations to reach ε-stationary-point. This can be up to a factor $|O|$ improvement over just using SGD, e.g. the DARTS bound in Section 6.3.3. In addition to fast convergence, GAEA encourages sparsity in the architecture weights by using a multiplicative update over the simplex and not an additive update over $\mathbb{R}^d$, reducing the effect of post-hoc pruning or sampling when obtaining a valid architecture $a \in \mathcal{A}$ from θ. Obtaining sparse final architecture parameters is critical for good performance, both for the mixture relaxation, where it alleviates the effect of overfitting, and for the stochastic relaxation, where it reduces noise when sampling architectures.

6.4 Convergence of Block-Stochastic Mirror Descent

Here we give a self-contained formalization of the Algorithm 4 guarantees governing the analysis in Section 6.3. We start with the following regularity assumptions:

Assumption 1. *Suppose ϕ is strongly-convex w.r.t. some norm $\|\cdot\|$ on a convex set Θ and the objective function $f : \mathbb{R}^d \times \Theta \mapsto [0, \infty)$ satisfies the following:*

1. *$f(\mathbf{w}, \theta) + \frac{\gamma}{2}\|\mathbf{w}\|_2^2 + \gamma\phi(\theta)$ is convex on $\mathbb{R}^d \times \Theta$ for some $\gamma > 0$.*
2. *$\mathbb{E}\|\tilde{\nabla}_{\mathbf{w}} f(\mathbf{w}, \theta)\|_2^2 \le G_{\mathbf{w}}^2$ and $\mathbb{E}\|\tilde{\nabla}_\theta f(\mathbf{w}, \theta)\|_*^2 \le G_\theta^2$ for some $G_{\mathbf{w}}, G_\theta \ge 0$.*

The first condition, *γ-relatively-weak-convexity* [Zhang and He, 2018], allows all γ-smooth nonconvex functions and certain non-smooth ones. The second bounds the expected stochastic gradient on each block. We also require a convergence measure; in the unconstrained case we could just use the gradient norm, but for constrained Θ the gradient may never be zero. Thus we use the *Bregman stationarity* [Zhang and He, 2018, Equation 2.11], which uses the proximal operator prox_λ to encode non-stationarity:

$$\Delta_\lambda(\mathbf{w}, \theta) = \frac{D_{2,\phi}(\mathbf{w}, \theta || \,\mathrm{prox}_\lambda(\mathbf{w}, \theta)) + D_{2,\phi}(\mathrm{prox}_\lambda(\mathbf{w}, \theta) || \mathbf{w}, \theta)}{\lambda^2} \tag{6.8}$$

Here $\lambda = \frac{1}{2\gamma}$, $D_{2,\phi}$ is the Bregman divergence of $\frac{1}{2}\|\mathbf{w}\|_2^2 + \phi(\theta)$, and the proximal operator $\mathrm{prox}_\lambda(\mathbf{w}, \theta) = \arg\min_{\mathbf{u} \in \mathbb{R}^d \times \Theta} \lambda f(\mathbf{u}) + D_{2,\phi}(\mathbf{u} || \mathbf{w}, \theta)$; note that the dependence of the measure on γ is standard [Dang and Lan, 2015, Zhang and He, 2018]. Roughly speaking, the Bregman stationarity is $\mathcal{O}(\varepsilon)$ if $(\mathbf{w}, \theta)$ is $\mathcal{O}(\varepsilon)$-close to an $\mathcal{O}(\varepsilon)$-approximate stationary-point [Davis and Drusvyatskiy, 2019]; in the smooth case over Euclidean geometries, i.e. SGD, if $\Delta_{\frac{1}{2\gamma}} \le \varepsilon$ then we have an $O(\varepsilon)$-bound on the squared gradient norm, recovering the standard definition of ε-stationarity.

Theorem 1. *Let $F = f(\mathbf{w}^{(1)}, \theta^{(1)})$ be the value of f at initialization. Under Assumption 1, if we run Algorithm 4 for $T = \frac{16\gamma F}{\varepsilon^2}(G_{\mathbf{w}}^2 + G_\theta^2)$ iterations with step-size $\eta = \sqrt{\frac{4F}{\gamma(G_{\mathbf{w}}^2+G_\theta^2)T}}$ then $\mathbb{E}\Delta_{\frac{1}{2\gamma}}(\mathbf{w}^{(r)}, \theta^{(r)}) \leq \varepsilon$. Here the expectation is over the randomness of the algorithm and over that of the stochastic gradients.*

The proof in Section 6.6 follows from a single-block analysis of mirror descent [Zhang and He, 2018, Theorem 3.1] and in fact holds for the general case of any finite number of blocks associated to any set of strongly-convex DGFs. Although there are prior results for the multi-block case [Dang and Lan, 2015], they do not hold for nonsmooth Bregman divergences such as the KL divergence, which we need for exponentiated gradient.

6.5 Experiments

We evaluate GAEA, our geometry-aware NAS method, on three different computer vision benchmarks: the CNN search space from DARTS [Liu et al., 2019], NAS-Bench-1Shot1 [Zela et al., 2020b], and NAS-Bench-201 [Dong and Yang, 2020]. Note that in Section 6.3.4 we described GAEA as a re-parameterization and modification of single-level first-order DARTS, but in fact we can use it to modify in a similar fashion a wide variety of related methods including bilevel DARTS, PC-DARTS [Xu et al., 2020], and GDAS [Dong and Yang, 2019]. This is described more thoroughly in Section 6.6. Thus on each benchmark we start off by attempting to modify the current state-of-the-art method on that benchmark. We conclude with an empirical evaluation of the convergence rate of the architecture parameters of GAEA compared to methods it modifies.

The three benchmarks we use comprise a diverse and complementary set of leading NAS search spaces for computer vision problems. The DARTS search space is large and heavily-studied [Chen et al., 2019, Liang et al., 2019, Nayman et al., 2019, Noy et al., 2019, Xie et al., 2019, Xu et al., 2020], while NAS-Bench-1Shot1 and NAS-Bench-201 are smaller benchmarks that allow for oracle evaluation of weight-sharing algorithms. NAS-Bench-1Shot1 further differs from the other two by applying operations per node instead of per edge. NAS-Bench-201 further differs from the other two by not requiring edge-pruning. We demonstrate that despite this diversity, GAEA is able to improve upon state-of-the-art across all three search spaces.

We defer details of the experimental setup and hyperparameter settings used for GAEA to Section 3.6.4. We note that we use the same learning rate for GAEA variants of DARTS/PC-DARTS and do not require additional weight-decay for architecture parameters. Furthermore, in the interest of reproducibility, we will be releasing all code, hyperparameters, and random seeds for our experiments.

6.5.1 GAEA on the DARTS Search Space

We evaluate GAEA on the task of designing a CNN cell by modifying PC-DARTS [Xu et al., 2020], the current state-of-the-art method, to use the simplex parameterization along with exponentiated gradient (6.7) for the architecture parameters. We consider the same search space as DARTS [Liu et al., 2019], and follow the same three stage process used in Chapter 5 for search and architecture evaluation.

Table 6.1: **DARTS Benchmark (CIFAR-10):** Comparison with manually designed networks and those found by SOTA NAS methods, mainly on the DARTS search space [Liu et al., 2019]. Results grouped by those for manual designs, architectures found by full-evaluation NAS, and architectures found by weight-sharing NAS. All test errors are for models trained with auxiliary towers and cutout (parameter counts exclude auxiliary weights). Test errors we report are averaged over 10 seeds. "-" indicates that the field does not apply while "N/A" indicates unknown. Note that search cost is hardware-dependent; our results used Tesla V100 GPUs.
* We show results for networks with a comparable number of parameters.
† For fair comparison to other work, we show the search cost for training the shared-weights network with a single initialization.
‡ Both the search space and backbone architecture (PyramidNet) differ from the DARTS setting.
# PDARTS results are not reported for multiple seeds. Additionally, PDARTS uses deeper weight-sharing networks during the search process, on which PC-DARTS has also been shown to improve performance. We expect GAEA PC-DARTS to further improve if combined with PDARTS's deeper weight-sharing networks.

		Test Error		Params	Search Cost	Comparable	Search
Architecture	**Source**	Best	Average	(M)	(GPU Days)	**Search Space**	**Method**
Shake-Shake	[Devries and Taylor, 2017]	N/A	2.56	26.2	-	-	manual
PyramidNet	[Yamada et al., 2018]	2.31	N/A	26	-	-	manual
NASNet-A*	[Zoph et al., 2018]	N/A	2.65	3.3	2000	N	RL
AmoebaNet-B*	[Real et al., 2018]	N/A	2.55 ± 0.05	2.8	3150	N	evolution
ProxylessNAS‡	[Cai et al., 2019]	2.08	N/A	5.7	4	N	gradient
ENAS	[Pham et al., 2018]	2.89	N/A	4.6	0.5	Y	RL
Random search WS†	Chapter 5	2.71	2.85 ± 0.08	3.8	0.7	Y	random
ASNG-NAS	[Akimoto et al., 2019]	N/A	2.83 ± 0.14	3.9	0.1	Y	gradient
SNAS	[Xie et al., 2019]	N/A	2.85 ± 0.02	2.8	1.5	Y	gradient
DARTS (1st-order)†	[Liu et al., 2019]	N/A	3.00 ± 0.14	3.3	0.4	Y	gradient
DARTS (2nd-order)†	[Liu et al., 2019]	N/A	2.76 ± 0.09	3.3	1	Y	gradient
PDARTS#	[Chen et al., 2019]	2.50	N/A	3.4	0.3	Y	gradient
PC-DARTS†	[Xu et al., 2020]	N/A	2.57 ± 0.07	3.6	0.1	Y	gradient
GAEA PC-DARTS†	Ours	2.39	2.50 ± 0.06	3.7	0.1	Y	gradient

6.5.1.1 GAEA PC-DARTS on CIFAR-10

As shown in Table 6.1, GAEA PC-DARTS finds an architecture that achieves lower error than comparably sized AmoebaNet-B and NASNet-A architectures, which required thousands of GPU days in search cost. We also demonstrate that geometry-aware co-design of the architecture parameterization and optimization scheme improves upon the original PC-DARTS, which uses Adam [Kingma and Ba, 2015] to optimize a softmax parameterization. GAEA PC-DARTS also outperforms all non-manual methods etoxcept ProxylessNAS [Cai et al., 2019], which takes 40 times as much time, uses 1.5 times as many parameters, and is run on a different search space. Thus on the DARTS search space itself our results improve upon the state-of-the-art.

Broad Reproducibility: Similar to Chapter 5, to meet a higher bar for reproducibility, we report 'broad reproducibility' results by repeating the entire 3 stage pipeline for GAEA PC-DARTS on additional sets of seeds. Our results in Table 6.2 show that while GAEA PC-DARTS is consistently able to find good architectures when selecting the best architecture across four independent weight-sharing trials, multiple trials are required due to sensitivity to the initialization of the weight-sharing network, as is the case for many weight-sharing approaches [Liu et al., 2019, Xu et al., 2020]. Specifically, the performance of GAEA PC-DARTS for one set is similar to that reported in Table 6.1, while the other is on par with the performance reported for PC-DARTS in Xu et al. [2020].

Stage 3 Evaluation		
Set 1 (Reported)	Set 2	Set 3
2.50 ± 0.07	2.50 ± 0.09	2.60 ± 0.09

Table 6.2: GAEA PC-DARTS Stage 3 Evaluation for 3 sets of random seeds.

We depict the top architecture found by GAEA PC-DARTS in Figure 6.1.

6.5.1.2 GAEA PC-DARTS on ImageNet

Following Xu et al. [2020], we also evaluate GAEA PC-DARTS by searching direct on ImageNet. Our evaluation methodology closely mirrors that of Xu et al. [2020]. Namely, we created a smaller dataset from ILSVRC2012 [Russakovsky et al., 2015] for architecture search: the training set included 10% of the images and the validation set included 2.5%. We fixed the architecture weights for the first 35 epochs before running GAEA PC-DARTS with a learning rate of 0.1. All other hyperparameters match that used by Xu et al. [2020]. We depict the architecture found by GAEA PC-DARTS for ImageNet in Figure 6.1.

Table 6.3 shows the final evaluation performance of both the architecture found by GAEA PC-DARTS on CIFAR-10 as well as the one searched directly on ImageNet when trained from scratch for 250 epochs using the same hyperparameter settings as Xu et al. [2020]. Our results demonstrate that the architecture found by GAEA PC-DARTS for ImageNet achieves a top-1 test error of 24.0%, which is state-of-the-art performance in the mobile setting on ImageNet when excluding additional training modifications like label smoothing [Müller et al., 2019], AutoAugment [Cubuk et al., 2019], swish activation [Ramachandran et al., 2017], and squeeze

Table 6.3: **DARTS Benchmark (ImageNet):** Comparison with manually designed networks and those found by SOTA NAS methods, mainly on the DARTS search space [Liu et al., 2019]. Results grouped by those for manual designs, architectures found by full-evaluation NAS, and architectures found by weight-sharing NAS. All test errors are for models trained with auxiliary towers and cutout but no other modifications, e.g., label smoothing [Müller et al., 2019], AutoAugment [Cubuk et al., 2019], swish activation [Ramachandran et al., 2017], squeeze and excite modules [Hu et al., 2018], etc. "-" indicates that the field does not apply while "N/A" indicates unknown. Note that search cost is hardware-dependent; our results used Tesla V100 GPUs.
* We show results for networks with a comparable number of parameters.
† For fair comparison to other work, we show the search cost for training the shared-weights network with a single initialization.
‡ Both the search space and backbone architecture (PyramidNet) differ from the DARTS setting.
# PDARTS results are not reported for multiple seeds. Additionally, PDARTS uses deeper weight-sharing networks during the search process, on which PC-DARTS has also been shown to improve performance. We expect GAEA PC-DARTS to further improve if combined with PDARTS's deeper weight-sharing networks.

		Test Error		Params	Search Cost	Comparable	Search
Architecture	**Source**	top-1	top-5	(M)	(GPU Days)	**Search Space**	**Method**
MobileNet	[Howard et al., 2017]	29.4	10.5	4.2	-	-	manual
ShuffleNet V2 2x	[Ma et al., 2018]	25.1	N/A	~ 5	-	-	manual
NASNet-A*	[Zoph et al., 2018]	26.0	8.4	5.3	1800	N	RL
AmoebaNet-C*	[Real et al., 2018]	24.3	7.6	6.4	3150	N	evolution
DARTS†	[Liu et al., 2019]	26.7	8.7	4.7	4.0	Y	gradient
SNAS	[Xie et al., 2019]	27.3	9.2	4.3	1.5	Y	gradient
ProxylessNAS‡	[Cai et al., 2019]	24.9	7.5	7.1	8.3	N	gradient
PDARTS#	[Chen et al., 2019]	24.4	7.4	4.9	0.3	Y	gradient
PC-DARTS (CIFAR-10)†	[Xu et al., 2020]	25.1	7.8	5.3	0.1	Y	gradient
PC-DARTS (ImageNet)†	[Xu et al., 2020]	24.2	7.3	5.3	3.8	Y	gradient
GAEA PC-DARTS (CIFAR-10)†	Ours	24.3	7.3	5.3	0.1	Y	gradient
GAEA PC-DARTS (ImageNet)†	Ours	24.0	7.3	5.6	3.8	Y	gradient

and excite modules [Hu et al., 2018]. Additionally, the architecture found by GAEA PC-DARTS for CIFAR-10 and transferred to ImageNet achieves a test error of 24.2%, which is comparable to the 24.2% error for the architecture found by PC-DARTS searched directly on ImageNet.

6.5.2 GAEA on NAS-Bench-1Shot1

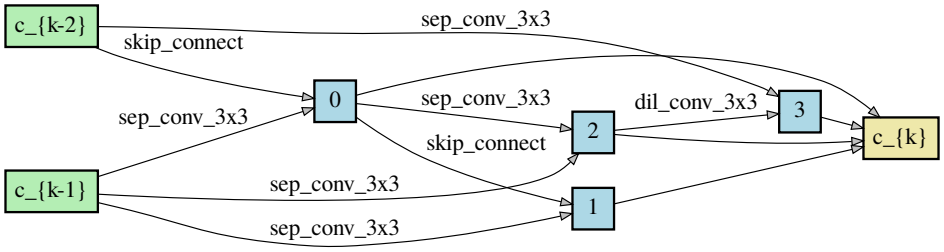
NAS-Bench-1Shot1 [Zela et al., 2020b] considers a subset of NAS-Bench-101 [Ying et al., 2019] that allows the benchmarking of weight-sharing methods on three different search spaces over CIFAR-10. The search spaces differ in the number of nodes considered as well as the number of input edges per node, which jointly affects the search space size. Specifically, search space 1 has 2487 possible architectures, search space 2 has 3609 possible architectures, and search space 3 has 24066 possible architectures. For all three search spaces, a single operation from $\{$3x3 convolution, 1x1 convolution, 3x3 max-pool$\}$ is applied to the sum of inputs to a node.

Of the weight-sharing methods benchmarked by Zela et al. [2020b], PC-DARTS achieves the best performance on 2 of 3 search spaces. Therefore, we apply GAEA again to PC-DARTS to investigate the impact of a geometry-aware approach on these search spaces. The results in Figure 6.2 show that GAEA PC-DARTS finds slightly better architectures than PC-DARTS across all 3 search spaces and exceeds the performance of the best method from Zela et al. [2020b] on 2 out of 3 . We hypothesize that the benefits of GAEA are limited on this benchmark due to the near saturation of NAS methods on its search spaces. In particular, NAS already obtains networks that get within 1% test error of the top architecture in each space, while the standard deviations of the test error for the top architectures in each space when evaluated with different initializations are 0.37%, 0.23% and 0.19% respectively.

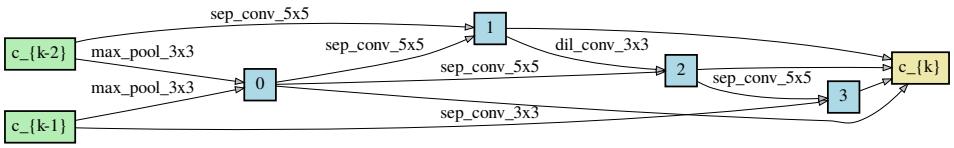
6.5.3 GAEA on NAS-Bench-201

NAS-Bench-201 considers one search space–evaluated on CIFAR10, CIFAR100, and ImageNet16-120–that includes 4-node architectures with an operation from $O = \{$none, skip connect, 1x1 convolution, 3x3 convolution, 3x3 avg pool$\}$ on each edge, yielding 15625 possible architectures. In Table 6.4 we report a subset of the results from Dong and Yang [2020] alongside our GAEA approaches, showing that GDAS Dong and Yang [2019] is the best previous weight-sharing method on all three datasets. Our reproduced results for GDAS are slightly worse than the published result on all three datasets but confirm GDAS as the top weight-sharing method. We evaluate GAEA GDAS on the three datasets and find that it achieves better results on CIFAR-100 than our reproduced runs of GDAS and similar results on the other two datasets.

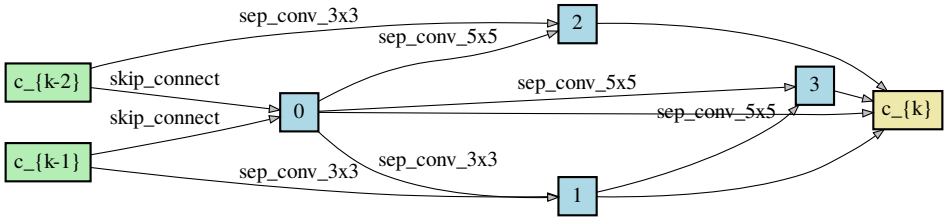
However, as we are interested in improving upon the reported results as well, we proceeded to investigate the performance of GAEA when applied to first-order DARTS. We evaluated GAEA DARTS with both single-level (ERM) and bilevel optimization for architecture parameters; recall that in the bilevel case we optimize the architecture parameters w.r.t. the validation loss and the shared weights w.r.t. the training loss, whereas in the ERM approach there is no training-validation split. Our results show that GAEA DARTS (ERM) achieves state-of-the-art performance on all three datasets, exceeding the test accuracy of both weight-sharing and traditional hyperparameter optimization approaches by a wide margin. This result also shows that our convergence theory, which is for the single-level case, is useful to directly understand strong NAS methods. GAEA



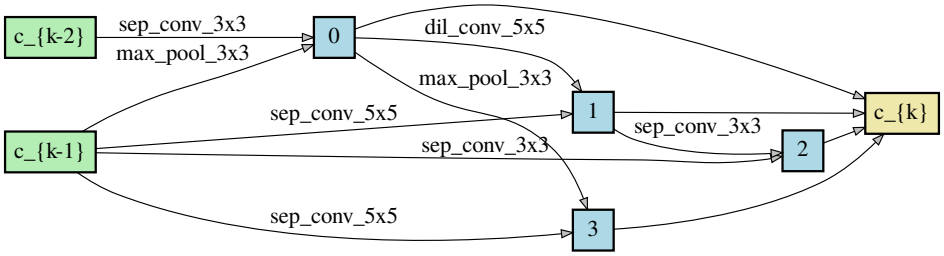
(a) CIFAR-10: Normal Cell



(b) CIFAR-10: Reduction Cell



(c) ImageNet: Normal Cell



(d) ImageNet: Reduction Cell

Figure 6.1: Normal and reduction cells found by GAEA PC-DARTS on CIFAR-10 and ImageNet.

