# AUTOMATED MACHINE LEARNING UNDER RESOURCE CONSTRAINTS

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Chengrun Yang

May 2022

AUTOMATED MACHINE LEARNING UNDER RESOURCE CONSTRAINTS

Chengrun Yang, Ph.D.

Cornell University 2022

Automated machine learning (AutoML) seeks to reduce the human and machine costs of finding machine learning models and hyperparameters with good predictive performance. AutoML is easy with unlimited resources: an exhaustive search across all possible solutions finds the best performing model. This dissertation studies resource-constrained AutoML, in which only limited resources (such as compute or memory) are available for model search. We present a wide variety of strategies for choosing a model under resource constraints, including meta-learning across datasets with low rank matrix and tensor decomposition and experiment design, and efficient neural architecture search (NAS) using weight sharing, reinforcement learning, and Monte Carlo sampling. We propose several AutoML frameworks that realize these ideas, and describe implementations and experimental results.

## BIOGRAPHICAL SKETCH

Chengrun Yang was born in Zhengzhou, Henan, China. He attended Zhengzhou Foreign Language Middle School and Zhengzhou No.1 High School, where he developed a strong interest in science and engineering subjects. He then attended Fudan University in Shanghai, China, where he first majored in Mechanical Science and Engineering and then switched to Physics. He was fascinated by how different fields in physics connect and exhibit the same set of principles in different ways. He then spent the spring semester of his junior year at University of California, Berkeley, where he also got a taste of the fields of electrical engineering and computer science. In his senior year at Fudan, he worked on his undergraduate thesis on circuit partitioning under the supervision of Prof. Fan Yang in the School of Microelectronics.

Chengrun joined the Electrical and Computer Engineering PhD program at Cornell University in Ithaca, NY, USA after his undergraduate studies. He was first intimidated by the remote location of Ithaca, but then quickly fell in love with all the gorgeous waterfalls, fields, lakes, and snow that nature offers, especially when working from home during the COVID-19 pandemic. He was fortunate to have Prof. Madeleine Udell as his dissertation chair, and have Prof. Thorsten Joachims and Prof. Kilian Q. Weinberger on the personal committee. He developed a broad research interest in optimization methods for machine learning, and worked on automated machine learning (AutoML) under resource constraints towards his PhD thesis. He was grateful for the professional and life skills he developed at Cornell, and the company of everyone around who offered help and shared sorrow and joy along the way.

To my family and my friends.

## ACKNOWLEDGEMENTS

It is impossible for me to complete my challenging but fruitful PhD journey without the firm support of many people.

**Advisor:** Madeleine Udell. Since my first day of working with Madeleine in 2017, I have been constantly amazed by her ability to visualize complicated concepts in simple pictures, her skill in drawing connections among objects that seem barely related, her enthusiasm exhibited in teaching and discussing cool ideas, and her attitude of making every tiny part of a work reasonable. Her pursuit of working on projects that are not only methodologically beautiful but also practically useful has greatly influenced my research taste. She spared no effort to ramp me up, and offered me ample guidance in work and life. She is an excellent researcher, collaborator, and educator.

**Other dissertation committee members (alphabetical order):** Thorsten Joachims, Kilian Q. Weinberger. Thorsten and Kilian are pioneers and experts in their respective fields that can see through fogs. I have attended classes offered by both of them. From their words and actions, I have the opportunity to learn how they find a way forward in a field that is under-explored or with intimidating obstacles. They strive to make methodological contributions with broader impacts, and are more than glad to teach students how to do that. They give me sincere and down-to-earth advice when I am faced with difficulties, and applaud me for my achievements. The interactions I have with them are valuable takeaways of my PhD studies.

**Other collaborators (alphabetical order):** Yuji Akimoto, Gabriel Bender, Jerry Chee, Chris De Sa, Lijun Ding, Jicong Fan, Yingjie (Tom) Fei, Da Huang, Dae Won Kim, Pieter-Jan Kindermans, Quoc Le, Hanxiao Liu, Yifeng Lu, Ziyang Wu,

Qiantong Xu. They are brilliant people that together made my work possible. The discussions and debates I had with them were essential to improving the quality of my work. With them, I was not lonely.

**Other Udell group members (alphabetical order):** Zachary Frangella, Yang Guo, Zuhao (Jason) Hua, Huichen Li, Xiaojie Mao, Nandini Nayar, Mike Van Ness, Richard Lanas Phillips, Yiming Sun, Miaolan Xie, Haoyue Yang, Yuqian Zhang, Shipu Zhao, Yuxuan Zhao, Song (Sam) Zhou, Zhengze Zhou. They are brilliant peers in group meetings and offline discussions.

**Other Cornell faculty members (alphabetical order):** Jayadev Acharya, Yudong Chen, A. Kevin Tang. Their valuable advice on research philosophy made my PhD journey less bumpy.

**Other colleagues and friends (alphabetical order):** Being an ECE[1] PhD student with an ORIE[2] advisor and two CS[3] minor committee members, I am fortunate to have the opportunity to work and have fun with a diverse cohort of smart and energetic people: Yu Gan, Emre Gonultas, Kursat Rasim Mestav, Buddhika Nettasinghe, Ziteng Sun, Huanyu Zhang in ECE; Raul Astudillo, Yilun Chen, Yichun Hu, Haici Tan, Matthew Zalesak, Xiangyu Zhang in ORIE; Junwen Bai, Di Chen, Xilun Chen, Junteng Jia, Yucheng Lu, Tianze Shi, Lequn (Luke) Wang, Zikai (Alex) Wen in CS; Yun Liu, Dongping Qi, Lijie Tu, Wangwei Wu, Yu Wu, Yao Yang, Tao Zhang in other fields, among many other people. I learned a lot from them in offices and classrooms, on hiking trails, in gyms, and around boiling hot pots on snow days.