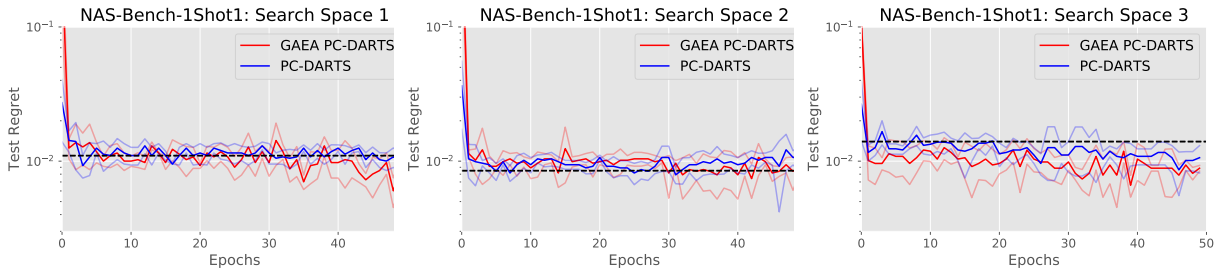


Figure 6.2: **NAS-Bench-1Shot1:** Chart of the ground truth performance of the proposed architecture by epoch of shared-weights training averaged over 4 different shared-weights initializations. The dashed black line indicates the performance of the best weight-sharing method originally evaluated on the benchmark [Zela et al., 2020b].

Method	Search (seconds)	CIFAR-10 test	CIFAR-100 test	ImageNet16-120 test
RSWS	7587	87.66 ± 1.69	58.33 ± 4.34	31.14 ± 3.88
DARTS	35781	54.30 ± 0.00	15.61 ± 0.00	16.32 ± 0.00
SETN	34139	87.64 ± 0.00	59.05 ± 0.24	32.52 ± 0.21
GDAS	31609	93.61 ± 0.09	70.70 ± 0.30	41.71 ± 0.98
GDAS (Reproduced)	27923†	93.52 ± 0.15	67.52 ± 0.15	40.91 ± 0.12
GAEA GDAS	16754†	93.55 ± 0.13	70.47 ± 0.47	40.91 ± 0.12
GAEA DARTS (ERM)	9061†	94.10 ± 0.29	73.43 ± 0.13	46.36 ± 0.00
GAEA DARTS (bilevel)	7930†	91.87 ± 0.57	71.87 ± 0.57	45.69 ± 0.56
REA	N/A	93.92 ± 0.30	71.84 ± 0.99	45.54 ± 1.03
RS	N/A	93.70 ± 0.36	71.04 ± 1.08	44.57 ± 1.25
REINFORCE	N/A	93.85 ± 0.37	71.71 ± 1.09	45.25 ± 1.18
BOHB	N/A	93.61 ± 0.52	70.85 ± 1.28	44.42 ± 1.49
ResNet	N/A	93.97	70.86	43.63
Optimal	N/A	94.37	73.51	47.31

† Search cost measured on NVIDIA P100 GPUs.

Table 6.4: **NAS-Bench-201:** Results are separated into those for weight-sharing methods (top) and those for traditional hyperparameter optimization methods.

DARTS (bilevel) does not perform as well as single-level but still exceeds all other methods on CIFAR-100 and ImageNet16-120.

6.5.4 Convergence of GAEA

While in the previous sections we focused on the test accuracy of GAEA, we now examine the impact of GAEA on the speed of convergence of the architecture parameters during training. Figure 6.3 shows the entropy of the operation weights averaged across nodes for a GAEA-

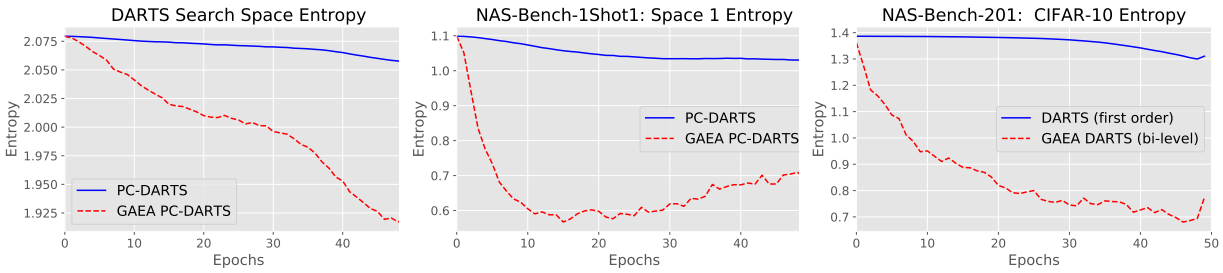


Figure 6.3: **GAEA Convergence:** Evolution over search phase epochs of the average entropy of the operation-weights for GAEA and the approaches it modifies when run on the DARTS search space (left), NAS-Bench-1Shot1 Search Space 1 (middle), and NASBench-201 on CIFAR-10 (right). GAEA reduces entropy much more quickly than previous gradient-based methods, allowing it to quickly obtain sparse architecture weights. This leads to both faster convergence to a single architecture and a lower loss when pruning at the end of the search phase.

variant versus the base method across all three benchmarks. It is evident that the entropy of operation weights decreases much faster for GAEA-modified approaches across all benchmarks relative to the base methods. This also validates our expectation that GAEA encourages sparsity in architecture parameters, which should alleviate the mismatch between using a continuous relaxation of the architecture space and the discrete architecture derivation performed at the end of search. Note that, in addition to GAEA's use of exponentiated gradient over the simplex, the entropy decrease of PC-DARTS may be further slowed by an additional weight decay term that is applied to architecture weights, which is not required for GAEA PC-DARTS.

6.5.5 Comparison to XNAS

As discussed in Section 6.1.1, XNAS is similar to GAEA in that it uses an exponentiated gradient update but is motivated from a regret minimization perspective. Nayman et al. [2019] provides regret bounds for XNAS relative to the observed sequence of validation losses, however, this is not equivalent to the regret relative to the best architecture in the search space, which would have generated a different sequence of validation losses.

XNAS also differs in its implementation in two ways: (1) a wipeout routine is used to zero out operations that cannot recover to exceed the current best operation within the remaining number of iterations and (2) architecture gradient clipping is applied per data point before aggregating to form the update. These differences are motivated from the regret analysis and meaningfully increase the complexity of the algorithm. Unfortunately, the authors do not provide the code for architecture search in their code release at https://github.com/NivNayman/XNAS. Nonetheless, we implemented XNAS [Nayman et al., 2019] for the NAS-Bench-201 search space to provide a point of comparison to GAEA.

Our results shown in Figure 6.4 demonstrate that XNAS exhibits much of the same behavior as DARTS in that the operations all converge to skip connections. We hypothesize that this is due to the gradient clipping, which obscures the signal kept by GAEA DARTS in favor of convolutional operations.

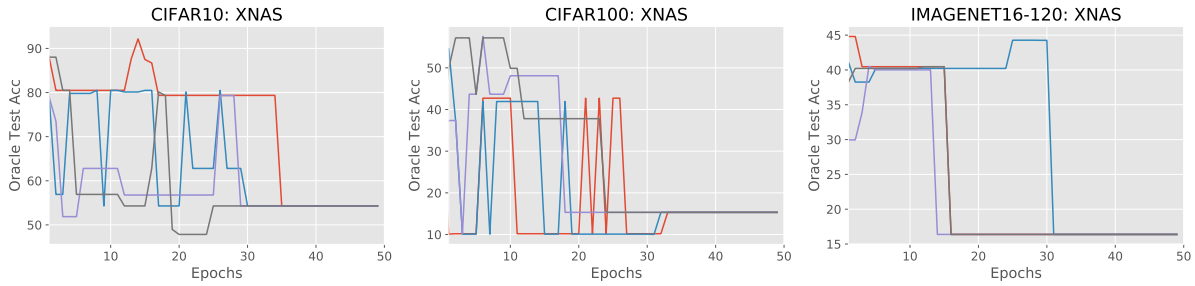


Figure 6.4: **NAS-Bench-201: XNAS Learning Curves.** Evolution over search phase epochs of the best architecture according 4 runs of XNAS. XNAS exhibits the same behavior as DARTS and converges to nearly all skip connections.

6.6 Proof of Theoretical Results

This section contains proofs and generalizations of the non-convex optimization results in Section 6.3. Throughout this section, $\mathbb{V}$ denotes a finite-dimensional real vector space with Euclidean inner product $\langle \cdot, \cdot \rangle$, $\mathbb{R}_+$ denotes the set of nonnegative real numbers, and $\overline{\mathbb{R}}$ denotes the set of extended real numbers $\mathbb{R} \cup \{\pm\infty\}$.

6.6.1 Preliminaries

Definition 1. *Consider a closed and convex subset $\mathcal{X} \subset \mathbb{V}$. For any $\alpha > 0$ and norm $\| \cdot \| : \mathcal{X} \mapsto \mathbb{R}_+$ an everywhere-subdifferentiable function $f : \mathcal{X} \mapsto \overline{\mathbb{R}}$ is called* α**-strongly-convex** *w.r.t. $\| \cdot \|$ if $\forall\, \mathbf{x}, \mathbf{y} \in \mathcal{X}$ we have*

$$f(\mathbf{y}) \geq f(\mathbf{x}) + \langle \nabla f(\mathbf{x}), \mathbf{y} - \mathbf{x} \rangle + \frac{\alpha}{2} \|\mathbf{y} - \mathbf{x}\|^2$$

Definition 2. *Consider a closed and convex subset $\mathcal{X} \subset \mathbb{V}$. For any $\beta > 0$ and norm $\| \cdot \| : \mathcal{X} \mapsto \mathbb{R}_+$ an continuously-differentiable function $f : \mathcal{X} \mapsto \mathbb{R}$ is called* β**-strongly-smooth** *w.r.t. $\| \cdot \|$ if $\forall\, \mathbf{x}, \mathbf{y} \in \mathcal{X}$ we have*

$$f(\mathbf{y}) \leq f(\mathbf{x}) + \langle \nabla f(\mathbf{x}), \mathbf{y} - \mathbf{x} \rangle + \frac{\beta}{2} \|\mathbf{y} - \mathbf{x}\|^2$$

Definition 3. *Let $\mathcal{X}$ be a closed and convex subset of $\mathbb{V}$. The* **Bregman divergence** *induced by a strictly convex, continuously-differentiable* **distance-generating function (DGF)** *$\phi : \mathcal{X} \mapsto \overline{\mathbb{R}}$ is*

$$D_\phi(\mathbf{x} || \mathbf{y}) = \phi(\mathbf{x}) - \phi(\mathbf{y}) - \langle \nabla \phi(\mathbf{y}), \mathbf{x} - \mathbf{y} \rangle \; \forall\, \mathbf{x}, \mathbf{y} \in \mathcal{X}$$

By definition, the Bregman divergence satisfies the following properties:

1. *$D_\phi(\mathbf{x} || \mathbf{y}) \geq 0 \; \forall\, x, y \in \mathcal{X}$ and $D_\phi(\mathbf{x} || \mathbf{y}) = 0 \iff x = y$.*
2. *If ϕ is α-strongly-convex w.r.t. norm $\| \cdot \|$ then so is $D_\phi(\cdot || \mathbf{y}) \; \forall\, \mathbf{y} \in \mathcal{X}$. Furthermore, $D_\phi(\mathbf{x} || \mathbf{y}) \geq \frac{\alpha}{2} \|\mathbf{x} - \mathbf{y}\|^2 \; \forall\, \mathbf{x}, \mathbf{y} \in \mathcal{X}$.*
3. *If ϕ is β-strongly-smooth w.r.t. norm $\| \cdot \|$ then so is $D_\phi(\cdot || \mathbf{y}) \; \forall\, \mathbf{y} \in \mathcal{X}$. Furthermore, $D_\phi(\mathbf{x} || \mathbf{y}) \leq \frac{\beta}{2} \|\mathbf{x} - \mathbf{y}\|^2 \; \forall\, \mathbf{x}, \mathbf{y} \in \mathcal{X}$.*

Definition 4. *[Zhang and He, 2018, Definition 2.1] Consider a closed and convex subset $\mathcal{X} \subset \mathbb{V}$. For any $\gamma > 0$ and $\phi : \mathcal{X} \mapsto \overline{\mathbb{R}}$ an everywhere-subdifferentiable function $f : \mathcal{X} \mapsto \mathbb{R}$ is called* **γ-relatively-weakly-convex (γ-RWC)** *w.r.t. ϕ if $f(\cdot) + \gamma\phi(\cdot)$ is convex on $\mathcal{X}$.*

Definition 5. *[Zhang and He, 2018, Definition 2.3] Consider a closed and convex subset $\mathcal{X} \subset \mathbb{V}$. For any $\lambda > 0$, function $f : \mathcal{X} \mapsto \mathbb{R}$, and DGF $\phi : \mathcal{X} \mapsto \overline{\mathbb{R}}$ the* **Bregman proximal operator** *of f is*

$$\operatorname{prox}_\lambda(\mathbf{x}) = \arg\min_{\mathbf{u} \in \mathcal{X}} \lambda f(\mathbf{u}) + D_\phi(\mathbf{u}||\mathbf{x})$$

Definition 6. *[Zhang and He, 2018, Equation 2.11] Consider a closed and convex subset $\mathcal{X} \subset \mathbb{V}$. For any $\lambda > 0$, function $f : \mathcal{X} \mapsto \mathbb{R}$, and DGF $\phi : \mathcal{X} \mapsto \overline{\mathbb{R}}$ the* **Bregman stationarity** *of f at any point $\mathbf{x} \in \mathcal{X}$ is*

$$\Delta_\lambda(\mathbf{x}) = \frac{D_\phi(\mathbf{x}||\operatorname{prox}_\lambda(\mathbf{x})) + D_\phi(\operatorname{prox}_\lambda(\mathbf{x})||\mathbf{x})}{\lambda^2}$$

6.6.2 Results

Throughout this subsection let $\mathbb{V} = \bigtimes_{i=1}^b \mathbb{V}_i$ be a product space of b finite-dimensional real vector spaces $\mathbb{V}_i$, each with an associated norm $\|\cdot\|_i : \mathbb{V}_i \mapsto \mathbb{R}_+$, and $\mathcal{X} = \bigtimes_{i=1}^b \mathcal{X}_i$ be a product set of b subsets $\mathcal{X}_i \subset \mathbb{V}_i$, each with an associated 1-strongly-convex DGF $\phi_i : \mathcal{X}_i \mapsto \overline{\mathbb{R}}$ w.r.t. $\|\cdot\|_i$. For each $i \in [b]$ will use $\|\cdot\|_{i,*}$ to denote the dual norm of $\|\cdot\|_i$ and for any element $\mathbf{x} \in \mathcal{X}$ we will use $\mathbf{x}_i$ to denote its component in block i and $\mathbf{x}_{-i}$ to denote the component across all blocks other than i. Define the functions $\|\cdot\| : \mathbb{V} \mapsto \mathbb{R}_+$ and $\|\cdot\|_* \mathbb{V} \mapsto \mathbb{R}_+$ for any $\mathbf{x} \in \mathbb{V}$ by $\|\mathbf{x}\|^2 = \sum_{i=1}^b \|\mathbf{x}_i\|_i^2$ and $\|\mathbf{x}\|_*^2 = \sum_{i=1}^b \|\mathbf{x}_i\|_{i,*}^2$, respectively, and the function $\phi : \mathcal{X} \mapsto \overline{\mathbb{R}}$ for any $\mathbf{x} \in \mathcal{X}$ by $\phi(\mathbf{x}) = \sum_{i=1}^b \phi_i(\mathbf{x})$. Finally, for any $n \in \mathbb{N}$ we will use $[n]$ to denote the set $\{1, \ldots, n\}$.

Setting 1. *For some fixed constants $\gamma_i, L_i > 0$ for each $i \in [b]$ we have the following:*

1. *$f : \mathcal{X} \mapsto \mathbb{R}$ is everywhere-subdifferentiable with minimum $f^* > -\infty$ and for all $\mathbf{x} \in \mathcal{X}$ and each $i \in [b]$ the restriction $f(\cdot, \mathbf{x}_{-i})$ is γ_i-RWC w.r.t. ϕ_i.*
2. *For each $i \in [b]$ there exists a stochastic oracle G_i that for input $\mathbf{x} \in \mathcal{X}$ outputs a random vector $G_i(\mathbf{x}, \xi)$ s.t. $\mathbb{E}_\xi G_i(\mathbf{x}, \xi) \in \partial_i f(\mathbf{x})$, where $\partial_i f(\mathbf{x})$ is the subdifferential set of the restriction $f(\cdot, \mathbf{x}_{-i})$ at $\mathbf{x}_i$. Moreover, $\mathbb{E}_\xi \|G_i(\mathbf{x}, \xi)\|_{i,*}^2 \leq L_i^2$.*

Define $\gamma = \max_{i \in [b]} \gamma_i$ and $L^2 = \sum_{i=1}^b L_i^2$.

Claim 1. *$\|\cdot\|$ is a norm on $\mathbb{V}$.*

Proof. Positivity and homogeneity are trivial. For the triangle inequality, note that for any

$\lambda \in [0,1]$ and any $\mathbf{x}, \mathbf{y} \in \mathcal{X}$ we have that

$$\begin{aligned}
\|\lambda\mathbf{x} + (1-\lambda)\mathbf{y}\| &= \sqrt{\sum_{i=1}^{b} \|\lambda\mathbf{x}_i + (1-\lambda)\mathbf{y}_i\|_i^2} \\
&\leq \sqrt{\sum_{i=1}^{b} (\lambda\|\mathbf{x}_i\|_i + (1-\lambda)\|\mathbf{y}_i\|_i)^2} \\
&\leq \lambda\sqrt{\sum_{i=1}^{b} \|\mathbf{x}_i\|_i^2} + (1-\lambda)\sqrt{\sum_{i=1}^{b} \|\mathbf{y}_i\|_i^2} \\
&= \lambda\|\mathbf{x}\| + (1-\lambda)\|\mathbf{y}\|
\end{aligned}$$

where the first inequality follows by convexity of the norms $\|\cdot\|_i \; \forall \; i \in [b]$ and the fact that the Euclidean norm on $\mathbb{R}^b$ is nondecreasing in each argument, while the second inequality follows by convexity of the Euclidean norm on $\mathbb{R}^b$. $\square$

Algorithm 5: Block-stochastic mirror descent over $\mathcal{X} = \bigtimes_{i=1}^{b} \mathcal{X}_i$ given associated DGFs $\phi_i : \mathcal{X}_i \mapsto \overline{\mathbb{R}}$.

Input: initialization $\mathbf{x}^{(1)} \in \mathcal{X}$, number of steps $T \geq 1$, step-size sequence $\{\eta_t\}_{t=1}^{T}$

for iteration $t \in [T]$ **do**
  sample $i \sim \mathrm{Unif}[b]$
  set $\mathbf{x}_{-i}^{(t+1)} = \mathbf{x}_{-i}^{(t)}$
  get $\mathbf{g} = G_i(\mathbf{x}^{(t)}, \xi_t)$
  set $\mathbf{x}_i^{(t+1)} = \arg\min_{\mathbf{u} \in \mathcal{X}_i} \eta_t \langle \mathbf{g}, \mathbf{u} \rangle + D_{\phi_i}(\mathbf{u} || \mathbf{x}_i^{(t)})$
end

Output: $\hat{\mathbf{x}} = \mathbf{x}^{(t)}$ w.p. $\frac{\eta_t}{\sum_{t=1}^{T} \eta_t}$.