**Internship mentors and managers:** Da Huang, Gabriel Bender, Hanxiao Liu,

[1]Electrical and Computer Engineering
[2]Operations Research and Information Engineering
[3]Computer Science

Pieter-Jan Kindermans, Yifeng Lu, Quoc Le at Google; Parth Gupta and Vamsi Salaka at Amazon; Yang Liu and Puyudi Yang at Facebook. They collectively hosted me in three wonderful summers, offered me a taste of the tech industry, helped me polish my engineering skills in multiple aspects, and gave me sincere advice on career development. They inspired and drove me to think about which kind of works are useful.

**Parents:** Aimin Wang, Hongwei Yang. "You raised me up, so I can stand on mountains." The values and views you teach me in words and actions powered me to move forward, to explore the beautiful world during and beyond my PhD studies.

**Girlfriend:** Lucy Wu. You are the special one that accompanied my struggles and shared my joys. You are a blessing to me, and I am fortunate to meet you in the beautiful Ithaca at a beautiful age.

## TABLE OF CONTENTS

## LIST OF TABLES

## LIST OF FIGURES

CHAPTER 1

# INTRODUCTION

Applications of machine learning have grown tremendously in the past decade due to the increasing feasibility of more expensive model training. However, modern machine learning models are increasingly expensive to train, and the sizes of modern datasets are growing rapidly. These trends have resulted in increasing costs despite the fast growth in computational capabilities of modern machines. For example, the training of a 1.5-billion parameter BERT language model [34] is estimated to cost $80k - $1.6M as of 2020 [112].

Besides the increasing costs of model training, new models are being developed [108, 131], each with its own set of *hyperparameters* to choose from. With a fast-growing number of such choices, practitioners need to specify a *search space* to choose the best candidate from within. These complexities increase both machine and human costs: it takes time, compute, and energy for machines to fit models and do inference, and it takes time for machine learning engineers to choose the appropriate models and hyperparameters.

Exhaustive search evaluates every model in the search space and chooses the best one under some performance metric, such as cross-validation (CV) error or area under the curve (AUC). Exhaustive search is straightforward to implement and returns the model with the best performance among all models in the search space. However, with a growing number of choices, exhaustive search is an (increasingly) poor choice for model selection and hyperparameter tuning: the time (or computation) required is proportional to the number of possible models, and choosing the model with the best performance can also result in overfitting [6].

Automated machine learning (AutoML) aims to automate model selection and hyperparameter tuning. It encompasses approaches that use *surrogate models* to learn the relationship among datasets, machine learning models, and hyperparameters. The goal is to find a model more quickly than exhaustive search, and one that generalizes better. To this end, AutoML approaches tend to be:

(a) *Cheap to run*: The AutoML approach should not take too much time or compute to train and test; it should be much cheaper than an exhaustive search over the search space.

(b) *Easy to tune*: The AutoML approach should not have hyperparameters that are more difficult to tune than the hyperparameters of machine learning models AutoML searches over.

(c) *Able to find cheap models*: The AutoML approach should find models that satisfy the resource constraint.

This thesis describes two major types of AutoML methods to achieve these goals:

(a) Meta-learning across datasets for model selection: These methods select machine learning models on a new dataset using information about how models perform on other datasets, using ideas from collaborative filtering. They collect performance information to form an error matrix (or tensor). Low rank matrix (or tensor) decomposition serves as a surrogate model to learn embeddings for datasets and models. To choose a good model for a new dataset, an initial set of cheap but informative models are evaluated, and the performance of other models is inferred by the low rank surrogate model. To select the set of cheap but informative models, we

solve a resource-constrained experiment design problem that seeks a low-variance embedding for the new dataset.

(b) Neural architecture search (NAS) on individual datasets: To select a cheap neural network architecture that performs well on a given dataset, this method uses reinforcement learning (RL) and Monte-Carlo (MC) sampling to select among candidate network components. The method interleaves weight training and RL update steps, so that the network weights of promising architectures determined by the RL controller are trained more often. *Weight-sharing* decreases the cost of training each architecture and reduces the size of the RL search space and make RL easier: each architecture is viewed as a *child network* of a *SuperNet*; weights for the child network are initialized by the weights of the same parameters of the SuperNet. Architectures are sampled from layer-wise distributions over different sizes – a *factorized* search space. The factorized structure can introduce bias: the RL controller learns the average effect of each layer size but is blind to interactions, and so can miss the global optimal solution. To mitigate this problem, we propose updating the RL controller by a rejection sampling mechanism that only accepts architectures that obey the resource constraint. A corresponding MC-based-correction applied to the RL policy gradient updates reduces the cost of rejection sampling.

Table 1.1 summarizes the objectives and constraints of the works to be presented in this dissertation. Since the machine learning models to be selected in Oboe [141] can be regarded as a special case of the machine learning pipelines in TensorOboe [144], we list them in the same row within the table.

Table 1.1: Objectives and constraints for resource-constrained AutoML

| Chapter | Work | Objective: find the best ... | Constraint |
|---|---|---|---|
| 2, 3 | Oboe [141], TensorOboe [144] | ML pipelines | maximum time allowed to evaluate pipelines |
| 4 | PEPPP [145] | low-precision configuration | largest allowable memory for network training |
| 5 | TabNAS [142] | architecture | number of parameters of the found architecture |

## 1.1 Previous work

There have been multiple pathways for AutoML. Naive approaches like grid search and random search remain popular candidates. In many settings, when the search space has multiple hyperparameters with different importance for performance, random search may outperform grid search in the number of evaluations [14]. Some works form differentiable objectives, and first-order approaches like gradient descent to optimize over hyperparameters [12] and choose optimizers [2].