Claim 2. *$\frac{1}{2}\|\cdot\|_*^2$ is the convex conjugate of $\frac{1}{2}\|\cdot\|^2$.*

Proof. Consider any $\mathbf{u} \in \mathbb{V}$. To upper-bound the convex conjugate note that

$$\begin{aligned}
\sup_{\mathbf{x} \in \mathbb{V}} \langle \mathbf{u}, \mathbf{x} \rangle - \frac{\|\mathbf{x}\|^2}{2} &= \sup_{\mathbf{x} \in \mathbb{V}} \sum_{i=1}^{b} \langle \mathbf{u}_i, \mathbf{x}_i \rangle - \frac{\|\mathbf{x}_i\|_i^2}{2} \\
&\leq \sup_{\mathbf{x} \in \mathbb{V}} \sum_{i=1}^{b} \|\mathbf{u}_i\|_{i,*} \|\mathbf{x}_i\|_i - \frac{\|\mathbf{x}_i\|_i^2}{2} \\
&= \frac{1}{2} \sum_{i=1}^{b} \|\mathbf{u}_i\|_{i,*}^2 \\
&= \frac{\|\mathbf{u}\|_*^2}{2}
\end{aligned}$$

where the first inequality follows by definition of a dual norm and the second by maximizing each term w.r.t. $\|\mathbf{x}_i\|_i$. For the lower bound, pick $\mathbf{x} \in \mathbb{V}$ s.t. $\langle \mathbf{u}_i, \mathbf{x}_i \rangle = \|\mathbf{u}_i\|_{i,*} \|\mathbf{x}_i\|_i$ and $\|\mathbf{x}_i\|_i = \|\mathbf{u}_i\|_{i,*}$ $\forall\, i \in [b]$, which must exist by the definition of a dual norm. Then

$$\langle \mathbf{u}, \mathbf{x} \rangle - \frac{\|\mathbf{x}\|^2}{2} = \sum_{i=1}^{b} \langle \mathbf{u}_i, \mathbf{x}_i \rangle - \frac{\|\mathbf{x}_i\|_i^2}{2} = \frac{1}{2} \sum_{i=1}^{b} \frac{\|\mathbf{u}_i\|_{i,*}^2}{2} = \frac{\|\mathbf{u}\|_*^2}{2}$$

so $\sup_{\mathbf{x} \in \mathbb{V}} \langle \mathbf{u}, \mathbf{x} \rangle - \frac{1}{2}\|\mathbf{x}\|^2 \geq \frac{1}{2}\|\mathbf{u}\|_*^2$, completing the proof. □

Theorem 2. *Let $\hat{x}$ be the output of Algorithm 5 after T iterations with non-increasing step-size sequence $\{\eta_t\}_{t=1}^T$. Then under Setting 1, for any $\hat{\gamma} > \gamma$ we have that*

$$\mathbb{E}\Delta_{\frac{1}{\hat{\gamma}}}(\hat{\mathbf{x}}) \leq \frac{\hat{\gamma}b}{\hat{\gamma}-\gamma} \frac{\min_{\mathbf{u}\in\mathcal{X}} f(\mathbf{u}) + \hat{\gamma} D_\phi(\mathbf{u}||\mathbf{x}^{(1)}) - f^* + \frac{\hat{\gamma}L^2}{2b}\sum_{t=1}^T \eta_t^2}{\sum_{t=1}^T \eta_t}$$

where the expectation is w.r.t. ξ_t and the randomness of the algorithm.

Proof. Define transforms $\mathbf{U}_i, i \in [b]$ s.t. $\mathbf{U}_i^T\mathbf{x} = \mathbf{x}_i$ and $\mathbf{x} = \sum_{i=1}^b \mathbf{U}_i\mathbf{x}_i \;\forall\, \mathbf{x} \in \mathcal{X}$. Let G be a stochastic oracle that for input $\mathbf{x} \in \mathcal{X}$ outputs $G(\mathbf{x}, i, \xi) = b\mathbf{U}_i G_i(\mathbf{x}, \xi)$. This implies $\mathbb{E}_{i,\xi} G(\mathbf{x}, i, \xi) = \frac{1}{b}\sum_{i=1}^b b\mathbf{U}_i\mathbb{E}_\xi G_i(\mathbf{x}, \xi) \in \sum_{i=1}^b \mathbf{U}_i\partial_i f(\mathbf{x}) = \partial f(\mathbf{x})$ and $\mathbb{E}_{i,\xi}\|G(\mathbf{x}, i, \xi)\|_*^2 = \frac{1}{b}\sum_{i=1}^b b^2\mathbb{E}_\xi\|\mathbf{U}_i G_i(\mathbf{x}, \xi)\|_{i,*}^2 \leq b\sum_{i=1}^b L_i^2 = bL^2$. Then

$$\begin{aligned}
\mathbf{x}^{(t+1)} &= \mathbf{U}_i\left[\arg\min_{\mathbf{u}\in\mathcal{X}_i} \eta_t\langle g, u\rangle + D_{\phi_i}(\mathbf{u}||\mathbf{x}_i^{(t)})\right] \\
&= \mathbf{U}_i\mathbf{U}_i^T\left[\arg\min_{\mathbf{u}\in\mathcal{X}} \eta_t\langle \mathbf{U}_i G_i(\mathbf{x}^{(t)}, \xi_t), \mathbf{u}\rangle + \sum_{i=1}^b D_{\phi_i}(\mathbf{u}_i||\mathbf{x}_i^{(t)})\right] \\
&= \arg\min_{\mathbf{u}\in\mathcal{X}} \frac{\eta_t}{b}\langle G(\mathbf{x}, i, \xi_t), \mathbf{u}\rangle + D_\phi(\mathbf{u}||\mathbf{x}^{(t)})
\end{aligned}$$

Thus Algorithm 5 is equivalent to [Zhang and He, 2018, Algorithm 1] with stochastic oracle $G(\mathbf{x}, i, \xi)$, step-size sequence $\{\eta_t/b\}_{t=1}^T$, and no regularizer. Note that ϕ is 1-strongly-convex w.r.t. $\|\cdot\|$ and f is γ-RWC w.r.t. ϕ, so in light of Claims 1 and 2 our setup satisfies [Zhang and He, 2018, Assumption 3.1]. The result then follows from [Zhang and He, 2018, Theorem 3.1]. □

Corollary 1. *Under Setting 1 let $\hat{\mathbf{x}}$ be the output of Algorithm 5 with constant step-size $\eta_t = \sqrt{\frac{2b(f^{(1)}-f^*)}{\gamma L^2 T}}\;\forall\, t \in [T]$, where $f^{(1)} = f(\mathbf{x}^{(1)})$. Then we have*

$$\mathbb{E}\Delta_{\frac{1}{2\gamma}}(\hat{\mathbf{x}}) \leq 2L\sqrt{\frac{2b\gamma(f^{(1)}-f^*)}{T}}$$

where the expectation is w.r.t. ξ_t and the randomness of the algorithm. Equivalently, we can reach a point $\hat{\mathbf{x}}$ satisfying $\mathbb{E}\Delta_{\frac{1}{2\gamma}}(\hat{\mathbf{x}}) \leq \varepsilon$ in $\frac{8\gamma bL^2(f^{(1)}-f^*)}{\varepsilon^2}$ stochastic oracle calls.

6.7 Experiment Details

We provide additional detail on the experimental setup and hyperparameter settings used for each benchmark studied in Section 6.5. We also provide a more detail discussion of how XNAS differs from GAEA, along with empirical results for XNAS on the NAS-Bench-201 benchmark.

6.7.1 DARTS Search Space

We consider the same search space as DARTS [Liu et al., 2019], which has become one of the standard search spaces for CNN cell search [Chen et al., 2019, Liang et al., 2019, Nayman et al., 2019, Noy et al., 2019, Xie et al., 2019]. Following the evaluation procedure used in Liu et al. [2019] and Xu et al. [2020] , our evaluation of GAEA PC-DARTS consists of three stages:

- **Stage 1:** In the search phase, we run GAEA PC-DARTS with 5 random seeds to reduce variance from different initialization of the shared-weights network.[3]
- **Stage 2:** We evaluate the best architecture identified by each search run by training from scratch.
- **Stage 3:** We perform a more thorough evaluation of the best architecture from stage 2 by training with ten different random seed initializations.

For completeness, we describe the convolutional neural network search space considered. A cell consists of 2 input nodes and 4 intermediate nodes for a total of 6 nodes. The nodes are ordered and subsequent nodes can receive the output of prior nodes as input so for a given node k, there are $k-1$ possible input edges to node k. Therefore, there are a total of $2+3+4+5=14$ edges in the weight-sharing network.

An architecture is defined by selecting 2 input edges per intermediate node and also selecting a single operation per edge from the following 8 operations: (1) 3×3 separable convolution, (2) 5×5 separable convolution, (3) 3×3 dilated convolution, (4) 5×5 dilated convolution, (5) max pooling, (6) average pooling, (7) identity (8) zero.

We use the same search space to design a "normal" cell and a "reduction" cell; the normal cells have stride 1 operations that do not change the dimension of the input, while the reduction cells have stride 2 operations that half the length and width dimensions of the input. In the experiments, for both cell types, , after which the output of all intermediate nodes are concatenated to form the output of the cell.

6.7.1.1 Stage 1: Architecture Search

For stage 1, as is done by DARTS and PC-DARTS, we use GAEA PC-DARTS to update architecture parameters for a smaller shared-weights network in the search phase with 8 layers and 16 initial channels. All hyperparameters for training the weight-sharing network are the same as that used by PC-DART:

train:

scheduler: cosine

[3]Note Liu et al. [2019] trains the weight-sharing network with 4 random seeds. However, since PC-DARTS is significantly faster than DARTS, the cost of an additional seed is negligible.

lr_anneal_cycles: 1
smooth_cross_entropy: false
batch_size: 256
learning_rate: 0.1
learning_rate_min: 0.0
momentum: 0.9
weight_decay: 0.0003
init_channels: 16
layers: 8
autoaugment: false
cutout: false
auxiliary: false
drop_path_prob: 0
grad_clip: 5

For GAEA PC-DARTS, we initialize the architecture parameters with equal weight across all options (equal weight across all operations per edge and equal weight across all input edges per node). Then, we train the shared-weights network for 10 epochs without performing any architecture updates to warmup the weights. Then, we use a learning rate of 0.1 for the exponentiated gradient update for GAEA PC-DARTS.

6.7.1.2 Stage 2 and 3: Architecture Evaluation

For stages 2 and 3, we train each architecture for 600 epochs with the same hyperparameter settings as PC-DARTS:

train:
scheduler: cosine
lr_anneal_cycles: 1
smooth_cross_entropy: false
batch_size: 128
learning_rate: 0.025
learning_rate_min: 0.0
momentum: 0.9
weight_decay: 0.0003
init_channels: 36
layers: 20
autoaugment: false
cutout: true
cutout_length: 16
auxiliary: true
auxiliary_weight: 0.4
drop_path_prob: 0.3
grad_clip: 5

6.7.2 NAS-Bench-1Shot1

The NAS-Bench-1Shot1 benchmark [Zela et al., 2020b] contains 3 different search spaces that are subsets of the NAS-Bench-101 search space. The search spaces differ in the number of nodes and the number of input edges selected per node. We refer the reader to Zela et al. [2020b] for details about each individual search space.

Of the NAS methods evaluated by Zela et al. [2020b], PC-DARTS had the most robust performance across the three search spaces and converged to the best architecture in search spaces 1 and 3. GDAS, a probabilistic gradient NAS method, achieved the best performance on search space 2. Hence, we focused on applying a geometry-aware approach to PC-DARTS. We implemented GAEA PC-DARTS within the repository provided by Zela et al. [2020b] available at https://github.com/automl/nasbench-1shot1. We used the same hyperparameter settings for training the weight-sharing network as that used by Zela et al. [2020b] for PC-DARTS. Similar to the previous benchmark, we initialize architecture parameters to allocate equal weight to all options. For the architecture updates, the only hyperparameter for GAEA PC-DARTS is the learning rate for exponentiated gradient, which we set to 0.1.

As mentioned in Section 6.5, the search spaces considered in this benchmark differ in that operations are applied after aggregating all edge inputs to a node instead of per edge input as in the DARTS and NAS-Bench-201 search spaces. This structure inherently limits the size of the weight-sharing network to scale with the number of nodes instead of the number of edges ($\mathcal{O}(|\text{nodes}|^2)$), thereby limiting the degree of overparameterization. Understanding the impact of overparameterization on the performance of weight-sharing NAS methods is a direction for future study.

6.7.3 NAS-Bench-201

The NAS-Bench-201 benchmark [Dong and Yang, 2020] evaluates a single search space across 3 datasets: CIFAR-10, CIFAR-100, and a miniature version of ImageNet (ImageNet16-120). ImageNet16-120 is a downsampled version of ImageNet with 16×16 images and 120 classes for a total of 151.7k training images, 3k validation images, and 3k test images.

Dong and Yang [2020] evaluated the architecture search performance of multiple weight-sharing methods and traditional hyperparameter optimization methods on all three datasets. According to the results by Dong and Yang [2020], GDAS outperformed other weight-sharing methods by a large margin. Hence, we first evaluated the performance of GAEA GDAS on each of the three datasets. Our implementation of GAEA GDAS uses an architecture learning rate of 0.1, which matches the learning rate used for GAEA approaches in the previous two benchmarks. Additionally, we run GAEA GDAS for 150 epochs instead of 250 epochs used for GDAS in the original benchmarked results; this is why the search cost is lower for GAEA GDAS. All other hyperparameter settings are the same. Our results for GAEA GDAS is comparable to the reported results for GDAS on CIFAR-10 and CIFAR-100 but slightly lower on ImageNet16-120. Compared to our reproduced results for GDAS, GAEA GDAS outperforms GDAS on CIFAR-100 and matches it on CIFAR-10 and ImageNet16-120.

Next, to see if we can use GAEA to further improve the performance of weight-sharing methods, we evaluated GAEA DARTS (first order) applied to both the single-level (ERM) and

bi-level optimization problems. Again, we used a learning rate of 0.1 and trained GAEA DARTS for 25 epochs on each dataset. The one additional modification we made was to exclude the zero operation, which limits GAEA DARTS to a subset of the search space. To isolate the impact of this modification, we also evaluated first-order DARTS with this modification. Similar to Dong and Yang [2020], we observe DARTS with this modification to also converge to architectures with nearly all skip connections, resulting in similar performance as that reported by Dong and Yang [2020].

For GAEA GDAS and GAEA DARTS, we train the weight-sharing network with the following hyperparameters:

```
train:
  scheduler: cosine
  lr_anneal_cycles: 1
  smooth_cross_entropy: false
  batch_size: 64
  learning_rate: 0.025
  learning_rate_min: 0.001
  momentum: 0.9
  weight_decay: 0.0003
  init_channels: 16
  layers: 5
  autoaugment: false
  cutout: false
  auxiliary: false
  auxiliary_weight: 0.4
  drop_path_prob: 0
  grad_clip: 5
```

Surprisingly, we observe single-level optimization to yield better performance than solving the bi-level problem with GAEA DARTS on this search space. In fact, the performance of GAEA DARTS (ERM) not only exceeds that of GDAS, but also outperforms traditional hyper-parameter optimization approaches on all three datasets, nearly reaching the optimal accuracy on all three datasets. In contrast, GAEA DARTS (bi-level) outperforms GDAS on CIFAR-100 and ImageNet16-120 but underperforms slightly on CIFAR-10. The single-level results on this benchmark provides concrete support for our convergence analysis, which only applies to the ERM problem.

As noted in Section 6.5, the search space considered in this benchmark differs from the prior two in that there is no subsequent edge pruning. Additionally, the search space is fairly small with only 3 nodes for which architecture decisions must be made. The success of GAEA DARTS (ERM) on this benchmark indicate the need for a better understanding of when single-level optimization should be used in favor of the default bi-level optimization problem and how the search space impacts the decision.

Chapter 7

Weight-Sharing Beyond NAS

In Chapter 5, we showed that combining random search with weight-sharing improved performance over ASHA on NAS benchmarks. Additionally, in the previous chapter, we developed a geometry-aware gradient-based weight-sharing approach that achieved state-of-the-art performance on image classification tasks. Given the surprising effectiveness of weight-sharing, it is natural to wonder whether it is possible to extend this approach to accelerate algorithms in other settings of interest, and in doing so to enhance our empirical and/or theoretical understanding of the original NAS problem. Since NAS is a specialized instance of hyperparameter optimization, the latter is a reasonable place to start.

In this chapter, we show that weight-sharing can be fruitfully applied to a subclass of hyperparameter optimization problems we refer to as *architecture search*, yielding:

- Fast and simple methods for two important problems: (1) feature map selection and (2) hyperparameter optimization for federated learning.
- Novel insight into the use of bilevel optimization in NAS by way of comparison with a simpler setting.

Our contributions, which we summarize below, demonstrate that weight-sharing is a powerful heuristic worthy of both further theoretical study and widespread application beyond the NAS setting.

- Starting from the observation that feature map selection—the problem of picking the best of a finite set of fixed data representations—can be viewed as a shallow NAS problem, we apply weight-sharing to construct a simple, fast, and accurate algorithm that we show outperforms strong baselines– random search [Bergstra and Bengio, 2012] and successive halving (Chapter 2)– on a feature map selection problem.
- Motivated by this success, we use feature map selection as a simple setting to study architecture search. Empirically, we observe several phenomena in this setting that are similar to those seen in NAS with weight-sharing. Theoretically, we give a possible explanation for why most modern NAS methods employ a bilevel objective, despite the fact that weight-sharing precludes standard model-selection-based justifications for doing so.
- Finally, we study hyperparameter optimization for federated learning, which performs massively distributed optimization over networks of heterogenous devices. We show that weight-sharing is particularly well-suited to this setting due to the limited amount of

computation and inherently noisy validation signal from ephemeral data. This motivates the development of *Federated Exponentiated-Gradient* (FedEx), a simple extension of the popular Federated Averaging (FedAvg) method [McMahan et al., 2017] that uses weight-sharing to automatically tune optimization parameters while simultaneously training across devices.

7.1 The Weight-Sharing Learning Problem

Recall from Chapter 5.3 that weight-sharing is an efficient evaluation method for NAS that trains a single supernetwork in lieu of individual architectures to obtain noisy signals of the performance of different architectures from the search space. In this chapter, we will use weight-sharing to refer to any technique that uses the same set of weights $\mathbf{w} \in \mathcal{W}$ to obtain noisy evaluations of the quality of multiple configurations $c \in \mathcal{C}$, thus removing the need to train custom weights for each one. Note that in NAS, $\mathcal{C}$ is the set of all possible architectures in a given search space and the goal is to find $c \in \mathcal{C}$ that can be optimized to achieve low population error, while d is the number of weights needed to parameterize the largest architecture in $\mathcal{C}$.

Similar to Chapter 6.2, for our more general search space $\mathcal{C}$, the joint hypothesis space over configurations and parameters is the following union:

$$H_{\mathcal{C}} = \bigcup_{c \in \mathcal{C}} H_c = \{h_{\mathbf{w},c} : X \mapsto Y, \mathbf{w} \in \mathcal{W}, c \in \mathcal{C}\}$$

As usual the goal of learning is to find $h_{\mathbf{w},c} \in H_{\mathcal{C}}$ with low population risk $\ell_{\mathcal{D}}(\mathbf{w}, c) = \ell_{\mathcal{D}}(h_{\mathbf{w},c}) = \mathbb{E}_{(x,y)\sim\mathcal{D}}\ell(h_{\mathbf{w},c}(x), y)$ for loss $\ell : \mathcal{Y}' \times \mathcal{Y} \mapsto \mathbb{R}$ and some distribution $\mathcal{D}$ over sample space $\mathcal{X} \times \mathcal{Y}$.