Surrogate models like Gaussian processes [13, 116, 7] or tree-based models [13] are popular choices for hyperparameter tuning. There are also works that combine these surrogate models into open-source packages like Auto-WEKA [125] and Ax [1], or use these approaches for further fine-tuning [42].

As a standard practice in recommender systems, collaborative filtering makes recommendations for a new task based on its similarity with previous tasks on which we have more information. In AutoML, collaborative filtering recommends models for a new dataset based on how well these models work on similar datasets. Each AutoML approach needs to characterize dataset sim-

[1] https://github.com/facebook/Ax

ilarity with a certain surrogate model. One line of work relies on dataset meta-features to characterize datasets [7, 43, 42]. The meta-features are simple, statistical or landmarking [102] metrics of the dataset. Other approaches avoid (only using) meta-features [46, 137], or use embeddings from dataset metadata to better characterize dataset similarity [38]. The types of surrogate models include nearest neighbors [42], matrix factorization [46], and Gaussian processes [7].

Some other AutoML approaches use evolutionary search [98, 25] or Monte-Carlo Tree Search [37] to explore the search space, and gradually dedicate more resources to more promising as the search proceeds.

Neural architecture search (NAS) [156] stems from the resurgence of deep learning. It focuses on tuning architectural hyperparameters in neural networks, like hidden layer sizes in feedforward networks or convolutional kernel size in convolutional networks. There are two types of surrogate models in NAS. The first type characterizes the performance of candidate networks, to avoid training each candidate network from scratch: a prohibitive practice because of its huge resource consumption. A popular approach is to share weights across candidate networks by training the weights in a SuperNet that includes all trainable weights [10, 86].

The other types of surrogate models transfer knowledge across architectures. Examples include reinforcement learning [156, 20, 11] that learns probability distributions over candidates, and parametric models [149, 135] that directly predict network performance.

It is worth noting that for both AutoML on general model types and NAS, there have been benchmarks that attempt to standardize the practice of experi-

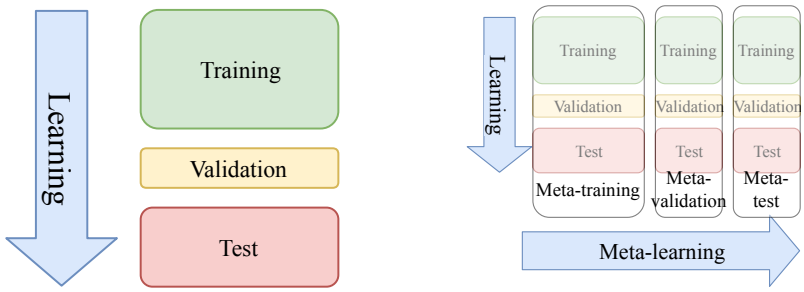
mentation and make different works comparable. The OpenML AutoML benchmarking framework [48] provides a list of datasets from different domains and with various difficulty levels. NAS benchmarks exhaustively evaluate network performance on a search space (e.g., [147, 35]), to not only provide a comparison standard but also ease the burden of resource-limited researchers. Although the generality of such benchmarks is often under debate, these benchmarks are valuable contributions to the goal of making AutoML and NAS research more accessible to a broader population.

## 1.2 Notation and terminology

**Math basics.** We define $[n] = \{1, \cdots, n\}$ for a positive integer $n$. With a Boolean variable $X$, the indicator function $\mathbb{1}(X)$ equals 1 if $X$ is true, and 0 otherwise. With a scalar variable $x$, we use $x_+$ to denote $\max\{x, 0\}$. $\subseteq$ and $\subset$ denote subset and strict subset, respectively.

**Meta-learning.** Meta-learning is the process of learning across individual datasets or problems, which are subsystems on which standard learning is performed [81]. Just as standard machine learning must avoid overfitting, experiments testing AutoML systems must avoid meta-overfitting! We divide our set of datasets into meta-training, meta-validation and meta-test sets, and report results on the meta-test set. Each of the three phases in meta-learning — meta-training, meta-validation and meta-test — is a standard learning process that includes training, validation and test.

**Linear algebra.** We define $[n] = \{1, \cdots, n\}$ for $n \in \mathbb{Z}$, and denote *vector*, *matrix*, and *tensor* variables respectively by lowercase letters ($x$), capital letters ($X$) and Euler




(a) Learning (b) Meta-learning

Figure 1.1: Learning vs meta-learning.

script letters ($\mathcal{X}$). The order of a tensor is the number of dimensions; matrices are order-two tensors. Each dimension is called a mode. All vectors are column vectors. Given a matrix $A \in \mathbb{R}^{m\times n}$, $A_{i,:}$ and $A_{:,j}$ denote the $i$th row and $j$th column of $A$, respectively. We define $[n] = \{1, \cdots, n\}$ for $n \in \mathbb{Z}$. Given an ordered set $\mathcal{S} = \{s_1, \cdots, s_k\}$ where $s_1 < \cdots < s_k \in [n]$, we write $A_{:\mathcal{S}} = \begin{bmatrix} A_{:,s_1} & A_{:,s_2} & \cdots & A_{:,s_k} \end{bmatrix}$. A fiber is a one-dimensional section of a tensor $\mathcal{X}$, defined by fixing every index but one; for example, one fiber of the order-3 tensor $\mathcal{X}$ is $X_{:jk}$. Fibers of a tensor are analogous to rows and columns of a matrix. A slice is an $(N-1)$-dimensional section of an order-$N$ tensor $\mathcal{X}$. The mode-$n$ matricization of $\mathcal{X}$, denoted as $\mathcal{X}^{(n)}$, is a matrix whose columns are the mode-$n$ fibers of $\mathcal{X}$. $\mathcal{X}$ has *multilinear rank* $(r_1, r_2, \cdots)$ if $r_n$ is the rank of $\mathcal{X}^{(n)}$. For example, given an order-3 tensor $\mathcal{X} \in \mathbb{R}^{I\times J\times K}$, we have $X^{(1)} \in \mathbb{R}^{I\times(J\times K)}$, and $\mathcal{X}$ has multilinear rank $(r_1, r_2, r_3)$ if $\mathcal{X}^{(n)}$ has rank $r_n$ for $n \in [3]$. We denote the *n-mode product* of a tensor $\mathcal{X} \in \mathbb{R}^{I_1\times I_2\times\cdots I_N}$ with a matrix $U \in \mathbb{R}^{J\times I_n}$ by $\mathcal{X} \times_n U \in \mathbb{R}^{I_1\times\cdots I_{n-1}\times J\times I_{n+1}\times\cdots I_N}$; the $(i_1, i_2, \cdots, i_{n-1}, j, i_{n+1}, \cdots, i_N)$-th entry is $\Sigma_{i_n=1}^{I_n} x_{i_1 i_2\cdots i_{n-1} i_n i_{n+1}\cdots i_N} u_{j i_n}$. Given two tensors with the same shape, we use $\odot$ to denote their entrywise product. Given an ordered set $\mathcal{S} = \{s_1, \cdots, s_k\}$ where $s_1 < \cdots < s_k \in [n]$, we write $A_{:\mathcal{S}} = [A_{:,s_1}, A_{:,s_2}, \cdots, A_{:,s_k}]$; given an ordinary set $S$, we use $A_{:S}$ to denote $A_{:\mathcal{S}}$, in which $\mathcal{S}$ is the ordered version of set $S$. The Euclidean norm of a vector $a \in \mathbb{R}^n$ is $\|a\| := \sqrt{\sum_{i=1}^n a_i^2}$. To denote a part of a matrix $A \in \mathbb{R}^{n\times d}$,

we use a colon to denote the varying dimension: $A_{i,:}$ and $A_{:,j}$ (or $a_j$) denote the $i$th row and $j$th column of $A$, respectively, and $A_{ij}$ denotes the $(i, j)$-th entry. Given two vectors $x, y \in \mathbb{R}^n$, $x \leq y$ means $x_i \leq y_i$ for each $i \in [n]$. Given a matrix $A \in \mathbb{R}^{n \times d}$ and a set of observed indices as $\Omega \subseteq [n] \times [d]$, the partially observed matrix $P_\Omega(A) \in \mathbb{R}^{n \times d}$ has entries $(P_\Omega(A))_{ij} = A_{ij}$ if $(i, j) \in \Omega$, and 0 otherwise.

**Parametric hierarchy.** We distinguish between three kinds of parameters:

- *Parameters* of a model (e.g., the splits in a decision tree) are obtained by training the model.
- *Hyperparameters* of an algorithm (e.g., the maximum depth of a decision tree) govern the training procedure. We use the word *model* to refer to an algorithm together with a particular choice of hyperparameters.
- *Hyper-hyperparameters* of an AutoML system (e.g., the total time budget for the system) govern the model search.

**Pipeline component.** A pipeline component is a model or model type. Examples include missing entry imputers, dimensionality reducers, supervised learners, and data visualizers. We consider the following components in this work:

- *Data imputer*: A preprocessor that fills in missing entries.
- *Encoder*: A transformer that converts categorical features to numerical codes. Here, we consider encoding categoricals as integers or with a one-hot encoder.
- *Standardizer*: A standardizer centers and rescales data.
- *Dimensionality reducer*: A transformer that reduces the dimensionality of

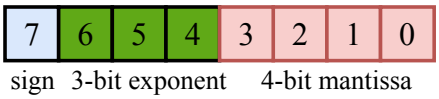



Figure 1.2: An 8-bit floating point number representing $(-1)^{\text{sign}} \cdot 2^{\text{exponent}-7} \cdot 1.b_3b_2b_1b_0$.

the dataset by either creating new features (like PCA) or subsampling features.

- *Estimator*: The supervised learner. For the classification tasks in this work, estimators are classifiers.

**Time target and time budget.** In Oboe systems [141, 144], the time target refers to the anticipated time spent running models to infer latent features of each fixed dimension and can be exceeded. However, the runtime does not usually deviate much from the target since our model runtime prediction works well. The time budget refers to the total time limit for Oboe and is never exceeded.

**Low-precision formats.** We use floating point numbers for low-precision training in this work. As an example, Figure 1.2 shows an 8-bit floating point number with 3 exponent bits and 4 mantissa bits. A specific low-precision representation with a certain number of bits for each part is called a *low-precision format*. We may use different low-precision formats for the weights, activations, and optimizer. In the case of using two formats to train and represent a neural network (as discussed in Section 4.2.1), these two formats are called *Format A* and *B*. A specific combination of these two formats is a hyperparameter setting of the neural network, and we call it a *low-precision configuration*.

**Pareto frontier.** Multi-objective optimization simultaneously minimizes $n$ costs $\{c_i\}_{i\in[n]}$. A feasible point $c^{(1)} = (c_1^{(1)}, \cdots, c_n^{(1)})$ is *Pareto optimal* if for any other feasible

| datasets | low-precision configurations a | b | c | d |
|---|---|---|---|---|
| 1 | 0.85 | 0.71 | 0.32 | 0.28 |
| 2 | 0.80 | 0.80 | 0.80 | 0.69 |
| 3 | 0.80 | 0.68 | 0.46 | 0.45 |

(a) error matrix $E$

| datasets | low-precision configurations a | b | c | d |
|---|---|---|---|---|
| 1 | 181 | 269 | 272 | 295 |
| 2 | 180 | 269 | 272 | 295 |
| 3 | 590 | 924 | 927 | 1032 |

(b) memory matrix $M$ (MB)

Figure 1.3: Example error and memory matrices for some datasets and low-precision configurations. Dataset 1: CIFAR-10, 2: CIFAR-100 (fruit and vegetables), 3: ImageNet-stick [33]. Configuration Format A (exponent bits, mantissa bits), Format B (exponent bits, mantissa bits). a: (3, 1), (6, 7); b: (3, 4), (7, 7); c: (4, 3), (8, 7); d: (5, 3), (6, 7).

point $c^{(2)} = (c_1^{(2)}, \cdots, c_n^{(2)})$, $c^{(2)} \preceq c^{(1)}$ implies $c^{(2)} = c^{(1)}$ [18]. The set of Pareto optimal points is the *Pareto frontier*.