We studied the empirical risk minimization problem (ERM) for a dataset of labeled examples in Chapter 6.3. However, as mentioned before, most NAS algorithms split the data into training and validation sets $T, V \subset \mathcal{X} \times \mathcal{Y}$ and instead solve the following bilevel optimization problem:

$$\min_{\mathbf{w}\in\mathcal{W}, c\in\mathcal{C}} \ \ell_V(\mathbf{w}, c) \quad \text{s.t.} \quad \mathbf{w} \in \arg\min_{\mathbf{u}\in\mathcal{W}} \ \mathcal{L}_T(\mathbf{u}, c) \tag{7.1}$$

where ℓ_V is the empirical risk over V while $\mathcal{L}_T$ is a (possibly) regularized objective over T. While the bilevel approach is very common in standard hyperparameter optimization and model selection [Franceschi et al., 2018], we will see that its widespread use in conjunction with weight-sharing is not an obvious decision. However, in Section 7.2.2 we show that this formulation can improve sample complexity via adaptation to the generalization error of the best configuration $c \in \mathcal{C}$ rather than to the average or the worst one, with explicit bounds on the excess risk for the case of feature map selection given in Corollaries 2 and 3.

7.1.1 The Unreasonable Effectiveness of Weight-Sharing

As we demonstrated in Chapter 5 and Chapter 6, despite the noise due to numerous SGD updates w.r.t. different architectures, weight-sharing has become an incredibly successful tool for NAS. The random search with weight-sharing (RSWS) approach introduced in Chapter 5 is a simple

example illustrating the surprising power of weight-sharing. The bilevel object solved by RSWS is effectively

$$\min_{c \in \mathcal{C}} \ \ell_V(\mathbf{w}^*, c) \quad \text{s.t.} \quad \mathbf{w}^* \in \arg\min_{\mathbf{u} \in \mathcal{W}} \ \mathop{\mathbb{E}}_{c \sim \text{Unif}(\mathcal{C})} \mathcal{L}_T(\mathbf{u}, c)$$

where the shared-weights are trained to optimize the expected performance of architectures sampled uniformly from $\mathcal{C}$. This shows how weight-sharing reduces the computational cost to that of training a single model, followed by (relatively cheap) evaluations of different configurations $c \in \mathcal{C}$ using the resulting shared weights $\mathbf{w}^*$. If the learned shared-weights $\mathbf{w}^*$ have high validation accuracy on the best architectures from the search space, then this approximation to the bilevel problem is useful for identifying good architectures. Surprisingly, this is the case for the two standard NAS benchmarks evaluated in Chapter 5.5, in Section 7.2 we observe analogous behavior for a much simpler hyperparameter configuration problem.

Instead of relying on uniform sampling, other NAS approaches attempt to adaptively learn a good distribution over configurations [Akimoto et al., 2019, Guo et al., 2019, Pham et al., 2018, Xie et al., 2019, Zheng et al., 2019]. For both problems that we study—feature map selection and federated hyperparameter tuning—we will see that simple distribution-updating schemes such as successive elimination or exponentiated gradient descent (multiplicative weights) are very effective.

7.1.2 Limitations and Questions

The above results highlight how shared weights are able to encode a remarkable amount of signal in over-parameterized NAS models and motivate our development of weight-sharing algorithms for other hyperparameter configuration problems. However, in addition to the inherent noisiness of the process, weight-sharing is limited to only providing signal for architectural hyperparameters; for a fixed set of weights, settings such as learning rate and regularization cannot be tuned because changing them does not affect the function being computed and thus leaves the loss unchanged. Although an important drawback, in Section 7.3 we show how it can be circumvented in the setting of federated learning with personalization.

This limitation also brings into question why most modern NAS methods employ weight-sharing to solve a bilevel optimization problem. Indeed, recent work has questioned this trend [Li et al., 2019b, Xie et al., 2019] since it seems perfectly justified to use the simpler single-level (regularized) ERM to directly inform the choice of architectural hyperparameters. In the next section, we use our study of feature map selection, a hyperparameter configuration problem that can be viewed as the simplest form of NAS, to provide concrete excess risk bounds as part of a broader theory of why bilevel optimization has been found to lead to better generalization in practice.

7.2 Feature Map Selection

In this section, we demonstrate how weight-sharing can be applied as a hyperparameter optimization method to simple feature map selection problems. Our empirical results show that weight-sharing outperforms random search and successive halving (Chapter 2) when selecting

Algorithm 6: Feature map selection using successive halving with weight-sharing.

Input: training set T, validation set V, convex loss ℓ, set of feature map ϕ_c configurations $\mathcal{C}$, regularization parameter $\lambda > 0$

for *round* $t = 1, \ldots, \log_2 |\mathcal{C}|$ **do**
  for *datapoint* $(x, y) \in T \cup V$ **do**
    assign $c_x \sim \text{Unif}(\mathcal{C})$ i.i.d.
  end
  $\mathbf{w}_t^* \leftarrow \arg\min_{\mathbf{w} \in \mathbb{R}^d} \ \lambda \|\mathbf{w}\|_2^2 + \sum_{(x,y) \in T} \ell(\langle \mathbf{w}, \phi_{c_x}(x) \rangle, y)$
  for *configuration* $c \in \mathcal{C}$ **do**
    $V_c \leftarrow \{(x, y) \in V : c_x = c\}$
    $s_c \leftarrow \frac{1}{|V_c|} \sum_{(x,y) \in V_c} \ell(\langle \mathbf{w}_t^*, \phi_{c_x}(x) \rangle, y)$
  end
  $\mathcal{C} = \{c \in \mathcal{C} : s_c \leq \text{Median}(\{s_c : c \in \mathcal{C}\})\}$
end

Result: Singleton set of configurations $\mathcal{C}$.

random Fourier features for CIFAR-10, motivating further analysis. We use this simple setting to help us understand the benefit of solving the bilevel problem for generalization.

In feature map selection we have a small set $\mathcal{C}$ of configurations c each corresponding to some feature map $\phi_c : \mathcal{X} \mapsto \mathbb{R}^d$ of the input that is to be passed to a linear classifier in $\mathcal{W} = \mathbb{R}^d$; the hypothesis space is then $H_{\mathcal{C}} = \{\langle \mathbf{w}, \phi_c(\cdot) \rangle : \mathbf{w} \in \mathbb{R}^d, c \in \mathcal{C}\}$. Observe that feature map selection can be viewed as a very simple NAS problem, with the difference that in neural networks the feature maps ϕ_c also depend on parameter-weights $\mathbf{w}$, whereas here we only parameterize the last layer.

For standard hyperparameter optimization, feature map selection is done via the following bilevel optimization for regularization parameter $\lambda > 0$:

$$\begin{aligned} &\min_{c \in \mathcal{C}} \quad \sum_{(x,y) \in V} \ell(\langle \mathbf{w}_c^*, \phi_c(x) \rangle, y) \\ &\mathbf{w}_c^* \ = \ \arg\min_{\mathbf{w} \in \mathbb{R}^d} \ \lambda \|\mathbf{w}\|_2^2 \ + \sum_{(x,y) \in T} \ell(\langle \mathbf{w}, \phi_c(x) \rangle, y) \end{aligned} \tag{7.2}$$

Note that λ is usually part of $\mathcal{C}$, but as a non-architectural hyperparameter it is not tuned by weight-sharing. [1]

7.2.1 Weight-Sharing for Feature Map Selection

How can we use weight-sharing to approximate Eq. (7.2) without solving a Ridge regression problem to obtain $\mathbf{w}_c^*$ for each configuration $c \in \mathcal{C}$? As with most hyperparameter optimization

[1] We do not find that including regularization as a hyperparameter significantly affects performance of random search. This is perhaps not surprising as architectural parameters such as kernel bandwidth and feature weighting play similar roles in this setting.

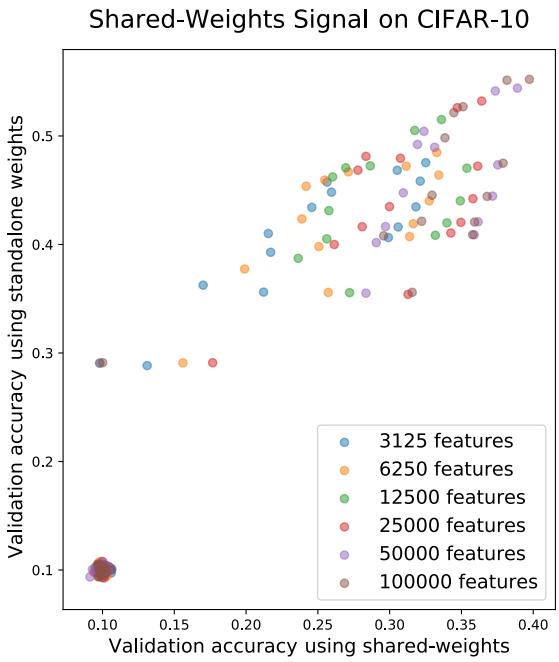


Figure 7.1: CIFAR-10 validation accuracy of each feature map (random Fourier kernel approximation) when using the jointly trained classifier (shared weights after a single round of training) compared to its validation accuracy when using its own independently trained classifier (standalone weights). The accuracies given by the shared-weights classifier correlate strongly with those of individual classifiers, providing a strong signal for selecting feature maps without solving Ridge regression many times.

problems, we need a method for resource allocation—a way to prioritize different configurations based on validation data. We take inspiration from the RSWS algorithm described in Section 7.1, which allocates a resource—a minibatch of training examples—to some configuration at each iteration. Analogously, in Algorithm 6, we propose to allocate training examples to feature maps: at each iteration t we solve an ℓ_2-regularized ERM problem in which each point is featurized by a random map $\phi_c, c \sim \mathrm{Unif}(\mathcal{C})$. The resulting classifier $\mathbf{w}_t$ is thus *shared* among the feature maps rather than being the minimizer for any single one; nevertheless, as with RSWS we find that, despite being trained using the uniform distribution over configurations, the shared-weights $\mathbf{w}_t$ are much better classifiers for data featurized using the best feature maps (c.f. Figure 7.1). We use this as validation signal in a successive halving procedure that approximates Eq. (7.2) in $\log_2 |\mathcal{C}|$ regression solves.

We evaluate Algorithm 6 on the feature map selection problem of kernel ridge regression over random Fourier features [Rahimi and Recht, 2007] on CIFAR-10. In addition to the weight-sharing approach described in Algorithm 6, we evaluate a variant we call *Fast Weight-Sharing* in which at each round t the feature dimension used is a constant multiplicative factor greater than that used on the previous round (we use $\sqrt{2}$, finding doubling to be too aggressive), with the final dimension d reached only at the last round. This is reminiscent of the successive halving resource allocation scheme since the promoted feature maps are given multiplicatively more random features. Following the experiments in Chapter 2.4.3, we tune the bandwidth, the preprocessing,

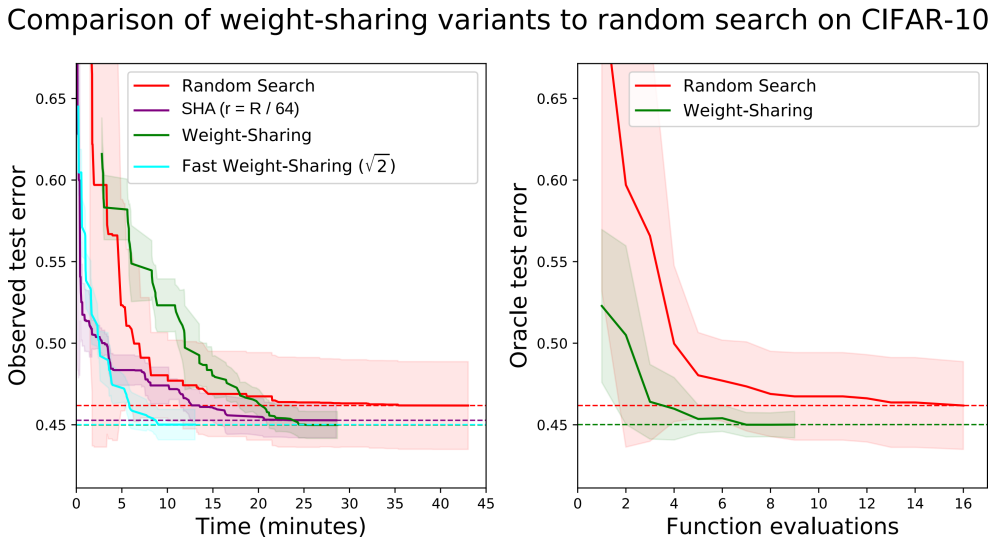


Figure 7.2: Observed test-error on CIFAR-10 as a function of time (left) and *oracle* test-error on CIFAR-10 as a function of number of calls to the Ridge regression solver. Here, *oracle* test-error refers to evaluation of a separately trained, non-weight-shared, classifier on the best config at any given round according to weight-sharing. All curves are averaged over 10 independent trials.

and replace their tuning of the regularization λ by tuning the choice of kernel instead; further details are provided in Section 7.5. For all experiments we set the maximum number of features to be $d = 100000$. We set the minimum number of features for successive halving to be 6250 for a total of 5 rounds. For random search, we continue sampling configurations until the cumulative number of features evaluated matches that of successive halving.

We find that both weight-sharing algorithms consistently outperform or match the performance of random search and successive halving in both speed and accuracy. In Figure 7.2 we show that weight-sharing based methods achieve lower observed test-error than the baseline methods. While successive halving and random search converge more quickly (to slightly less optimal configurations) in terms of wall-clock time, fast weight-sharing converges to the same best observed test-error as weight-sharing in significantly less time. We also find that, while random search appears to converge faster than weight-sharing in terms of observed test-error measured on shared weights over wall-clock time, weight-sharing converges faster in terms of *oracle* test-error over the number of function evaluations. The lower observed test error during intermediate rounds of weight-sharing is due the shared classifiers not being optimal for any single configuration. Note that we do not compare to SHA or fast weight-sharing in the right-hand plot because they compute many more solutions to lower-dimensional Ridge regression problems and so the evaluations are incomparable.

7.2.2 Generalization Guarantees for the Bilevel Problem

Bilevel objectives such as Eq. (6.3) and Eq. (7.2) are closely related to the well-studied problems of *model selection* and *cross-validation*. However, a key difference in our case is that the choice of configuration $c \in \mathcal{C}$ does not necessarily provide any control over the complexity of the hypothesis

space; for example, in NAS it is often unclear how the hypothesis space changes due to the change in one decision. By contrast, the theory of model selection is often directly concerned with control of model complexity. Indeed, in possibly the most common setting the hypothesis classes are nested according to some total order of increasing complexity, forming a structure [Kearns et al., 1997, Vapnik, 1982]. This is for example the case in most norm-based regularization schemes. Even in the non-nested case, there is often an explicit tradeoff between parsimony and accuracy [Vuong, 1989].

On the other hand, in our feature map experiments we do not tune the one explicit capacity-controlling parameter, λ, because it is non-architectural. This brings into question why we need the bilevel formulation in the first place; instead we could still use weight-sharing but solve the joint ERM problem over the combined data:

$$\min_{c\in\mathcal{C},\mathbf{w}\in\mathcal{W}} \quad \lambda\|\mathbf{w}\|_2^2 \quad + \sum_{(x,y)\in T\cup V} \ell(\langle \mathbf{w}, \phi_c(x)\rangle, y)$$

We are especially interested in this question due to the widespread use of the bilevel formulation in NAS with weight-sharing, despite the identical limitations on tuning non-architectural parameters discussed in Section 7.1.2. Especially when continuous relaxation [Liu et al., 2019] is applied, architecture parameters in NAS appear more similar to regular model parameters rather than controls on the model complexity, so it becomes reasonable to wonder why most NAS practitioners have used the bilevel formulation. In this section we give an analysis suggesting that decomposing the objective can improve generalization by adapting to the sample complexity of the best configuration in $\mathcal{C}$, whereas ERM suffers that of the worst. For feature map selection our theory gives concrete excess risk bounds.

To see why the bilevel formulation Eq. (6.3) might be useful, we first note that a key aspect of the optima of the bilevel problem is the restriction on the model weights: they must be in the set $\arg\min_{\mathbf{w}\in\mathcal{W}} \mathcal{L}_T(\mathbf{w}, c)$ of the inner objective $\mathcal{L}_T$. We refer to the subset $H_{c,T} = \{h_{\mathbf{w},c} : \mathbf{w} \in \arg\min_{\mathbf{u}\in\mathcal{W}} \mathcal{L}_T(\mathbf{u}, c)\}$ of the associated hypotheses associated as the *version space* [Kearns et al., 1997, Equation 6] induced by some configuration c and the objective function. Then letting $N(F, \varepsilon)$ be the L^∞*-covering-number* of a set of functions F at scale $\varepsilon > 0$, i.e. the number of L^∞ balls required to construct an ε-cover of F [Mohri et al., 2012, Equation 3.60], we define the following empirical complexity measure:

Definition 7. *The* **version entropy** *of $H_{\mathcal{C}}$ at scale $\varepsilon > 0$ induced by the objective $\mathcal{L}_T$ over training data T is $\Lambda(H, \varepsilon, T) = \log N\left(\bigcup_{c\in\mathcal{C}} H_{c,T}, \varepsilon\right)$.*

The version entropy is a data-dependent quantification of how much the hypothesis class is restricted by the inner optimization. For finite $\mathcal{C}$, a naive bound shows that $\Lambda(H, \varepsilon, T)$ is bounded by $\log|\mathcal{C}| + \max_{c\in\mathcal{C}} \log N(H_{c,T}, \varepsilon)$, so that the second term measures the worst-case complexity of the global minimizers of $\mathcal{L}_T$. In the feature selection problem, $\mathcal{L}_T$ is usually a strongly-convex loss due to regularization and so all version spaces are singleton sets, making the version entropy $\log|\mathcal{C}|$. In the other extreme case of nested model selection the version entropy reduces to the complexity of the version space of the largest model and so may not be informative. However, in practical problems such as NAS an inductive bias is often imposed via constraints on the number of input edges.

To bound the excess risk in terms of the version entropy, we first discuss an important assumption describing cases when we expect the bilevel approach to perform well:

Assumption 2. *There exists a good $c^* \in \mathcal{C}$—one such that $(\mathbf{w}^*, c^*) \in \arg\min_{\mathcal{W}\times\mathcal{C}} \ell_{\mathcal{D}}(\mathbf{w}, c)$ for some $\mathbf{w}^* \in \mathcal{W}$—such that w.h.p. over the drawing of training set T at least one of the minima of the optimization induced by c^* and T has low excess risk, i.e. w.p. $1-\delta$ there exists $\mathbf{w} \in \arg\min_{\mathbf{u}\in\mathcal{W}} \mathcal{L}_T(\mathbf{u}, c^*)$ s.t. $\ell_{\mathcal{D}}(h_{\mathbf{w},c^*}) - \ell_{\mathcal{D}}(h^*) \leq \varepsilon^*(|T|, \delta)$ for excess risk ε^* and all $h^* \in H_{\mathcal{C}}$.*

This assumption requires that w.h.p. the inner optimization objective does not exclude all low-risk classifiers for the optimal configuration. Note that it asks nothing of either the other configurations in $\mathcal{C}$, which may be arbitrarily bad, nor of the hypotheses found by the procedure. It does however prevent the case where even minimizing the objective $\mathcal{L}_T(\cdot, c^*)$ does not provide a set of good weights. Note that if the inner optimization is simply ERM over the training set T, i.e. $\mathcal{L}_T = \ell_T$, then standard learning-theoretic guarantees will give $\varepsilon^*(|T|, \delta)$ decreasing in the size $|T|$ of the training set and increasing at most poly-logarithmically in $\frac{1}{\delta}$. With this assumption, we can show the following guarantee for solutions to the bilevel optimization Eq. (6.3):

Theorem 3. *Let $\hat{h} = h_{\mathbf{w},c}$ be a hypothesis corresponding to a solution $(\mathbf{w}, c) \in \mathcal{W}\times\mathcal{C}$ of the bilevel objective:*

$$\min_{\mathbf{w}\in\mathcal{W}, c\in\mathcal{C}} \ \ell_V(\mathbf{w}, c) \quad \text{s.t.} \quad \mathbf{w} \in \arg\min_{\mathbf{u}\in\mathcal{W}} \ \mathcal{L}_T(\mathbf{u}, c)$$

for space $H_{\mathcal{C}}$ and data V, T, as in Section 7.1. Under Assumption 2 if ℓ is B-bounded then w.p. $1-3\delta$ we have

$$\ell_{\mathcal{D}}(\hat{h}) \leq \min_{h\in H_{\mathcal{C}}} \ell_{\mathcal{D}}(h) + \varepsilon^*(|T|, \delta) + \inf_{\varepsilon>0} 3\varepsilon + \frac{3B}{2}\sqrt{\frac{2}{|V|}\left(\Lambda(H, \varepsilon, T) + \log\frac{1}{\delta}\right)}$$

Proof Sketch. For all $h^* \in H_{\mathcal{C}}$ decompose the excess risk $\ell_{\mathcal{D}}(\hat{h}) - \ell_D(h^*)$ as

$$\begin{aligned} &\ell_{\mathcal{D}}(\hat{h}) - \ell_V(\hat{h}) \quad + \quad \ell_V(\hat{h}) - \ell_V(h_{\mathbf{w},c^*}) \\ &+ \quad \ell_V(h_{\mathbf{w},c^*}) - \ell_{\mathcal{D}}(h_{\mathbf{w},c^*}) \quad + \quad \ell_{\mathcal{D}}(h_{\mathbf{w},c^*}) - \ell_{\mathcal{D}}(h^*) \end{aligned}$$

The first difference is bounded by version entropy via $\hat{h} \in H_c$, the second by optimality of $\hat{h} \in V$, the third by Hoeffding's inequality, and the last by Assumption 2. □

7.2.2.1 Excess Risk of Feature Map Selection

To obtain meaningful guarantees from this theorem we must bound the version entropy. By strong-convexity of the inner objective, in feature map selection each ϕ_c induces a unique minimizing weight $\mathbf{w} \in \mathcal{W}$ and thus a singleton version space, so the version entropy is bounded by $\log|\mathcal{C}|$. Furthermore, for Lipschitz (e.g. hinge) losses and appropriate regularization, standard results for ℓ_2-regularized ERM for linear classification (e.g. Sridharan et al. [2008]) show that Assumption 2 holds for $\varepsilon^*(|T|, \delta) = O\left(\sqrt{\frac{\|\mathbf{w}^*\|_2^2+1}{|T|}\log\frac{1}{\delta}}\right)$. Applying Theorem 3 yields

Corollary 2. *For feature map selection the bilevel optimization Eq. (7.2) with $\lambda = \sqrt{\frac{1}{|T|}\log\frac{1}{\delta}}$ yields a hypothesis with excess risk w.r.t. $H_{\mathcal{C}}$ bounded by $\mathcal{O}\left(\sqrt{\frac{\|\mathbf{w}^*\|_2^2+1}{|T|}\log\frac{1}{\delta}} + \sqrt{\frac{1}{|V|}\log\frac{|\mathcal{C}|}{\delta}}\right)$.*

In the special case of selection random Fourier approximations of kernels, we can apply associated generalization guarantees [Rudi and Rosasco, 2017, Theorem 1] to show that we can compete with the optimal RKHS from among those associated with one of the configurations :

Corollary 3. *In feature map selection suppose each map $\phi_c, c \in \mathcal{C}$ is associated with a random Fourier feature approximation of a continuous shift-invariant kernel that approximates an RKHS $\mathcal{H}_\phi$ and ℓ is the square loss. If the number of features $d = \mathcal{O}(\sqrt{|T|}\log|T|/\delta)$ and $\lambda = 1/\sqrt{|T|}$ then w.p. $1-\delta$ solving Eq. (7.2) yields a hypothesis with excess risk w.r.t. $\mathcal{H}_\phi$ bounded by* $\mathcal{O}\left(\frac{\log^2\frac{1}{\delta}}{\sqrt{|T|}} + \sqrt{\frac{1}{|V|}\log\frac{|\mathcal{C}|}{\delta}}\right)$.

In this case we are able to get risk bounds almost identical to the excess risk achievable if we knew the optimal configuration beforehand, up to an additional term depending weakly on the number of configurations. This would not be possible with solving the regular ERM objective instead of the bilevel optimization as we would then have to contend with the possibly high complexity of the hypothesis space induced by the worst configuration.

7.2.2.2 Version Entropy and NAS

As shown above, the significance of Theorem 3 in simple settings is that a bound on the version entropy can guarantee excess risk almost as good as that of the (unknown) optimal configuration without assuming anything about the complexity or behavior of sub-optimal configurations. In the case of NAS we do not have a bound on the version entropy, which now depends on all of $\mathcal{C}$. Whether the version space, and thus the complexity, of deep networks is small compared to the number of samples is unclear, although we gather some evidence below. The question amounts to how many (functional) global optima are induced by a training set of size $|T|$. In an idealized spin-glass model, Choromanska et al. [2015, Theorem 11.1] suggest that the number of critical points is exponential *only* in the number of layers, which would yield a small version entropy. It is conceivable that the quantity may be further bounded by the complexity of solutions explored by the algorithm when optimizing $\mathcal{L}_T$ [Bartlett et al., 2017, Nagarajan and Kolter, 2017]. On the other hand, Nagarajan and Kolter [2019] argue, with evidence in restricted settings, that even the most stringent implicit regularization cannot lead to a non-vacuous uniform convergence bound; if true more generally this would imply that the NAS version entropy is quite large.

7.3 Federated Hyperparameter Optimization

Having established the effectiveness of weight-sharing for feature map selection, we now argue for its application to more difficult and larger-scale configuration problems. We focus on the problem of hyperparameter optimization in the federated learning setting, where it remains a major challenge [Kairouz et al., 2019]. In particular, because of the ephemeral nature of multi-device learning, in which the data is bound to the device and the number of training rounds is limited due to computation and communication constraints, running standard cross-validation and more sophisticated hyperparameter optimization approaches necessarily involves making use of subsets of devices to train individual configurations. Hence, with existing methods, we cannot update all configurations using data from all devices, which can be problematic when data is non-i.i.d

Algorithm 7: Federated Exponentiated-Gradient (FedEx) algorithm for federated hyperparameter optimization.

Input: model initialization w, initial distribution $\mathcal{D}_\theta(\mathcal{C})$ over configurations $\mathcal{C}$, configuration learning rates $\eta_t > 0$

for *round* $t = 1, \dots, T$ **do**
  for *device* $k = 1, \dots, B$ **do**
    send $(\mathbf{w}, \theta)$ to device
    $c \sim \mathcal{D}_\theta(\mathcal{C})$ // each device gets a random configuration c
    $\hat{\mathbf{w}}_k \leftarrow \mathrm{SGD}_{\mathbf{w},c}(T_{tk})$ // run local SGD from $\mathbf{w}$ with configuration c
    get $(\hat{\mathbf{w}}_k, \hat{g}_k)$ from device, where $\mathbb{E}_{c|\theta}\hat{g}_k = \nabla_\theta \mathbb{E}_{c|\theta} F_{V_{tk}}(\hat{\mathbf{w}}_k)$ // stochastic gradient w.r.t. θ
  end
  $\mathbf{w} \leftarrow \sum_{k=1}^B |T_{tk}|\hat{\mathbf{w}}_k / \sum_{k=1}^B |T_{tk}|$ // FedAvg update
  $\theta \leftarrow \theta \odot \exp\left(-\eta_t \sum_{k=1}^B |V_{tk}|\hat{g}_k / \sum_{k=1}^B |V_{tk}|\right)$ // exponentiated-gradient update
  $\theta \leftarrow \theta / \|\theta\|_1$ // project back to the probability simplex
end

Result: model initialization $\mathbf{w}$ and configuration $c \sim \mathcal{D}_\theta$ of local SGD for on-device personalization.

across devices. On the other hand, if we treat each device as a data-point to be allocated to some configuration in order to make a local update, weight-sharing naturally aligns with the dominant method in federated learning, Federated Averaging FedAvg, in which at each communication round multiple devices run local SGD from a shared initialization before averaging the output [McMahan et al., 2017].

7.3.1 Tuning the Hyperparameters of FedAvg

In federated learning we have a set of clients $k = 1, \dots, K$ each with a training dataset $T_k \subset \mathcal{X} \times \mathcal{Y} = \mathcal{Z}$ consisting of examples $z \in \mathcal{Z}$ corresponding to functions $f_z : \mathbb{R}^d \mapsto \mathbb{R}_+$, over the parameter space $\mathbf{w} \in \mathbb{R}^d$ of some prediction function. The standard optimization problem [Li et al., 2019c] is then of form

$$\min_{\mathbf{w} \in \mathcal{W}} \quad \sum_{k=1}^K |T_k| F_{T_k}(w) \quad \text{for} \quad F_T(\mathbf{w}) = \frac{1}{|T|} \sum_{z \in T} f_z(\mathbf{w})$$

Optimization is a major challenge in federated learning due to non-i.i.d. ephemeral data, limited communication, and relatively weak devices [Kairouz et al., 2019, Li et al., 2019c]. As a result, many of the hyperparameters tuned by practitioners, such as the learning rate, on-device batch-size, and number of epochs per-device, are algorithmic rather than architectural [McMahan et al., 2017]. Following the discussion in Section 7.1.2, this brings into question the applicability of weight-sharing, which relies on validation signal that cannot be directly obtained for such hyperparameters.

To resolve this we take advantage of the recent re-interpretation of federated learning with personalization over non-i.i.d. data as a meta-learning problem [Chen et al., 2019, Jiang et al., 2019, Khodak et al., 2019, Li et al., 2019b], with the goal being to learn a shared initialization $\mathbf{w} \in \mathbb{R}^d$ and configuration c for $\text{SGD}_{\mathbf{w},c} : 2^{\mathcal{Z}} \mapsto \mathbb{R}^d$, an algorithm that takes training data $T \subset \mathcal{Z}$ as input and outputs a model parameter. We can then construct a *single-level* federated hyperparameter optimization objective:

$$\min_{\mathbf{w} \in \mathcal{W}, c \in \mathcal{C}} \quad \sum_{k=1}^{K} |V_k| F_{V_k}(\text{SGD}_{\mathbf{w},c}(T_k)) \tag{7.3}$$

Thus the formal goal is to learn a model initialization $\mathbf{w} \in \mathbb{R}^d$ and an SGD configuration $c \in \mathcal{C}$ such that on any device with training/validation data T/V, if we run $\text{SGD}_{\mathbf{w},c}$ initialized at $\mathbf{w}$ using samples T then the validation loss of the resulting parameter over V will be small. Here the shared weights correspond to the SGD initialization, which is used to evaluate multiple configurations of SGD per round via the on-device validation accuracy after local training. Note that although we split on-device data into training and validation sets, the objective is single level since here devices are data-points with loss corresponding to the error on its validation data of the model obtained by running local SGD. This highlights the Section 7.1.2 point that weight-sharing is not inherently tied to the bilevel formulation, although our theory does not hold for this setting.

In Algorithm 7 we use this formulation to develop FedEx, a weight-sharing extension of FedAvg [McMahan et al., 2017]. Note that the update to the model parameters w is exactly the same, except that local SGD may be run using a different configuration on different devices. Following the approach used by stochastic relaxation methods for differentiable NAS [Akimoto et al., 2019], we note that the goal of learning $c \in \mathcal{C}$ is equivalent to learning a parameterization θ of a categorical distribution $\mathcal{D}_\theta(\mathcal{C})$ over the configuration space, where if $c \sim \mathcal{D}_\theta(\mathcal{C})$ then $\mathbb{P}_\theta(c = i) = \theta_i$. We can then use evaluations of these configurations, in the form of the per-device validation accuracies $F_{V_{tk}}(\hat{\mathbf{w}}_k)$ of models $\hat{\mathbf{w}}_k$ trained locally using the shared initialization $\mathbf{w}$, to obtain a stochastic estimate of the gradient using the re-parameterization trick [Rubinstein and Shapiro, 1993]. Specifically, we can write the expected value of the gradient for a categorical architecture distribution with a baseline objective value λ as follows:

$$\begin{aligned}
\nabla_{\theta_{ij}} \mathbb{E}_{c|\theta} F_{V_{tk}}(\hat{w}_k) &= \nabla_{\theta_{ij}} \mathbb{E}_{c|\theta} (F_{V_{tk}}(\hat{w}_k) - \lambda) \\
&= \mathbb{E}_{c|\theta} \left((F_{V_{tk}}(\hat{\mathbf{w}}_k) - \lambda) \nabla_{\theta_{ij}} \log \mathbb{P}_\theta(c) \right) \\
&= \mathbb{E}_{c|\theta} \left((F_{V_{tk}}(\hat{\mathbf{w}}_k) - \lambda) \nabla_{\theta_{ij}} \log \prod_{i=1}^{n} \mathbb{P}_\theta(c_i = j) \right) \\
&= \mathbb{E}_{c|\theta} \left((F_{V_{tk}}(\hat{\mathbf{w}}_k) - \lambda) \sum_{i=1}^{n} \nabla_{\theta_{ij}} \log \mathbb{P}_\theta(c_i = j) \right) \\
&= \mathbb{E}_{c|\theta} \left(\frac{(F_{V_{tk}}(\hat{\mathbf{w}}_k) - \lambda) 1_{c_i = j}}{\theta_{ij}} \right) \\
&= \mathbb{E}_{c|\theta} \hat{g}_{kij}.
\end{aligned}$$

Hence, the stochastic gradient $\hat{g}_{ki}$ w.r.t. θ_i of the kth device on round t

$$\hat{g}_{ki} = \frac{(F_{V_{tk}}(\hat{\mathbf{w}}_k) - \lambda)1_{c=i}}{\theta_i}$$

is an unbiased estimate of the architecture gradient. As with weight-sharing NAS, our algorithm alternates between updating this distribution using exponentiated-gradient descent and updating the shared-weights using the configured SGD algorithm on each device.

7.3.2 Ongoing Work

As part of ongoing work, we are evaluating FedEx and ASHA on the federated learning benchmarks available within the LEAF framework [Caldas et al., 2018]. The two benchmarks of interest include a federated version of MNIST and a language-modeling task over the Shakespeare dataset. While CNNs and LSTMs [Hochreiter and Schmidhuber, 1997] have seen heavy development for the respective benchmarks, they still require a great deal of hyperparameter tuning for parameters such as learning rate, weight-decay, batch size, and dropout probability. Our empirical evaluation aims to compare the performance of automated hyperparameter tuning with FedEx and ASHA relative to that of manual-tuning [McMahan et al., 2017].

7.4 Generalization Results

This section contains proofs of the generalization results in Section 7.2.2.

7.4.1 Settings and Main Assumption

We first describe the setting for which we prove our general result.

Setting 2. *Let $\mathcal{C}$ be a set of possible architecture/configurations of finite size such that each $c \in \mathcal{C}$ is associated with a parameterized hypothesis class H_c=$\{h_{w,c} : \mathcal{X} \mapsto \mathcal{Y}' : w \in \mathcal{W}\}$ for input space $\mathcal{X}$, output space $\mathcal{Y}'$, and fixed set of possible weights $\mathcal{W}$. We will measure the performance of a hypothesis $h_{w,c}$ on an input $x, y \in \mathcal{X} \times \mathcal{Y}$ for some output space $\mathcal{Y}$ using a B-bounded loss function $\ell : \mathcal{Y}' \times \mathcal{Y} \mapsto [0, B]$. Note that while the examples below have unbounded loss functions, in practice they are explicitly or implicitly bounded by explicit or implicit regularization.*

We are given a training sample $T \sim \mathcal{D}^{|T|}$ and a validation sample $V \sim \mathcal{D}^{|V|}$, where $\mathcal{D}$ is some distribution over $\mathcal{X} \times \mathcal{Y}$. We will denote the the population risk by $\ell_{\mathcal{D}}(h_{w,c}) = \ell_{\mathcal{D}}(w, c) = \mathbb{E}_{(x,y)\sim\mathcal{D}}\ell(h_{w,c}(x), y)$ and for any finite subset $S \subset \mathcal{X} \times \mathcal{Y}$ we will denote the empirical risk over S by $\ell_S(h_{w,c}) = \ell_S(w, c) = \frac{1}{|S|}\sum_{(x,y)\in S}\ell(h_{w,c}(x), y)$.

Finally, we will consider solutions of optimization problems that depend on the training data and architecture. Specifically, for any configuration $c \in \mathcal{C}$ and finite subset $S \subset \mathcal{X} \times \mathcal{Y}$ let $\mathcal{W}_c(S) \subset \mathcal{W}$ be the set of global minima of some optimization problem induced by S and c and let the associated version space [Kearns et al., 1997] be $H_c(S) = \{h_{w,c} : w \in \mathcal{W}_c(S)\}$.

We next give as examples two specific settings encompassed by Setting 2.

Setting 3. *For* **feature map selection***, in Setting 2 the configuration space $\mathcal{C}$ is a set of feature maps $\phi_c : \mathcal{X} \mapsto \mathbb{R}^d$, the set of weights $\mathcal{W} \subset \mathbb{R}^d$ consists of linear classifiers, for inputs $x \in \mathcal{X}$ the hypotheses are $h_{w,c}(x) = \langle w, \phi_c(x) \rangle$ for $w \in \mathcal{W}$, and so $\mathcal{W}_c(S)$ is the singleton set of solutions to the regularized ERM problem*

$$\arg\min_{w \in \mathcal{W}} \quad \lambda \|w\|_2^2 \quad + \sum_{(x,y) \in S} \ell(\langle w, \phi_c(x) \rangle, y)$$

for square loss $\ell : \mathcal{Y}' \times \mathcal{Y} \mapsto \mathbb{R}_+$ and some coefficient $\lambda > 0$.

Setting 4. *For* **neural architecture search***, in Setting 2 the configuration space consists of all possible choices of edges on a DAG of N nodes and a choice from one of K operations at each edge, for a total number of configurations bounded by K^{N^2}. The hypothesis $h_{w,c} : \mathcal{X} \mapsto \mathcal{Y}'$ is determined by a choice of architecture $c \in \mathcal{C}$ and a set of network weights $w \in \mathcal{W}$ and the loss $\ell : \mathcal{Y}' \times \mathcal{Y} \mapsto \mathbb{R}_+$ is the cross-entropy loss. In the simplest case $\mathcal{W}_c(S)$ is the set of global minima of the ERM problem*

$$\min_{w \in \mathcal{W}} \quad \sum_{(x,y) \in S} \ell(h_{w,c}(x), y)$$

We now state the main assumption we require.

Assumption 3. *In Setting 2 there exists a good architecture $c^* \in \mathcal{C}$, i.e. one satisfying $(w^*, c^*) \in \arg\min_{\mathcal{W} \times \mathcal{C}} \ell_{\mathcal{D}}(w, c)$ for some weights $w^* \in \mathcal{W}$, such that w.p. $1 - \delta$ over the drawing of training set $T \sim \mathcal{D}^{|T|}$ at least one of the minima of the optimization problem induced by c^* and T has low excess risk, i.e. $\exists\, w \in \mathcal{W}_{c^*}(T)$ s.t.*

$$\ell_{\mathcal{D}}(w, c^*) - \ell_{\mathcal{D}}(w^*, c^*) \leq \varepsilon^*(|T|, \delta) \tag{7.4}$$

for some error function ε^*.

Clearly, we prefer error functions ε^* that are decreasing in the number of training samples $|T|$ and increasing at most poly-logarithmically in $\frac{1}{\delta}$. This assumption requires that if we knew the optimal configuration *a priori*, then the provided optimization problem will find a good set of weights for it. We will show how, under reasonable assumptions, Assumption 3 can be formally shown to hold in Settings 3 and 4.

7.4.2 Main Result

Our general result will be stated in terms of covering numbers of certain function classes.

Definition 8. *Let H be a class of functions from $\mathcal{X}$ to $\mathcal{Y}'$. For any $\varepsilon > 0$ the associated L^∞* **covering number** *$N(H, \varepsilon)$ of H is the minimal positive integer k such that H can be covered by k balls of L^∞-radius ε.*

The following is then a standard result in statistical learning theory (see e.g. Lafferty et al. [2010, Theorem 7.82]):

Theorem 4. *Let H be a class of functions from $\mathcal{X}$ to $\mathcal{Y}$ and let $\ell : \mathcal{Y}' \times \mathcal{Y} \mapsto [0, B]$ be an L-Lipschitz, B-bounded loss function. Then for any distribution $\mathcal{D}$ over $\mathcal{X} \times \mathcal{Y}$ we have*

$$\Pr_{S\sim\mathcal{D}^m}\left(\sup_{h\in H}|\ell_{\mathcal{D}}(h) - \ell_S(h)| \geq 3\varepsilon\right) \leq 2N\left(H, \varepsilon\right)\exp\left(-\frac{m\varepsilon^2}{2B^2}\right)$$

where we use the loss notation from Setting 2.

Before stating our theorem, we define a final quantity, which measures the log covering number of the version spaces induced by the optimization procedure over a given training set.