**(PEPPP) Tasks, datasets, measurements and evaluations.** A *task* carries out a process (classification, regression, image segmentation, etc.) on a *dataset*. Given a deep learning model and a dataset, the training and testing of the model on the dataset is called a *measurement*. In our low-precision context, we *evaluate* a configuration on a dataset to make a measurement.

**(PEPPP) Error matrix and memory matrix.** Given a neural network, the errors of different low-precision configurations on meta-training datasets form an *error matrix*, whose $(i, j)$-th entry $E_{ij}$ is the test error of the $j$th configuration on the $i$th dataset. To compute the error matrix, we split the $i$-th dataset into training and test subsets (or use a given split), train the neural network at the $j$-th configuration on the training subset, and evaluate the test error on the test subset. The memory required for each measurement forms a *memory matrix* $M$, which has the same shape as the corresponding error matrix. Example error and memory matrices are shown in Figure 1.3.

CHAPTER 2

# OBOE: COLLABORATIVE FILTERING FOR FAST MACHINE LEARNING MODEL SELECTION

This chapter presents Oboe [141], an AutoML framework that uses low rank matrix factorization and experiment design to select promising machine learning models in the meta-learning setting.

## 2.1 Introduction

It is often difficult to find the best algorithm and hyperparameter settings for a new dataset, even for experts in machine learning or data science. The large number of machine learning algorithms and their sensitivity to hyperparameter values make it practically infeasible to enumerate all configurations. Automated machine learning (AutoML) seeks to efficiently automate the selection of model (e.g., [42, 25, 46]) or pipeline (e.g., [36]) configurations, and has become more important as the number of machine learning applications increases.

We propose an algorithmic system, OBOE [1], that provides an initial tuning for AutoML: it selects a good algorithm and hyperparameter combination from a discrete set of options. The resulting model can be used directly, or the hyperparameters can be tuned further. Briefly, OBOE operates as follows.

During an offline training phase, it forms a matrix of the cross-validated errors of a large number of supervised-learning models (algorithms together with hyperparameters) on a large number of datasets. It then fits a low rank model to this matrix to learn latent low-dimensional meta-features for the models and

[1] The eponymous musical instrument plays the initial note to tune an orchestra.

datasets. Our optimization procedure ensures these latent meta-features best predict the cross-validated errors, among all bilinear models.

To find promising models for a new dataset, OBOE chooses a set of fast but informative models to run on the new dataset and uses their cross-validated errors to infer the latent meta-features of the new dataset. Given more time, OBOE repeats this procedure using a higher rank to find higher-dimensional (and more expressive) latent features. Using a low rank model for the error matrix is a very strong structural prior.

This system addresses two important problems: 1) *Time-constrained initialization*: how to choose a promising initial model under time constraints. OBOE adapts easily to short times by using a very low rank and by restricting its experiments to models that will run very fast on the new dataset. 2) *Active learning*: how to improve on the initial guess given further computational resources. OBOE uses extra time by allowing higher ranks and more expensive computational experiments, accumulating its knowledge of the new dataset to produce more accurate (and higher-dimensional) estimates of its latent meta-features.

OBOE uses collaborative filtering for AutoML, selecting models that have worked well on similar datasets, as have many previous methods including [7, 121, 148, 42, 93, 29]. In collaborative filtering, the critical question is how to characterize dataset similarity so that training datasets "similar" to the test dataset faithfully predict model performance. One line of work uses dataset meta-features — simple, statistical or landmarking metrics — to characterize datasets [102, 43, 42, 46, 29]. Other approaches (e.g., [137]) avoid meta-features. Our approach builds on both of these lines of work. OBOE relies on model performance to characterize datasets, and the low rank representations it learns

for each dataset may be seen (and used) as latent meta-features. Compared to AutoML systems that compute meta-features of the dataset before running any models, the flow of information in OBOE is exactly opposite: OBOE uses only the performance of various models on the datasets to compute lower dimensional latent meta-features for models and datasets.

The active learning subproblem is to gain the most information to guide further model selection. Some approaches choose a function class to capture the dependence of model performance on hyperparameters; examples are Gaussian processes [106, 116, 13, 46, 110, 59, 89, 120], sparse Boolean functions [58] and decision trees [8, 65]. OBOE chooses the set of bilinear models as its function class: predicted performance is linear in each of the latent model and dataset meta-features.

Bilinearity seems like a rather strong assumption, but confers several advantages. Computations are fast and easy: we can find the global minimizer by PCA, and can infer the latent meta-features for a new dataset using least squares. Moreover, recent theoretical work suggests that this model class is more general than it appears: roughly, and under a few mild technical assumptions, any $m \times n$ matrix with independent rows and columns whose entries are generated according to a fixed function (here, the function computed by training the model on the dataset) has an approximate rank that grows as $\log(m + n)$ [128]. Hence large data matrices tend to look low rank.

Originally, the authors conceived of OBOE as a system to produce a good set of initial models, to be refined by other local search methods, such as Bayesian optimization. However, in our experiments, we find that OBOE's performance, refined by fitting models of ever higher rank with ever more data, actually im-

proves faster than competing methods that use local search methods more heavily.

One key component of our system is the prediction of model runtime on new datasets. Many authors have previously studied algorithm runtime prediction using a variety dataset features [66], via ridge regression [63], neural networks [115], Gaussian processes [64], and more. Several measures have been proposed to trade-off between accuracy and runtime [80, 15]. We predict algorithm runtime using only the number of samples and features in the dataset. This model is particularly simple but surprisingly effective.

Classical experiment design (ED) [132, 95, 68, 104, 18] selects features to observe to minimize the variance of the parameter estimate, assuming that features depend on the parameters according to known, linear, functions. OBOE's bilinear model fits this paradigm, and so ED can be used to select informative models. Budget constraints can be added, as we do here, to select a small number of promising machine learning models or a set predicted to finish within a short time budget [76, 152].

## 2.2 Methodology

### 2.2.1 Model performance prediction

It can be difficult to determine *a priori* which meta-features to use so that algorithms perform similarly well on datasets with similar meta-features. Also, the computation of meta-features can be expensive. To infer model performance on

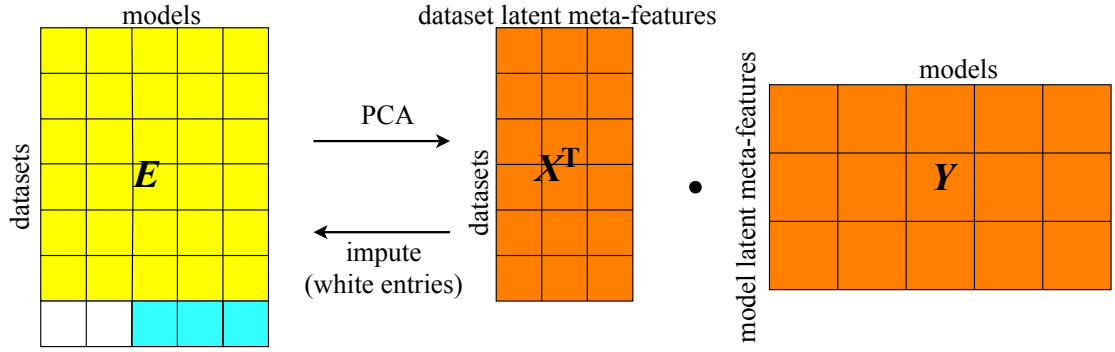



Figure 2.1: Illustration of model performance prediction via the error matrix $E$ (yellow blocks only). Perform PCA on the error matrix (offline) to compute dataset ($X$) and model ($Y$) latent meta-features (orange blocks). Given a new dataset (row with white and blue blocks), pick a subset of models to observe (blue blocks). Use $Y$ together with the observed models to impute the performance of the unobserved models on the new dataset (white blocks).

a dataset without any expensive meta-feature calculations, we use collaborative filtering to infer latent meta-features for datasets.

As shown in Figure 2.1, we construct an empirical error matrix $E \in \mathbb{R}^{m \times n}$, where every entry $E_{ij}$ records the cross-validated error of model $j$ on dataset $i$. Empirically, $E$ has approximately low rank: Figure 2.2 shows the singular values $\sigma_i(E)$ decay rapidly as a function of the index $i$. This observation serves as foundation of our algorithm. The value $E_{ij}$ provides a noisy but unbiased estimate of the true performance of a model on the dataset: $\mathbb{E}E_{ij} = \mathcal{A}_j(\mathcal{D}_i)$.

To denoise this estimate, we approximate $E_{ij} \approx x_i^\top y_j$ where $x_i$ and $y_j$ minimize $\sum_{i=1}^{m} \sum_{j=1}^{n} (E_{ij} - x_i^\top y_j)^2$ with $x_i, y_j \in \mathbb{R}^k$ for $i \in [M]$ and $j \in [N]$; the solution is given by PCA. Thus $x_i$ and $y_j$ are the latent meta-features of dataset $i$ and model $j$, respectively. The rank $k$ controls model fidelity: small $k$s give coarse approximations, while large $k$s may overfit. We use a doubling scheme to choose $k$ within time budget; see Section 2.3.2 for details.

Given a new meta-test dataset, we choose a subset $\mathcal{S} \subseteq [N]$ of models and

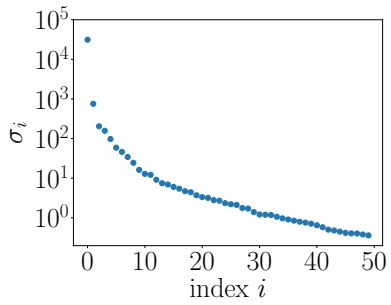



Figure 2.2: Singular value decay of an error matrix. The entries are calculated by 5-fold cross validation of machine models (listed in Appendix A.1, Table A.1) on midsize OpenML datasets.

observe performance $e_j$ of model $j$ for each $j \in \mathcal{S}$. A good choice of $S$ balances information gain against time needed to run the models; we discuss how to choose $S$ in Section 2.2.3. We then infer latent meta-features for the new dataset by solving the least squares problem: minimize $\sum_{j\in\mathcal{S}}(e_j - \hat{x}^\top y_j)^2$ with $\hat{x} \in \mathbb{R}^k$. For all unobserved models, we predict their performance as $\hat{e}_j = \hat{x}^\top y_j$ for $j \notin \mathcal{S}$.