Definition 9. *In Setting 2, for any sample $S \subset \mathcal{X} \times \mathcal{Y}$ define the* **version entropy** *to be $\Lambda(H, \varepsilon, S) = \log N\left(\bigcup_{c\in\mathcal{C}} H_c(S), \varepsilon\right)$.*

Theorem 5. *In Setting 2 let $(\hat{w}, \hat{c}) \in \mathcal{W}\times\mathcal{C}$ be obtained as a solution to the following optimization problem:*

$$\underset{w\in\mathcal{W}, c\in\mathcal{C}}{\arg\min} \quad \ell_V(w, c) \qquad \textit{s.t.} \qquad w \in \mathcal{W}_c(T)$$

Then under Assumption 3 we have w.p. $1 - 3\delta$ that

$$\begin{aligned}\ell_{\mathcal{D}}(\hat{w}, \hat{c}) &\leq \ell_{\mathcal{D}}(w^*, c^*) \\ &\quad + \varepsilon^*(|T|, \delta) + B\sqrt{\frac{1}{2|V|}\log\frac{1}{\delta}} + \inf_{\varepsilon>0} 3\varepsilon + B\sqrt{\frac{2}{|V|}\left(\Lambda(H, \varepsilon, T) + \log\frac{1}{\delta}\right)}\end{aligned}$$

Proof. We have for any $w \in \mathcal{W}_{c^*}(T)$ satisfying Equation 7.4, whose existence holds by Assumption 3, that

$$\begin{aligned}\ell_{\mathcal{D}}(\hat{w}, \hat{c}) - \ell_{\mathcal{D}}(w^*, c^*) \quad \leq \quad & \underbrace{\ell_{\mathcal{D}}(\hat{w}, \hat{c}) - \ell_V(\hat{w}, \hat{c})}_{1} \quad + \quad \underbrace{\ell_V(\hat{w}, \hat{c}) - \ell_V(w, c^*)}_{2} \\ & + \quad \underbrace{\ell_V(w, c^*) - \ell_{\mathcal{D}}(w, c^*)}_{3} \quad + \quad \underbrace{\ell_{\mathcal{D}}(w, c^*) - \ell_{\mathcal{D}}(w^*, c^*)}_{4}\end{aligned}$$

each term of which can be bounded as follows:

1. Since $\hat{w} \in \mathcal{W}_{\hat{c}}(T)$ for some $\hat{c} \in \mathcal{C}$ the hypothesis space can be covered by the union of the coverings of $H_c(T)$ over $c \in \mathcal{C}$, so by Theorem 4 we have that w.p. $1 - \delta$

$$\ell_{\mathcal{D}}(\hat{w}, \hat{c}) - \ell_V(\hat{w}, \hat{c}) \leq \inf_{\varepsilon>0} 3\varepsilon + B\sqrt{\frac{2}{|V|}\left(\Lambda(H, \varepsilon, T) + \log\frac{1}{\delta}\right)}$$

2. By optimality of the pair $(\hat{w}, \hat{c})$ and the fact that $w \in \mathcal{W}_{c^*}(T)$ we have

$$\ell_V(\hat{w}, \hat{c}) = \min_{c \in \mathcal{C}, w' \in \mathcal{W}_c(T)} \ell_V(w', \hat{c}) \leq \min_{w' \in \mathcal{W}_{c^*}(T)} \ell_V(w', c^*) \leq \ell_V(w, c^*)$$

3. Hoeffding's inequality yields $\ell_V(w, c^*) - \ell_{\mathcal{D}}(w, c^*) \leq B\sqrt{\frac{1}{2|V|} \log \frac{1}{\delta}}$ w.p. $1 - \delta$
4. Assumption 3 states that $\ell_{\mathcal{D}}(w, c^*) - \ell_{\mathcal{D}}(w^*, c^*) \leq \varepsilon^*(|T|\delta)$ w.p. $1 - \delta$.

□

7.4.3 Applications

For the feature map selection problem, Assumption 3 holds by standard results for ℓ_2-regularized ERM for linear classification (e.g. Sridharan et al. [2008]):

Corollary 4. *In Setting 3, suppose the loss function ℓ is Lipschitz. Then for regularization parameter $\lambda = \sqrt{\frac{1}{|T|} \log \frac{1}{\delta}}$ we have*

$$\ell_{\mathcal{D}}(w, c^*) - \ell_{\mathcal{D}}(w^*, c^*) \leq \mathcal{O}\left(\sqrt{\frac{\|w^*\|_2^2 + 1}{|T|} \log \frac{1}{\delta}}\right)$$

We can then directly apply Theorem 5 and the fact that the version entropy is bounded by $\log |\mathcal{C}|$ because the minimizer over the training set is always unique to get the following:

Corollary 5. *In Setting 3 let $(\hat{w}, \hat{c}) \in \mathcal{W} \times \mathcal{C}$ be obtained as a solution to the following optimization problem:*

$$\operatorname*{arg\,min}_{w \in \mathcal{W}, c \in \mathcal{C}} \quad \ell_V(w, c) \qquad \textit{s.t.} \qquad w = \operatorname*{arg\,min}_{w \in \mathcal{W}} \quad \lambda \|w\|_2^2 + \sum_{(x,y) \in T} \ell(\langle w, \phi_c(x) \rangle, y)$$

Then

$$\ell_{\mathcal{D}}(\hat{w}, \hat{c}) - \ell_{\mathcal{D}}(w^*, c^*) \leq \mathcal{O}\left(\sqrt{\frac{\|w^*\|_2^2 + 1}{|T|} \log \frac{1}{\delta}} + \sqrt{\frac{1}{|V|} \log \frac{|\mathcal{C}| + 1}{\delta}}\right)$$

In the special case of kernel selection we can apply generalization results for learning with random features to show that we can compete with the optimal RKHS from among those associated with one of the configurations [Rudi and Rosasco, 2017, Theorem 1]:

Corollary 6. *In Setting 3, suppose each configuration $c \in \mathcal{C}$ is associated with a random Fourier feature approximation of a continuous shift-invariant kernel that approximates an RKHS $\mathcal{H}_c$. Suppose ℓ is the squared loss so that $(\hat{w}, \hat{c}) \in \mathcal{W} \times \mathcal{C}$ is obtained as a solution to the following optimization problem:*

$$\operatorname*{arg\,min}_{w \in \mathcal{W}, c \in \mathcal{C}} \quad \ell_V(w, c) \qquad \textit{s.t.} \qquad w = \operatorname*{arg\,min}_{w \in \mathcal{W}} \quad \lambda \|w\|_2^2 + \sum_{(x,y) \in T} (\langle w, \phi_c(x) \rangle - y)^2$$

If the number of random features $d = \mathcal{O}(\sqrt{|T|} \log \sqrt{|T|}/\delta)$ and $\lambda = 1/\sqrt{|T|}$ then w.p. $1 - \delta$ we have

$$\ell_{\mathcal{D}}(h_{\hat{w}, \hat{c}}) - \min_{c \in \mathcal{C}} \min_{h \in \mathcal{H}_c} \ell_{\mathcal{D}}(h) \leq \mathcal{O}\left(\frac{\log^2 \frac{1}{\delta}}{\sqrt{|T|}} + \sqrt{\frac{1}{|V|} \log \frac{|C| + 1}{\delta}}\right)$$

In the case of neural architecture search we are often solving (unregularized) ERM in the inner optimization problem. In this case we can make an assumption weaker than Assumption 3, namely that the set of empirical risk minimizers contains a solution that, rather than having low excess risk, simply has low generalization error; then applying Hoeffding's inequality yields the following:

Corollary 7. *In Setting 2 let $(\hat{w}, \hat{c}) \in \mathcal{W} \times \mathcal{C}$ be obtained as a solution to the following optimization problem:*

$$\arg\min_{w \in \mathcal{W}, c \in \mathcal{C}} \quad \ell_V(w, c) \qquad \textit{s.t.} \qquad w \in \arg\min_{w' \in \mathcal{W}} \ell_T(w', c)$$

Suppose there exists $c^* \in \mathcal{C}$ satisfying $(w^*, c^*) \in \arg\min_{\mathcal{W} \times \mathcal{C}} \ell_{\mathcal{D}}(w, c)$ for some weights $w^* \in \mathcal{W}$ such that w.p. $1 - \delta$ over the drawing of training set $T \sim \mathcal{D}^{|T|}$ at least one of the minima of the optimization problem induced by c^* and T has low generalization error, i.e. $\exists\ w \in \arg\min_{w' \in \mathcal{W}} \ell_T(w', c^*)$ s.t.

$$\ell_{\mathcal{D}}(w, c^*) - \ell_T(w^*, c^*) \leq \varepsilon^*(|T|, \delta)$$

for some error function ε^*. Then we have w.p. $1 - 4\delta$ that

$$\ell_{\mathcal{D}}(\hat{w}, \hat{c}) \leq \ell_{\mathcal{D}}(w^*, c^*) + \varepsilon^*(|T|, \delta) + B\sqrt{\frac{1}{2|V|}\log\frac{1}{\delta}} + B\sqrt{\frac{1}{2|T|}\log\frac{1}{\delta}}$$
$$+ \inf_{\varepsilon > 0} 3\varepsilon + B\sqrt{\frac{2}{|V|}\left(\Lambda(H, \varepsilon, T) + \log\frac{1}{\delta}\right)}$$

7.5 Feature Map Selection Details

For selecting feature maps on CIFAR-10 we used the Ridge regression solver from scikit-learn [Pedregosa et al., 2011] to solve the inner ℓ_2-regularized ERM problem. The regularization was fixed to $\lambda = \frac{1}{2}$ since weight-sharing does not tune non-architectural hyperparameters; the same λ was used for all search algorithms. We tested whether including λ in the search space helped random search and found that it did not, or even caused random search to do worse; this is likely due to the bandwidth parameter playing a similar role.

For the search space we used the same kernel configuration settings as Table 2.3 but replacing the regularization parameter by the option to use the Laplace kernel instead of the Gaussian kernel. The data split used was the standard 40K/10K/10K for training/validation/testing. Our main results in Figure 7.2 are for 10 independent trials of each algorithm with the maximum number of features used by all competing algorithms set to 100K. For both the Weight-Sharing and Fast Weight-Sharing algorithms we started with 256 different configurations. For Figure 7.1 we varied the feature dimension as shown and considered the correlation between shared-weights performance and standalone performance fo 32 different configurations.

Chapter 8

Conclusion

In this chapter, we provide a summary of our contributions by chapter before concluding with a discussion of the future of AutoML.

8.1 Hyperparameter Optimization

Motivated by the rising computational cost of training modern machine learning models and the ever larger search spaces accompanying such models, the first part of this thesis presented efficient methods for hyperparameter optimization that exploit principled early-stopping to drastically decrease the cost associated with evaluating a hyperparameter configuration.

Chapter 2: Principled Early-Stopping with Hyperband

1. We introduced HYPERBAND (Chapter 2.3), which runs successive halving (SHA) with multiple early-stopping rates to automatically tradeoff between exploration and exploitation when the optimal early-stopping rate is unknown.
2. Our theoretical analysis of HYPERBAND (Chapter 2.5) demonstrated that we only require a small constant factor more budget than optimal to find a good configuration.
3. We benchmarked HYPERBAND against random search, Hyperopt, Spearmint, SMAC, and SMAC with early-stopping via performance prediction. Our results showed HYPERBAND to be up to an order-of-magnitude faster than methods that use full-training (Chapter 2.4).

Chapter 3: Hyperparameter Optimization in the Large-Scale Regime

1. We presented the Asynchronous Successive Halving Algorithm (ASHA) to address the synchronization bottleneck associated with SHA (Chapter 3.3). Using simulated workloads, we showed ASHA is indeed more robust to stragglers and dropped jobs than synchronous SHA.
2. We evaluated ASHA against mature state-of-the-art hyperparameter optimization methods, i.e., PBT, BOHB, and Vizier, on a comprehensive suite of tuning tasks (Chapter 3.4). Our

results showed ASHA outperforms or matches the top competing method on all benchmarks. Additionally, we showed ASHA can scale linearly with the number of workers and is well-suited to the large-scale regime.

3. In response to the systems considerations we faced when implementing ASHA, we discussed design decisions we made to improve usability, parallel training efficiency, resource utilization, and reproducibility (Chapter 3.5).

Chapter 4: Reuse in Pipeline-Aware Hyperparameter Optimization

1. Recognizing the potential speedups from reuse when tuning the hyperparameters of multi-stage machine learning pipelines, we presented a pipeline-aware approach to hyperparameter optimization that maximizes shared computation across multiple pipelines; balances training heavy search spaces; and captures the benefits from reuse with a suitable caching strategy (Chapter 4.4).
2. Our experiments demonstrated that combining early-stopping via successive halving with reuse of shared computation can offer over an order-of-magnitude speedups over naive evaluation of a set of pipelines (Chapter 4.5).

8.2 Neural Architecture Search

First generation NAS methods were prohibitively expensive since they relied on full-training to evaluate different architectures. Since then, weight-sharing has become a common component in NAS methods due to its computational efficiency. The second part of this thesis focused on developing a better understanding of weight-sharing and using our insights to inform the design of better algorithms for NAS.

Chapter 5: Random Search Baselines for NAS

1. Although NAS is a specialized hyperparameter optimization problem, it was unclear how much of a gap existed between NAS-specific methods and traditional hyperparameter optimization approaches. The dearth of ablation studies also obscured the impact of different NAS components. We introduced random search weight-sharing (RSWS) to isolate the impact of weight-sharing on performance and provide a better baseline for NAS (Chapter 5.4).
2. Our empirical studies established ASHA and RSWS as two competitive random search baselines for NAS. In fact, we showed that RSWS achieves state-of-the-art performance on designing RNN cells for language modeling on PTB (Chapter 5.5).

Chapter 6: Geometry-Aware Optimization for Gradient-Based NAS

1. We developed a framework for gradient-based weight-sharing NAS by drawing upon the theory of mirror descent (Chapter 6.3). Our framework connects the convergence rate

with the underlying optimization geometry, motivating Geometry-Aware Exponentiated Algorithms (GAEA) for NAS.

2. Under this framework, we provided finite-time convergence guarantees for many existing gradient-based weight-sharing methods (Chapter 6.4).
3. Applying GAEA to existing NAS methods, we achieved state-of-the-art performance on the DARTS search space for CIFAR-10 and ImageNet; and reach near-oracle optimality on the NASBench-201 benchmarks (Chapter 6.5).

Chapter 7: Weight-Sharing Beyond NAS

1. We applied weight-sharing to feature map selection and showed that it is competitive with successive halving and outperforms random search on a kernel classification task (Chapter 7.2). In this simplified setting, we demonstrated how solving the bi-level problem provides better generalization guarantees than that possible with empirical risk minimization.
2. We made a case for the suitability of weight-sharing for hyperparameter optimization in the federated learning setting due to its resource efficiency and potential to preserve information from ephemeral devices in the shared weights (Chapter 7.3). We introduced FedEx for this setting and defer an empirical study of the method to future work.

8.3 Future Work

While, collectively, the community has made great strides towards efficient automated machine learning, there remains important outstanding directions for future research. Externally, the deleterious environmental impact associated with large-scale machine learning and the shortage of ML talent present tough challenges but also opportunities for advances in AutoML to make an outsized impact. We present some thoughts on the current state of research and promising directions for future work below.

8.3.1 Hyperparameter Optimization

With the methods presented in Part 1 of this thesis, as well as other state-of-the-art hyperparameter approaches [Falkner et al., 2018, Jaderberg et al., 2017], we have fairly efficient algorithms for one-off hyperparameter tuning tasks. However, as more experience is collected from diverse hyperparameter tuning tasks, we should ideally be able to transfer information from prior experience to speed up the search for a good configuration on a new problem. While there has been some work on learning from prior experience, existing methods offer modest speedups over tuning each tasks independently [Feurer et al., 2015b, Perrone et al., 2018, 2019]. Developing more sophisticated methods that better capture the relationship between the objective of interest, the search space, and the underlying datasets across multiple hyperparameter tuning experiments is hence an interesting direction for future research.

A second thread of future work is motivated by the development of new learning paradigms like federated learning, meta-learning, and never-ending learning, which each have their own nuances. Hyperparameter optimization in these settings will require novel techniques that move beyond one-off model training workloads. We proposed one such approach for federated learning in Chapter 7.3 but, as we move away from one-off models towards methods that continually learn and adapt, a new paradigm for hyperparameter optimization is needed. Under such a paradigm, hyperparameters will need to be continually refined along with the underlying model for changing data distributions.

8.3.2 Neural Architecture Search

Great strides have been made to reduce the computational cost and improve the robustness of NAS methods, thereby increasing the value proposition of NAS. NAS designed CNN architectures like AmoebaNet [Real et al., 2018] and EfficientNet [Tan and Le, 2019] are becoming the new go-to models for computer vision tasks. Weight-sharing NAS approaches are also used at Google to design architectures for specific deployment environments. Future work will continue to improve the value proposition of NAS by extending the search space to jointly optimize over deployment constraints and compression techniques. We have already seen some exciting developments in this direction: Cai et al. [2020] explore the performance of training a single super network encompassing high quality architectures for multiple deployment scenarios and Wang et al. [2020] develop a method for jointly learning neural network architecture, pruning, and quantization.

A more grandiose goal is to design algorithms that can automatically discover novel fundamental building blocks akin to spatial convolution, batch normalization, and residual connections. This will require abandoning existing NAS search spaces, which incorporate expert knowledge by design, in favor of larger spaces with less inherent structure and lower level primitives. Prior work has succeeded in designing better data augmentation strategies [Cubuk et al., 2019] and activation functions [Ramachandran et al., 2017], but each of these search spaces is fairly narrow and was designed with human expertise. A key question going forward is how can we automatically discover and capture the right invariances for any dataset?