### 2.2.2 Runtime prediction

Estimating model runtime allows us to trade off between running slow, informative models and fast, less informative models. We use a simple method to estimate runtimes, using polynomial regression on $n^{\mathcal{D}}$ and $p^{\mathcal{D}}$, the numbers of data points and features in $\mathcal{D}$, and their logarithms, since the theoretical complexities of machine learning algorithms we use are $O((n^{\mathcal{D}})^3, (p^{\mathcal{D}})^3, (\log(n^{\mathcal{D}}))^3)$. Hence we fit an independent polynomial regression model for each model:

$$f_j = \text{argmin}_{f_j\in\mathcal{F}} \sum_{i=1}^{M} \left(f_j(n^{\mathcal{D}_i}, p^{\mathcal{D}_i}, \log(n^{\mathcal{D}_i})) - t_j^{\mathcal{D}_i}\right)^2, \; j \in [n]$$

where $t_j^{\mathcal{D}}$ is the runtime of machine learning model $j$ on dataset $\mathcal{D}$, and $\mathcal{F}$ is the set of all polynomials of order no more than 3. We denote this procedure by $f_j =$ `fit_runtime`$(n, p, t)$.

We observe that this model predicts runtime within a factor of two for half of the machine learning models on more than 75% midsize OpenML datasets, and within a factor of four for nearly all models, as shown in Section 2.5.2 and visualized in Figure 2.6.

### 2.2.3 Time-constrained information gathering

To select a subset $\mathcal{S}$ of models to observe, we adopt an approach that builds on classical experiment design: we suppose fitting each machine learning model $j \in [n]$ returns a linear measurement $x^\top y_j$ of $x$, corrupted by Gaussian noise. To estimate $x$, we would like to choose a set of observations $y_j$ that span $\mathbb{R}^k$ and form a well-conditioned submatrix, but that corresponds to models which are fast to run. In passing, we note that the pivoted QR algorithm on the matrix $Y$ (heuristically) finds a well conditioned set of $k$ columns of $Y$. However, we would like to find a method that is runtime-aware.

Our experiment design (ED) procedure minimizes a scalarization of the covariance of the estimated meta-features $\hat{x}$ of the new dataset subject to runtime constraints [132, 95, 68, 104, 18]. Formally, define an indicator vector $v \in \{0,1\}^n$, where entry $v_j$ indicates whether to fit model $j$. Let $\hat{t}_j$ denote the predicted runtime of model $j$ on a meta-test dataset, and let $y_j$ denote its latent meta-features, for $j \in [n]$. Now relax to allow $v \in [0,1]^n$ to allow for non-Boolean values and solve the optimization problem

$$\begin{array}{ll} \text{minimize} & \log\det\left(\sum_{j=1}^n v_j y_j y_j^\top\right)^{-1} \\ \text{subject to} & \sum_{j=1}^n v_j \hat{t}_j \le \tau \\ & v_j \in [0,1], \forall\, j \in [n] \end{array} \tag{2.1}$$

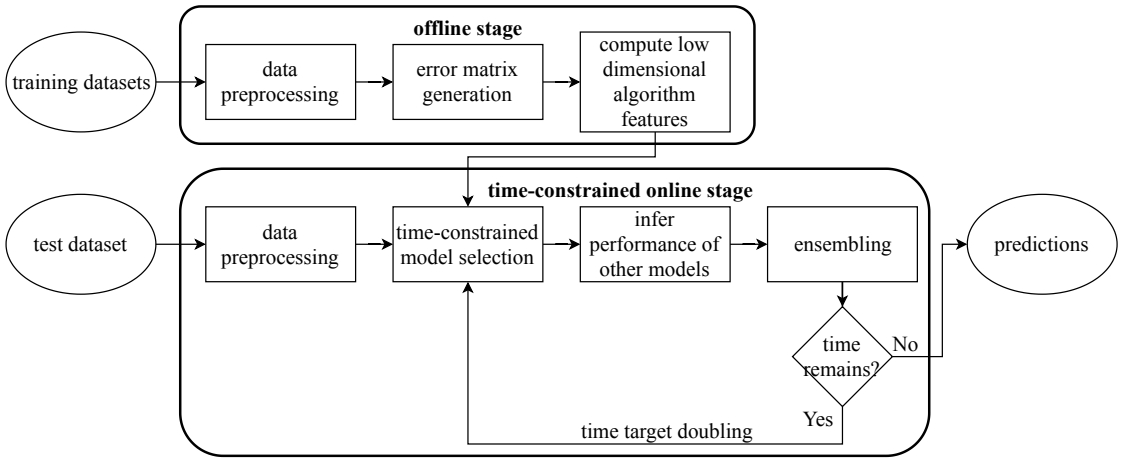



Figure 2.3: Diagram of data processing flow in the Oboe system.

with variable $v \in \mathbb{R}^n$. We call this method ED (time). Scalarizing the covariance by minimizing the determinant is called D-optimal design. Several other scalarizations can also be used, including covariance norm (E-optimal) or trace (A-optimal). Replacing $t_i$ by 1 gives an alternative heuristic that bounds the *number* of models fit by $\tau$; we call this method ED (number).

Problem 2.1 is a convex optimization problem, and we obtain an approximate solution by rounding the largest entries of $v$ up to 1 until the selected models exceed the time limit $\tau$. Let $\mathcal{S} \subseteq [n]$ be the set of indices of $e$ that we choose to observe, i.e. the set such that $v_s$ rounds to 1 for $s \in \mathcal{S}$. We denote this process by $\mathcal{S} = \texttt{min\_variance\_ED}(\hat{t}, \{y_j\}_{j=1}^n, \tau)$.

## 2.3 The Oboe system

Shown in Figure 2.3, the Oboe system can be divided into offline and online stages. The offline stage is executed only once and explores the space of model performance on meta-training datasets. Time taken on this stage does not affect

the runtime of Oboe on a new dataset; the runtime experienced by user is that of the online stage.

One advantage of Oboe is that the vast majority of the time in the online phase is spent training standard machine learning models, while very little time is required to decide which models to sample. Training these standard machine learning models requires running algorithms on datasets with thousands of data points and features, while the meta-learning task — deciding which models to sample — requires only solving a small least-squares problem.