Bibliography

J. Abernethy, E. Hazan, and A. Rakhlin. Competing in the dark: An efficient algorithm for bandit linear optimization. In *Proceedings of the International Conference on Computational Learning Theory*, 2008. 6.1.1

A. Agarwal, J. Duchi, P. L. Bartlett, and C. Levrard. Oracle inequalities for computationally budgeted model selection. In *Conference on Learning Theory*, 2011. 2.1, 2.2.1

A. Agarwal, S. Kakade, N. Karampatziakis, L. Song, and G. Valiant. Least squares revisited: Scalable approaches for multi-class prediction. In *International Conference on Machine Learning*, 2014. 2.4.2.2

T. Ajanthan, K. Gupta, P. H. S. Torr, R. Hartley, and P. K. Dokania. Mirror descent view for neural network quantization. *arXiv*, 2019. 6.1.1

T. Akiba, S. Sano, T. Yanase, T. Ohta, and M. Koyama. Optuna: A next-generation hyperparameter optimization framework. In *International Conference on Knowledge Discovery and Data Mining*. 1.4, 3.2

Y. Akimoto, S. Shirakawa, N. Noshinari, K. Uchida, S. Saito, and K. Nishida. Adaptive stochastic natural gradient method for one-shot neural architecture search. In *International Conference on Machine Learning*, 2019. 6.1, 6.1.1, 6.2.2, 6.1, 7.1.1, 7.3.1

S. Arora, E. Hazan, and S. Kale. The multiplicative weights update method: a meta-algorithm and applications. *Theory of Computing*, 8:121–164, 2012. 6.1.1

N. Bansal, N. Buchbinder, and J. S. Naor. Randomized competitive algorithms for generalized caching. In *Symposium on Theory of Computing*. ACM, 2008. 4.2

P. Bartlett, D. J. Foster, and M. Telgarsky. Spectrally-normalized margin bounds for neural networks. In *Neural Information Processing Systems*, 2017. 7.2.2.2

A. Beck. *First-Order Methods in Optimization*. MOS-SIAM, 2017. 6.1.1

A. Beck and M. Teboulle. Mirror descent and nonlinear projected subgradient methods for convex optimization. *Operations Research Letters*, 31:167–175, 2003. 6.1, 6.1.1, 6.3.1, 6.3.1

G. Bender, P.-J. Kindermans, B. Zoph, V. Vasudevan, and Q. Le. Understanding and simplifying one-shot architecture search. In *International Conference on Machine Learning*, 2018. 1.3.3, 5.1, 5.2.2, 5.1, 5.2.3, 4, 5.4.2

Y. Bengio. Gradient-based optimization of hyperparameters. *Neural Computation*, 12:1889–1900, 2000. 1.2.2, 5.1

J. Bergstra and Y. Bengio. Random search for hyper-parameter optimization. *Journal of Machine*

Learning Research, 13(Feb):281–305, 2012. 1.2.1, 1.2.2, 2.1, 4.2, 7

J. Bergstra, R. Bardenet, Y. Bengio, and B. Kegl. Algorithms for hyper-parameter optimization. In *Neural Information Processing Systems*, 2011. 1.2.2, 2.1, 2.2.1, 4.1, 5.1

L. M. Bregman. The relaxation method of finding the common point of convex sets and its application to the solution of problems in convex programming. *USSR Computational Mathematics and Mathematical Physics*, 7:200–217, 1967. 6.3.1

D. Britz, A. Goldie, M.-T. Luong, and Q. Le. Massive exploration of neural machine translation architectures. In *Conference on Empirical Methods in Natural Language Processing*, Copenhagen, Denmark, 2017. Association for Computational Linguistics. 1.1

A. Brock, T. Lim, J. Ritchie, and N. Weston. SMASH: One-shot model architecture search through hypernetworks. In *International Conference on Learning Representations*, 2018. 1.3.3, 5.1, 5.1, 5.2.3

S. Bubeck, R. Munos, and G. Stoltz. Pure exploration in multi-armed bandits problems. In *International Conference on Algorithmic Learning Theory*, 2009. 2.2.2, 2.5.3.3

S. Bubeck, R. Munos, G. Stoltz, and C. Szepesvari. X-armed bandits. *Journal of Machine Learning Research*, 12:1655–1695, 2011. 2.2.2

S. Bubeck, M. B. Cohen, J. R. Lee, Y. T. Lee, , and A. Mądry. K-server via multiscale entropic regularization. In *Symposium on Theory of Computing*, 2018. 6.1.1

H. Cai, J. Yang, W. Zhang, S. Han, and Y. Yu. Path-level network transformation for efficient architecture search. In *International Conference on Machine Learning*, 2018. 1.3.3, 5.1, 5.2.1, 5.1, 5.2.3, 5.2.3

H. Cai, L. Zhu, and S. Han. ProxylessNAS: Direct neural architecture search on target task and hardware. In *International Conference on Learning Representations*, 2019. 1.3.2, 1.3.3, 1.4, 5.1, 5.2.2, 5.1, 5.4.2, 5.5, 3, 6.1, 6.2.2, 6.1, 6.5.1.1, 6.3

H. Cai, C. Gan, T. Wang, Z. Zhang, and S. Han. Once-for-all: Train one network and specialize it for efficient deployment. In *International Conference on Learning Representations*, 2020. 8.3.2

S. Caldas, P. Wu, T. Li, J. Konečný, H. B. McMahan, V. Smith, and A. Talwalkar. LEAF: A benchmark for federated settings. *arXiv*, 2018. 7.3.2

S. Cao, X. Wang, and K. M. Kitani. Learnable embedding space for efficient neural architecture compression. In *International Conference on Learning Representations*, 2019. 5.1

F. M. Carlucci, P. M. Esperança, M. Singh, V. Gabillon, A. Yang, H. Xu, Z. Chen, and J. Wang. MANAS: Multi-Agent Neural Architecture Search. *arXiv*, 2019. 1.4, 5.6, 6.1, 6.1.1

A. Carpentier and M. Valko. Simple regret for infinitely many armed bandits. In *International Conference on Machine Learning*, 2015. 2.2.2, 2.5.3.2, 2.5.3.3

C. Chen, A. Seff, A. Kornhauser, and J. Xiao. Deepdriving: Learning affordance for direct perception in autonomous driving. In *International Conference on Computer Vision*, 2015. 1.1

F. Chen, Z. Dong, Z. Li, and X. He. Federated meta-learning with fast convergence and efficient communication. *arXiv*, 2019. 7.3.1

X. Chen, L. Xie, J. Wu, and Q. Tian. Progressive Differentiable Architecture Search: Bridging

the Depth Gap between Search and Evaluation. *arXiv*, 2019. 6, 6.2.2, 6.5, 6.1, 6.3, 6.7.1

M. Cho, M. Soltani, and C. Hegde. One-shot neural architecture search via compressive sensing. *arXiv*, 2019. 1.4, 1.4, 5.6

A. Choromanska, M. Henaff, M. M. G. B. Arous, and Y. LeCun. The loss surface of multilayer networks. In *International Conference on Artificial Intelligence and Statistics*, 2015. 7.2.2.2

E. Contal, V. Perchet, and N. Vayatis. Gaussian process optimization with mutual information. In *International Conference on Machine Learning*, 2014. 2.2.1

E. D. Cubuk, B. Zoph, D. Mane, V. Vasudevan, and Q. V. Le. Autoaugment: Learning augmentation strategies from data. In *Conference on Computer Vision and Pattern Recognition*, 2019. 6.5.1.2, 6.3, 8.3.2

C. D. Dang and G. Lan. Stochastic block mirror descent methods for nonsmooth and stochastic optimization. *SIAM Journal on Optimization*, 25:856–881, 2015. 6.1.1, 6.4, 6.4

C. Daskalakis, A. Ilyas, V. Syrgkanis, and H. Zeng. Training GANs with optimism. In *International Conference on Learning Representations*, 2018. 6.1.1

D. Davis and D. Drusvyatskiy. Stochastic model-based minimization of weakly convex functions. *SIAM Journal on Optimization*, 29(1):207–239, 2019. 6.4

T. Devries and G. W. Taylor. Improved regularization of convolutional neural networks with cutout. *arXiv*, 2017. 5.5, 6.1

T. Domhan, J. T. Springenberg, and F. Hutter. Speeding up automatic hyperparameter optimization of deep neural networks by extrapolation of learning curves. In *International Joint Conference on Artificial Intelligence*, 2015. 1.2.3, 2.1, 2.4, 2.4.1, 2.3, 2.4.1, 2.6.1, 3.6.1

X. Dong and Y. Yang. Searching for a robust neural architecture in four gpu hours. In *Conference on Computer Vision and Pattern Recognition*, 2019. 6.1, 6.2.1, 6.2.2, 6.3.4, 6.5, 6.5.3

X. Dong and Y. Yang. NAS-Bench-201: Extending the scope of reproducible neural architecture search. In *International Conference on Learning Representations*, 2020. 1.4, 1.4, 5.6, 6.1, 3c, 6.2.1, 6.5, 6.5.3, 6.7.3

A. Dvoretzky, J. Kiefer, and J. Wolfowitz. Asymptotic minimax character of the sample distribution function and of the classical multinomial estimator. *The Annals of Mathematical Statistics*, 27:642–669, 1956. 3.3.4

K. Eggensperger, M. Feurer, F. Hutter, J. Bergstra, J. Snoek, H. Hoos, and K. Leyton-Brown. Towards an empirical foundation for assessing Bayesian optimization of hyperparameters. In *NIPS Bayesian Optimization Workshop*, 2013. 2.1, 2.2.1, 2.4

T. Elsken, J. H. Metzen, and F. Hutter. Multi-objective Architecture Search for CNNs. *arXiv*, 2018a. 1.3.3, 5.1, 5.2.2

T. Elsken, J. H. Metzen, and F. Hutter. Neural Architecture Search: A Survey. *arXiv*, 2018b. 1.3.1

E. Even-Dar, S. Mannor, and Y. Mansour. Action elimination and stopping conditions for the multi-armed bandit and reinforcement learning problems. *Journal of Machine Learning Research*, 7: 1079–1105, 2006. 2.2.2

S. Falkner, A. Klein, and F. Hutter. BOHB: Robust and efficient hyperparameter optimization at

scale. In *International Conference on Machine Learning*, 2018. 1.2.3, 2.1, 11, 2.4, 3.1, 3.2, 12, 3.3.3, 3.4.1, 8.3.1

M. Feurer. Personal communication, 2015. 2.4.2.1

M. Feurer, J. Springenberg, and F. Hutter. Using meta-learning to initialize Bayesian optimization of hyperparameters. In *ECAI Workshop on Meta-Learning and Algorithm Selection*, 2014. 2.2.1

M. Feurer, A. Klein, K. Eggensperger, J. Springenberg, M. Blum, and F. Hutter. Efficient and robust automated machine learning. In *Neural Information Processing Systems*, 2015a. 1.2.1, 2.2.1, 2.4.2, 2.4.2.1, 2.6.2, 3.2, 4.1, 3

M. Feurer, J. T. Springenberg, and F. Hutter. Initializing Bayesian hyperparameter optimization via meta-learning. In *Association for the Advancement of Artificial Intelligence*, 2015b. 8.3.1

A. Fiat, R. M. Karp, M. Luby, L. A. McGeoch, D. D. Sleator, and N. E. Young. Competitive paging algorithms. *J. Algorithms*, 12(4):685–699, Dec. 1991. 4.2

L. Franceschi, P. Frasconi, S. Salzo, R. Grazzi, and M. Pontil. Bilevel programming for hyperparameter optimization and meta-learning. In *International Conference on Machine Learning*, 2018. 6.2.1, 2, 7.1

P. Gijsbers, J. Vanschoren, and R. S. Olson. Layered tpot: Speeding up tree-based pipeline optimization. *arXiv*, 2018. 4.2

D. Ginsbourger, R. Le Riche, and L. Carraro. Kriging is well-suited to parallelize optimization. In *Computational Intelligence in Expensive Optimization Problems*, pages 131–162. Springer, 2010. 3.2

D. Golovin, B. Sonik, S. Moitra, G. Kochanski, J. Karro, and D.Sculley. Google Vizier: A service for black-box optimization. In *International Conference on Knowledge Discovery and Data Mining*, 2017. 1.2.3, 1.4, 3.1, 3.2, 3.4.5

J. González, D. Zhenwen, P. Hennig, and N. Lawrence. Batch Bayesian optimization via local penalization. In *International Conference on Artificial Intelligence and Statistics*, 2016. 3.2

Google. Google repo for amoebanet. https://github.com/tensorflow/tpu/tree/master/models/official/amoeba_net, 2018a. 5.2.3

Google. Google automl. https://cloud.google.com/automl/, 2018b. 5.1

Google. Google repo for nasnet. https://github.com/tensorflow/models/tree/master/research/slim/nets/nasnet, 2018c. 5.2.3

P. Goyal, P. Dollár, R. B. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He. Accurate, large minibatch SGD: training imagenet in 1 hour. *arXiv*, 2017. 3.5.2

H. Greenspan, B. Van Ginneken, and R. M. Summers. Guest editorial deep learning in medical imaging: Overview and future promise of an exciting new technique. *IEEE Transactions on Medical Imaging*, 35(5):1153–1159, 2016. 1.1

S. Grünewälder, J. Audibert, M. Opper, and J. Shawe–Taylor. Regret bounds for Gaussian process bandit problems. In *International Conference on Artificial Intelligence and Statistics*, 2010. 2.2.1

P. K. Gunda, L. Ravindranath, C. A. Thekkath, Y. Yu, and L. Zhuang. Nectar: Automatic management of data and computation in datacenters. In *Symposium on Operating Systems Design and Implementation*, volume 10, pages 1–8, 2010. 4.1, 4.2

Z. Guo, X. Zhang, H. Mu, W. Heng, Z. Liu, Y. Wei, and J. Sun. Single path one-shot neural architecture search with uniform sampling. *arXiv*, 2019. 7.1.1

A. György and L. Kocsis. Efficient multi-start strategies for local search algorithms. *Journal of Artificial Intelligence Research*, 41, 2011. 2.1, 2.2.1

S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Computation*, 9:1735–1780, 1997. 7.3.2

E. Hoffer, I. Hubara, and D. Soudry. Train longer, generalize better: closing the generalization gap in large batch training of neural networks. In *Neural Information Processing Systems*, 2017. 5.2.2

A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv*, 2017. 6.3

J. Hu, L. Shen, and G. Sun. Squeeze-and-excitation networks. In *Conference on Computer Vision and Pattern Recognition*, 2018. 6.3, 6.5.1.2

P.-S. Huang, H. Avron, T. N. Sainath, V. Sindhwani, and B. Ramabhadran. Kernel methods match deep neural networks on TIMIT. In *International Conference on Acoustics, Speech and Signal Processing*. IEEE, 2014. 4.5

A. Hundt, V. Jain, and G. Hager. sharpdarts: Faster and more accurate differentiable architecture search. *arXiv*, 2019. 6.2.2

F. Hutter, H. Hoos, and K. Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *International Conference on Learning and Intelligent Optimization*, 2011. 1.2.2, 2.1, 2.2.1, 4.1, 5.1

M. Jaderberg, V. Dalibard, S. Osindero, W. Czarnecki, J. Donahue, A. Razavi, O. Vinyals, T. Green, I. Dunning, K. Simonyan, et al. Population based training of neural networks. *arXiv*, 2017. 2.1, 2.4, 3.1, 3.2, 3.6.2, 5.1, 8.3.1

K. Jamieson and R. Nowak. Best-arm identification algorithms for multi-armed bandits in the fixed confidence setting. In *Conference on Information Sciences and Systems*, pages 1–6. IEEE, 2014. 2.2.1

K. Jamieson and A. Talwalkar. Non-stochastic best arm identification and hyperparameter optimization. In *International Conference on Artificial Intelligence and Statistics*, 2015. 2.1, 2.2.1, 2.2.2, 2.3.1, 2, 2.8, 2.5.3

K. Jamieson, M. Malloy, R. Nowak, and S. Bubeck. lil'UCB: An optimal exploration algorithm for multi-armed bandits. In *Conference On Learning Theory*, pages 423–439, 2014. 2.5.3.3

K. G. Jamieson, D. Haas, and B. Recht. The power of adaptivity in identifying statistical alternatives. In *Neural Information Processing Systems*, pages 775–783, 2016. 2.5.3.2

Y. Jiang, J. Konečný, K. Rush, and S. Kannan. Improving federated learning personalization via

model agnostic meta learning. *arXiv*, 2019. 7.3.1

H. Jin, Q. Song, and X. Hu. Auto-Keras: Efficient Neural Architecture Search with Network Morphism. *arXiv*, 2018. 1.3.2, 1.3.3, 5.1, 5.2.1, 5.2.2

P. Kairouz, H. B. McMahan, B. Avent, A. Bellet, M. Bennis, A. N. Bhagoji, K. Bonawitz, Z. Charles, G. Cormode, R. Cummings, R. G. L. D'Oliveira, S. E. Rouayheb, D. Evans, J. Gardner, Z. Garrett, A. Gascón, B. Ghazi, P. B. Gibbons, M. Gruteser, Z. Harchaoui, C. He, L. He, Z. Huo, B. Hutchinson, J. Hsu, M. Jaggi, T. Javidi, G. Joshi, M. Khodak, J. Konečný, A. Korolova, F. Koushanfar, S. Koyejo, T. Lepoint, Y. Liu, P. Mittal, M. Mohri, R. Nock, A. Özgür, R. Pagh, M. Raykova, H. Qi, D. Ramage, R. Raskar, D. Song, W. Song, S. U. Stich, Z. Sun, A. T. Suresh, F. Tramèr, P. Vepakomma, J. Wang, L. Xiong, Z. Xu, Q. Yang, F. X. Yu, H. Yu, and S. Zhao. Advances and open problems in federated learning. *arXiv*, 2019. 7.3, 7.3.1

K. Kandasamy, J. Schneider, and B. Póczos. High dimensional Bayesian optimization and bandits via additive models. In *International Conference on Machine Learning*, 2015. 1, 2.2.1

K. Kandasamy, G. Dasarathy, J. B. Oliva, J. Schneider, and B. Póczos. Gaussian process bandit optimisation with multi-fidelity evaluations. In *Neural Information Processing Systems*, 2016. 1.2.2, 1.2.3

K. Kandasamy, G. Dasarathy, J. Schneider, and B. Póczos. Multi-fidelity bayesian optimisation with continuous approximations. In *International Conference on Machine Learning*, 2017. 3

K. Kandasamy, W. Neiswanger, J. Schneider, B. Poczos, and E. Xing. Neural Architecture Search with Bayesian Optimization and Optimal Transport. *arXiv*, 2018. 1.3.2, 5.1, 5.2.2, 5.1, 5.2.3

Z. Karnin, T. Koren, and O. Somekh. Almost optimal exploration in multi-armed bandits. In *International Conference on Machine Learning*, 2013. 1.4, 2.2.1, 2.8, 2.5.3, 2.5.3.3

E. Kaufmann, O. Cappé, and A. Garivier. On the complexity of best arm identification in multi-armed bandit models. *The Journal of Machine Learning Research*, 2015. 2.5.3.3

M. Kearns, Y. Mansour, A. Y. Ng, and D. Ron. An experimental and theoretical comparison of model selection methods. *Machine Learning*, 27:7–50, 1997. 7.2.2, 2

M. Khodak, M.-F. Balcan, and A. Talwalkar. Adaptive gradient-based meta-learning methods. In *Neural Information Processing Systems*, 2019. 7.3.1

M. Khodak, L. Li, T. Li, N. Roberts, V. Smith, M. Balcan, and A. Talwalkar. Weight-sharing beyond neural architecture search. *Preprint: In Submission*, 2020. 1.4, 1.4

J. Kim, M. Kim, H. Park, E. Kusdavletov, D. Lee, A. Kim, J. Kim, J. Ha, and N. Sung. CHOPT : Automated hyperparameter optimization framework for cloud-based machine learning platforms. *arXiv*, 2018. 3.2

D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. In *International Conference on Learning Representations*, 2015. 6.1, 10, 6.3.3, 6.5.1.1

A. Klein, S. Falkner, S. Bartels, P. Hennig, and F. Hutter. Fast Bayesian optimization of machine learning hyperparameters on large datasets. In *International Conference on Artificial Intelligence and Statistics*, 2017a. 1.2.2, 1.2.3, 2.1, 2.4, 3.1, 3.2, 3.4.1, 3.6.1, 1, 3.6.1, 3

A. Klein, S. Falkner, J. T. Springenberg, and F. Hutter. Learning curve prediction with Bayesian

neural networks. In *International Conference On Learning Representation*, 2017b. 11, 3.2

B. Kleinberg, Y. Li, and Y. Yuan. An alternative view: When does SGD escape local minima? In *International Conference on Machine Learning*, 2018. 6.3.3

J. Kober, J. A. Bagnell, and J. Peters. Reinforcement learning in robotics: A survey. *The International Journal of Robotics Research*, 32(11):1238–1274, 2013. 1.1

L. Kotthoff, C. Thornton, H. H. Hoos, F. Hutter, and K. Leyton-Brown. Auto-weka 2.0: Automatic model selection and hyperparameter optimization in weka. *The Journal of Machine Learning Research*, 18(1):826–830, 2017. 3.2

A. Krizhevsky. Learning multiple layers of features from tiny images. In *Technical report, Department of Computer Science, Univsersity of Toronto*, 2009. 2.4.1, 2, 5.5

A. Krizhevsky. One weird trick for parallelizing convolutional neural networks. *arXiv*, 2014. 3.5.2

T. Krueger, D. Panknin, and M. Braun. Fast cross-validation via sequential testing. *Journal of Machine Learning Research*, 16:1103–1155, 2015. 2.2.1, 4.1

G. Kunapuli and J. Shavlik. Mirror descent for metric learning: A unified approach. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, 2012. 6.1.1

J. Lafferty, H. Liu, and L. Wasserman. Statistical machine learning. 2010. 7.4.2

H. Larochelle, D. Erhan, A. Courville, J. Bergstra, and Y. Bengio. An empirical evaluation of deep architectures on problems with many factors of variation. In *International Conference on Machine Learning*, 2007. 2.4.1

K. A. Laube and A. Zell. Prune and replace nas. *arXiv preprint https://arxiv.org/abs/1906.07528*, 2019. 6.1, 6.2.2

A. Li, O. Spyra, S. Perel, V. Dalibard, M. Jaderberg, C. Gu, D. Budden, T. Harley, and P. Gupta. A generalized framework for population based training. *arXiv*, 2019a. 3.2

G. Li, X. Zhang, Z. Wang, Z. Li, and T. Zhang. StacNAS: Towards stable and consistent differentiable neural architecture search. *arXiv*, 2019b. 6.1, 1, 6.2.1, 7.1.2, 7.3.1

L. Li and A. Talwalkar. Random search and reproducibility for neural architecture search. In *Uncertainty in Artificial Intelligence*, 2019. 1.4, 1.4

L. Li, K. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar. Hyperband: Bandit-based configuration evaluation for hyperparameter optimization. *International Conference on Learning Representation*, 17, 2017. 1.4, 1.4

L. Li, K. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. *Journal of Machine Learning Research*, 18(185):1–52, 2018a. 1.4, 1.4

L. Li, E. Sparks, K. Jamieson, and A. Talwalkar. Exploiting reuse in pipeline-aware hyperparameter tuning. In *Workshop on Systems for ML at NeurIPS*, 2018b. 1.4, 1.4

L. Li, K. G. Jamieson, A. Rostamizadeh, E. Gonina, M. H. Jonathan Ben-Tzur, B. Recht, and A. Talwalkar. A system for massively parallel hyperparameter tuning. In *Conference on*

Machine Learning and Systems, 2020a. 1.4, 1.4

L. Li, M. Khodak, M. Balcan, and A. Talwalkar. Geometry-aware gradient algorithms for neural architecture search. *arXiv*, 2020b. 1.4, 1.4

T. Li, A. K. Sahu, A. Talwalkar, and V. Smith. Federated learning: Challenges, methods, and future directions. *arXiv*, 2019c. 7.3.1

H. Liang, S. Zhang, J. Sun, X. He, W. Huang, K. Zhuang, and Z. Li. DARTS+: Improved Differentiable Architecture Search with Early Stopping. *arXiv*, 2019. 6.1, 6.2.2, 6.5, 6.7.1

R. Liaw, E. Liang, R. Nishihara, P. Moritz, J. E. Gonzalez, and I. Stoica. Tune: A research platform for distributed model selection and training. *arXiv*, 2018. 1.4, 3.2

C. Liu, B. Zoph, M. Neumann, J. Shlens, W. Hua, L.-J. Li, L. Fei-Fei, A. Yuille, J. Huang, and K. Murphy. Progressive Neural Architecture Search. *arXiv*, 2018a. 1.3.3, 5.1, 5.2.2, 6.1

H. Liu, K. Simonyan, O. Vinyals, C. Fernando, and K. Kavukcuoglu. Hierarchical representations for efficient architecture search. In *International Conference on Learning Representations*, 2018b. 5.1, 5.1

H. Liu, K. Simonyan, and Y. Yang. DARTS: differentiable architecture search. *arXiv*, 2018c. 5.7.1

H. Liu, K. Simonyan, and Y. Yang. DARTS: Differentiable architecture search. In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=S1eYHoC5FX. 1.3.1, 1.3.2, 1.3.3, 1.4, 3.4.3, 3.6.4, 5.1, 1, 1, 5.2.1, 5.2.2, 5.1, 5.2.3, 5.2.3, 5.3.1, 5.3.1.1, 5.3.1.1, 5.3.1.2, 5.3.1.2, 5.4, 5.5, 5.5.1.1, 5.5.1.1, 5.2, 5.3, 5.5.1.3, 5.5.2, 5.5.2.1, 5.5, 1, 5.7.1, 5.7.2, 6, 6.1, 1, 3a, 6.1.1, 6.2.1, 6.2.1, 6.2.2, 10, 6.3.3, 6.3.4, 6.5, 6.5.1, 6.1, 6.5.1.1, 6.3, 6.7.1, 3, 7.2.2

R. Luo, F. Tian, T. Qin, E. Chen, and T.-Y. Liu. Neural Architecture Optimization. *arXiv*, 2018. 5.1, 5.2.2, 5.1, 5.2.3

D. V. N. Luong, P. Parpas, D. Rueckert, and B. Rustem. Solving MRF minimization by mirror descent. In *Advances in Visual Computing*, 2012. 6.1.1

N. Ma, X. Zhang, H.-T. Zheng, and J. Sun. Shufflenet v2: Practical guidelines for efficient cnn architecture design. In *European Conference on Computer Vision*, pages 116–131, 2018. 6.3

D. Maclaurin, D. Duvenaud, and R. P. Adams. Gradient-based hyperparameter optimization through reversible learning. *arXiv*, 2015. 1.2.2, 5.1

S. Mahadevan and B. Liu. Sparse q-learning with mirror descent. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence*, 2012. 6.1.1

M. Marcus, M. Marcinkiewicz, and B. Santorini. Building a large annotated corpus of english: The penn treebank. *Computational Linguistics*, 19(2):313–330, 1993. 1.4, 3.4.3, 3.6.6, 5.5

O. Maron and A. Moore. The racing algorithm: Model selection for lazy learners. *Artificial Intelligence Review*, 11:193–225, 1997. 2.2.1

J. McAuley, C. Targett, Q. Shi, and A. Van Den Hengel. Image-based recommendations on styles and substitutes. In *Conference on Research and Development in Information Retrieval*. ACM, 2015. 4.5

L. A. McGeoch and D. D. Sleator. A strongly competitive randomized paging algorithm. *Algorithmica*, 6(1):816–825, Jun 1991. 4.1

H. B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. Aguera y Arcas. Communication-efficient learning of deep networks from decentralized data. In *International Conference on Artifical Intelligence and Statistics*, 2017. 7, 7.3, 7.3.1, 7.3.1, 7.3.2

X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen, D. Xin, R. Xin, M. J. Franklin, R. Zadeh, M. Zaharia, and A. Talwalkar. Mllib: Machine learning in apache spark. *Journal of Machine Learning Research*, 2016. 4.1

S. Merity, N. Keskar, and R. Socher. Regularizing and optimizing LSTM language models. In *International Conference on Learning Representations*, 2018. 3.6, 3.4.4, 5.2

V. Mnih and J.-Y. Audibert. Empirical Bernstein stopping. In *International Conference on Machine Learning*, 2008. 2.2.1

M. Mohri, A. Rostamizadeh, and A. Talwalkar. *Foundations of Machine Learning*. MIT Press, 2012. 7.2.2

R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995. 4.2

R. Müller, S. Kornblith, and G. E. Hinton. When does label smoothing help? In *Neural Information Processing Systems*, 2019. 6.5.1.2, 6.3

D. Murray et al. Announcing tensorflow 0.8 now with distributed computing support!, 2016. URL https://research.googleblog.com/2016/04/announcing-tensorflow-08-now-with.html. 3.5.2

V. Nagarajan and J. Z. Kolter. Generalization in deep networks: The role of distance from initialization. *arXiv*, 2017. 7.2.2.2

V. Nagarajan and J. Z. Kolter. Uniform convergence may be unable to explain generalization in deep learning. In *Neural Information Processing Systems*, 2019. 7.2.2.2

N. Nayman, A. Noy, T. Ridnik, I. Friedman, R. Jin, and L. Zelnik. Xnas: Neural architecture search with expert advice. In *Neural Information Processing Systems*, 2019. 6, 6.1, 6.1.1, 6.2.2, 6.5, 6.5.5, 6.7.1

R. Negrinho and G. Gordon. DeepArchitect: Automatically Designing and Training Deep Architectures. *arXiv*, 2017. 5.1

A. Nemirovski and D. Yudin. *Problem Complexity and Method Efficiency in Optimization*. Wiley, 1983. 6.1, 6.1.1, 6.3.1

Y. Netzer, T. Wang, A. Coates, A. Bissacco, B. Wu, and A. Y. Ng. Reading digits in natural images with unsupervised feature learning. In *NIPS Workshop on Deep Learning and Unsupervised Feature Learning*, 2011. 2.4.1, 3, 3.6.3

A. Noy, N. Nayman, T. Ridnik, N. Zamir, S. Doveh, I. Friedman, R. Giryes, and L. Zelnik-Manor. ASAP: Architecture Search, Anneal and Prune. *arXiv*, 2019. 6.1, 6.1.1, 6.2.2, 6.5, 6.7.1

R. S. Olson and J. H. Moore. Tpot: A tree-based pipeline optimization tool for automating machine learning. In *Workshop on Automatic Machine Learning*, 2016. 1.2.1, 1.2.2, 5.1

F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Pretten-

hofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011. 4.1, 7.5

V. Perrone, R. Jenatton, M. W. Seeger, and C. Archambeau. Scalable hyperparameter transfer learning. In *Advances in Neural Information Processing Systems*, pages 6845–6855, 2018. 8.3.1

V. Perrone, H. Shen, M. W. Seeger, C. Archambeau, and R. Jenatton. Learning search spaces for bayesian optimization: Another view of hyperparameter transfer learning. In *Advances in Neural Information Processing Systems*, pages 12751–12761, 2019. 8.3.1

H. Pham, M. Guan, B. Zoph, Q. Le, and J. Dean. Efficient neural architecture search via parameters sharing. In *International Conference on Machine Learning*, 2018. 1.3.1, 1.3.3, 1.4, 5.1, 1, 5.2.1, 5.2.2, 5.1, 5.2.3, 5.5, 5.5.1.1, 5.2, 5.5.2.1, 5.5, 3, 6.1, 1, 6.1.1, 6.2.1, 6.2.2, 10, 6.3.3, 6.1, 7.1.1

H. Qi, E. R. Sparks, and A. Talwalkar. Paleo: A performance model for deep neural networks. In *International Conference on Learning Representation*, 2017. 3.2, 3.5.2

A. Rahimi and B. Recht. Random features for large-scale kernel machines. In *Neural Information Processing Systems*, 2007. 2.3.3, 2.4.3, 7.2.1

A. Rakhlin and K. Sridharan. Optimization, learning, and games with predictable sequences. In *Neural Information Processing Systems*, 2013. 6.1.1

P. Ramachandran, B. Zoph, and Q. V. Le. Searching for activation functions. *arXiv*, 2017. 6.5.1.2, 6.3, 8.3.2

E. Real, S. Moore, A. Selle, S. Saxena, Y. Leon Suematsu, Q. Le, and A. Kurakin. Large-scale evolution of image classifiers. In *International Conference on Machine Learning*, 2017. 1.3.2, 1.3.3, 1.4, 5.2.2

E. Real, A. Aggarwal, Y. Huang, and Q. V. Le. Regularized Evolution for Image Classifier Architecture Search. *arXiv*, 2018. 1.1, 1.3.2, 1.3.3, 1.4, 5.1, 5.2.1, 5.2.2, 5.2.3, 5.5, 6.1, 6.3, 8.3.2

R. Rifkin and A. Klautau. In defense of one-vs-all classification. *Journal of Machine Learning Research*, 5:101–141, 2004. 2.4.2.2

R. Y. Rubinstein and A. Shapiro. *Discrete Event Systems: Sensitivity Analysis and Stochastic Optimization by the Score Function Method*. John Wiley & Sons, Inc., 1993. 7.3.1

A. Rudi and L. Rosasco. Generalization properties of learning with random features. In *Neural Information Processing Systems*, 2017. 7.2.2.1, 7.4.3

O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision*, 115(3):211–252, 2015. 6.5.1.2

A. Sabharwal, H. Samulowitz, and G. Tesauro. Selecting near-optimal learners via incremental data allocation. In *Association for the Advancement of Artificial Intelligence*, pages 2007–2015, 2016. 4.1

J. Sánchez, F. Perronnin, T. Mensink, and J. Verbeek. Image classification with the fisher vector:

Theory and practice. *International Journal of Computer Vision*, 105(3):222–245, 2013. 4.1

P. Sermanet, S. Chintala, and Y. LeCun. Convolutional neural networks applied to house numbers digit classification. In *International Association for Pattern Recognition*, 2012. 2.6.1, 3.6.3

S. Shalev-Shwartz. Online learning and online convex optimization. *Foundations and Trends in Machine Learning*, 4(2):107—-194, 2011. 6.3.1

D. D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Commun. ACM*, 28(2):202–208, 1985. 4.1, 4.2

J. Snoek, H. Larochelle, and R. P. Adams. Practical bayesian optimization of machine learning algorithms. In *Neural Information Processing Systems*, 2012. 1.2.1, 1.2.2, 2.1, 2.2.1, 2.4.1, 2.4.1, 2.6.1, 4.1, 5.1

J. Snoek, O. Rippel, K. Swersky, R. Kiros, N. Satish, N. Sundaram, M. Patwary, Prabhat, and R. Adams. Scalable Bayesian optimization using deep neural networks. In *International Conference on Machine Learning*, 2015. 2.1, 2.2.1

E. Sparks. *End-to-End Large Scale Machine Learning with KeystoneML*. PhD thesis, EECS Department, University of California, Berkeley, Dec 2016. URL http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-200.html. 4

E. Sparks, A. Talwalkar, D. Haas, M. J. Franklin, M. I. Jordan, and T. Kraska. Automating model search for large scale machine learning,. In *ACM Symposium on Cloud Computing*, 2015. 2.1, 2.2.1

E. R. Sparks, S. Venkataraman, T. Kaftan, M. J. Franklin, and B. Recht. Keystoneml: Optimizing pipelines for large-scale advanced analytics. In *International Conference on Data Engineering*, 2017. 4.1

J. Springenberg, A. Klein, S. Falkner, and F. Hutter. Bayesian optimization with robust Bayesian neural networks. In *Neural Information Processing Systems*, 2016. 2.2.1

K. Sridharan, N. Srebro, and S. Shalev-Schwartz. Fast rates for regularized objectives. In *Neural Information Processing Systems*, 2008. 7.2.2.1, 7.4.3

N. Srinivas, A. Krause, M. Seeger, and S. M. Kakade. Gaussian process optimization in the bandit setting: No regret and experimental design. In *International Conference on Machine Learning*, 2010. 2.2.1, 2.2.2

K. Swersky, J. Snoek, and R. Adams. Multi-task Bayesian optimization. In *Neural Information Processing Systems*, 2013. 1.2.2

K. Swersky, J. Snoek, and R. P. Adams. Freeze-thaw Bayesian optimization. *arXiv*, 2014. 2.1, 3.6.1

C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015. 3.5.2

M. Tan and Q. Le. Efficientnet: Rethinking model scaling for convolutional neural networks. In *International Conference on Machine Learning*, pages 6105–6114, 2019. 8.3.2

C. Thornton, F. Hutter, H. Hoos, and K. Leyton-Brown. Auto-weka: Combined selection and

hyperparameter optimization of classification algorithms. In *International Conference on Knowledge Discovery and Data Mining*, 2013. 2.1, 2.2.1

A. van der Vaart and H. van Zanten. Information rates of nonparametric Gaussian process methods. *Journal of Machine Learning Research*, 12:2095–2119, 2011. 2.2.1

V. Vapnik. *Estimation of Dependences Based on Empirical Data*. Springer-Verlag, 1982. 7.2.2

Q. H. Vuong. Likelihood ratio tests for model selection and non-nested hypotheses. *Econometrica: Journal of the Econometric Society*, pages 307–333, 1989. 7.2.2

T. Wang, K. Wang, H. Cai, J. Lin, Z. Lu, H. Wang, Y. Lin, and S. Han. APQ: Joint search for network architecture, pruning, and quantization policy. In *Conference on Computer Vision and Pattern Recognition*, 2020. 8.3.2

Z. Wang, M. Zoghi, F. Hutter, D. Matheson, and N. de Freitas. Bayesian optimization in high dimensions via random embeddings. In *International Joint Conference on Artificial Intelligence*, 2013. 2.2.1

Z. Wang, B. Zhou, and S. Jegelka. Optimization as estimation with Gaussian processes in bandit settings. In *International Conference on Artificial Intelligence and Statistics*, 2016. 2.2.1

T. Wei, C. Wang, Y. Rui, and C. W. Chen. Network morphism. In *International Conference on Machine Learning*, 2016. 5.1

A. C. Wilson, R. Roelofs, M. Stern, N. Srebro, and B. Recht. The marginal value of adaptive gradient methods in machine learning. In *Neural Information Processing Systems*, 2017. 6.1

A. G. Wilson, C. Dann, and H. Nickisch. Thoughts on massively scalable Gaussian processes. *arXiv*, 2015. 2.2.1

W. Xiao, R. Bhardwaj, R. Ramjee, M. Sivathanu, N. Kwatra, Z. Han, P. Patel, X. Peng, H. Zhao, Q. Zhang, et al. Gandiva: Introspective cluster scheduling for deep learning. In *Symposium on Operating Systems Design and Implementation*, 2018. 3.2

S. Xie, H. Zheng, C. Liu, and L. Lin. SNAS: stochastic neural architecture search. In *International Conference on Learning Representations*, 2019. 1.3.3, 5.1, 5.2.2, 5.1, 5.2.3, 5.5, 5.5, 3, 6.1, 6.2.1, 6.2.2, 6.5, 6.1, 6.3, 6.7.1, 7.1.1, 7.1.2

Y. Xu, L. Xie, X. Zhang, X. Chen, G.-J. Qi, Q. Tian, and H. Xiong. PC-DARTS: Partial channel connections for memory-efficient architecture search. In *International Conference on Learning Representations*, 2020. 5.6, 6, 6.1, 6.5, 6.5.1, 6.1, 6.5.1.1, 6.5.1.2, 6.3, 6.7.1

Y. Yamada, M. Iwamura, and K. Kise. Shakedrop regularization. *arXiv*, 2018. 5.5, 6.1

Z. Yang, Z. Dai, R. Salakhutdinov, and W. W. Cohen. Breaking the softmax bottleneck: A high-rank RNN language model. In *International Conference on Learning Representations*, 2018. 5.2, 5.5.1.1

Z. Yang, Y. Wang, X. Chen, B. Shi, C. Xu, C. Xu, Q. Tian, and C. Xu. CARS: Continuous Evolution for Efficient Neural Architecture Search. *arXiv*, 2019. 6.1

Q. Yao, J. Xu, W.-W. Tu, and Z. Zhu. Differentiable Neural Architecture Search via Proximal Iterations. *arXiv*, 2019. 6.1.1

C. Ying, A. Klein, E. Christiansen, E. Real, K. Murphy, and F. Hutter. NAS-bench-101: Towards

reproducible neural architecture search. In *International Conference on Machine Learning*, 2019. 6.5.2

Y. You, I. Gitman, and B. Ginsburg. Scaling SGD batch size to 32k for imagenet training. *arXiv*, 2017. 3.5.2

Y. You, Z. Zhang, C.-J. Hsieh, J. Demmel, and K. Keutzer. 100-epoch ImageNet Training with AlexNet in 24 Minutes. *arXiv*, 2017. 3.5.2

K. Yu, R. Ranftl, and M. Salzmann. How to train your super-net: An analysis of training heuristics in weight-sharing nas. *arXiv*, 2020. 1.4, 5.6

W. Zaremba, I. Sutskever, and O. Vinyals. Recurrent neural network regularization. *arXiv*, 2014. 3.4.5, 3.4.5

A. Zela, T. Elsken, T. Saikia, Y. Marrakchi, T. Brox, and F. Hutter. Understanding and robustifying differentiable architecture search. In *International Conference on Learning Representations*, 2020a. 6.1

A. Zela, J. Siems, and F. Hutter. NAS-Bench-1Shot1: Benchmarking and dissecting one-shot neural architecture search. In *International Conference on Learning Representations*, 2020b. 1.4, 5.6, 3b, 6.5, 6.5.2, 6.2, 6.7.2

C. Zhang, M. Ren, and R. Urtasun. Graph hypernetworks for neural architecture search. In *International Conference on Learning Representations*, 2019. 1.3.3, 5.1, 5.1, 5.2.3, 5.5, 5.5.2.1, 3

S. Zhang and N. He. On the convergence rate of stochastic mirror descent for nonsmooth nonconvex optimization. *arXiv*, 2018. 6.1.1, 6.4, 6.4, 6.4, 4, 5, 6, 6

Y. Zhang, M. T. Bahadori, H. Su, and J. Sun. Flash: Fast Bayesian optimization for data analytic pipelines. In *International Conference on Knowledge Discovery and Data Mining*, 2016. 4.2

X. Zheng, R. Ji, L. Tang, B. Zhang, J. Liu, and Q. Tian. Multinomial Distribution Learning for Effective Neural Architecture Search. *arXiv*, 2019. 6.1, 7.1.1

J. Zhou, A. Velichkevich, K. Prosvirov, A. Garg, Y. Oshima, and D. Dutta. Katib: A distributed general automl platform on kubernetes. In *Conference on Operational Machine Learning*, 2019. 1.4

B. Zoph and Q. V. Le. Neural Architecture Search with Reinforcement Learning. In *International Conference on Learning Representation*, 2017. 1.3, 1.3.2, 1.3.3, 1.4, 5.1, 5.2.2, 5.2.3, 5.5, 5.2

B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le. Learning transferable architectures for scalable image recognition. In *Conference on Computer Vision and Pattern Recognition*, 2018. 1.3.3, 5.1, 5.2.2, 5.2.3, 5.5, 6.1, 6.3