### 2.3.1 Offline stage

The $(i, j)$th entry of error matrix $E \in \mathbb{R}^{m \times n}$, denoted as $E_{ij}$, records the performance of the $j$th model on the $i$th meta-training dataset. We generate the error matrix using the *balanced error rate* metric, the average of false positive and false negative rates across different classes. At the same time we record runtime of machine learning models on datasets. This is used to fit runtime predictors described in Section 2.2. Pseudocode for the offline stage is shown as Algorithm 1.

### 2.3.2 Online stage

Recall that we repeatly double the time target of each round until we use up the total time budget. Thus each round is a subroutine of the entire online stage and is shown as Algorithm 2, `fit_one_round`.

- **Time-constrained model selection (`fit_one_round`)** Our active learning

**Algorithm 1** Offline Stage

**Input:** meta-training datasets $\{\mathcal{D}_i\}_{i=1}^m$, models $\{\mathcal{A}_j\}_{j=1}^n$, algorithm performance metric $\mathcal{M}$

**Output:** error matrix $E$, runtime matrix $T$, fitted runtime predictors $\{f_j\}_{j=1}^n$

**for** $i = 1, 2, \cdots, m$ **do**
  $n^{\mathcal{D}_i}, p^{\mathcal{D}_i} \leftarrow$ number of data points and features in $\mathcal{D}_i$
  **for** $j = 1, 2, \cdots, n$ **do**
    $E_{ij} \leftarrow$ error of model $\mathcal{A}_j$ on dataset $\mathcal{D}_i$ according to metric $\mathcal{M}$
    $T_{ij} \leftarrow$ observed runtime for model $\mathcal{A}_j$ on dataset $\mathcal{D}_i$
  **end for**
**end for**
**for** $j = 1, 2, \cdots, n$ **do**
  fit $f_j =$ `fit_runtime`$(n, p, T_j)$
**end for**

procedure selects a fast and informative collection of models to run on the meta-test dataset. Oboe uses the results of these fits to estimate the performance of all other models as accurately as possible. The procedure is as follows. First predict model runtime on the meta-test dataset using fitted runtime predictors. Then use experiment design to select a subset $\mathcal{S}$ of entries of $e$, the performance vector of the test dataset, to observe. The observed entries are used to compute $\hat{x}$, an estimate of the latent meta-features of the test dataset, which in turn is used to predict every entry of $e$. We build an ensemble out of models predicted to perform well within the time target $\tilde{\tau}$ by means of greedy forward selection [24, 23]. We denote this subroutine as $\tilde{A} =$`ensemble_selection`$(\mathcal{S}, e_{\mathcal{S}}, z_{\mathcal{S}})$, which takes as input the set of base learners $\mathcal{S}$ with their cross-validation errors $e_{\mathcal{S}}$ and predicted labels $z_{\mathcal{S}} = \{z_s | s \in \mathcal{S}\}$, and outputs ensemble learner $\tilde{A}$. The hyperparameters used by models in the ensemble can be tuned further, but in our experiments we did not observe substantial improvements from further hyperparameter tuning.

- **Time target doubling** To select rank $k$, Oboe starts with a small initial

**Algorithm 2** `fit_one_round`$(\{y_j\}_{j=1}^n, \{f_j\}_{j=1}^n, \mathcal{D}_{tr}, \tilde{\tau})$

**Input:** model latent meta-features $\{y_j\}_{j=1}^n$, fitted runtime predictors $\{f_j\}_{j=1}^n$, training fold of the meta-test dataset $\mathcal{D}_{\text{tr}}$, number of best models $N$ to select from the estimated performance vector, time target for this round $\tilde{\tau}$

**Output:** ensemble learner $\tilde{A}$

**for** $j = 1, 2, \cdots, n$ **do**
  $\hat{t}_j \leftarrow f_j(n^{\mathcal{D}_{\text{tr}}}, p^{\mathcal{D}_{\text{tr}}})$
**end for**
$\mathcal{S} =$ `min_variance_ED`$(\hat{t}, \{y_j\}_{j=1}^n, \tilde{\tau})$
**for** $k = 1, 2, \cdots, |\mathcal{S}|$ **do**
  $e_{\mathcal{S}_k} \leftarrow$ cross-validation error of model $\mathcal{A}_{\mathcal{S}_k}$ on $\mathcal{D}_{\text{tr}}$
**end for**
$\hat{x} \leftarrow (\begin{bmatrix} y_{\mathcal{S}_1} & y_{\mathcal{S}_2} & \cdots & y_{\mathcal{S}_{|\mathcal{S}|}} \end{bmatrix}^\top)^\dagger e_{\mathcal{S}}$
$\hat{e} \leftarrow \begin{bmatrix} y_1 & y_2 & \cdots & y_n \end{bmatrix}^\top \hat{x}$
$\mathcal{T} \leftarrow$ the $N$ models with lowest predicted errors in $\hat{e}$
**for** $k = 1, 2, \cdots, |\mathcal{T}|$ **do**
  $e_{\mathcal{T}_k}, z_{\mathcal{T}_k} \leftarrow$ cross-validation error of model $\mathcal{A}_{\mathcal{T}_k}$ on $\mathcal{D}_{\text{tr}}$
**end for**
$\tilde{A} \leftarrow$`ensemble_selection`$(\mathcal{T}, e_{\mathcal{T}}, z_{\mathcal{T}})$

rank along with a small time target, and then doubles the time target for `fit_one_round` until the elapsed time reaches half of the total budget. The rank $k$ increments by 1 if the validation error of the ensemble learner decreases after doubling the time target, and otherwise does not change. Since the matrices returned by PCA with rank $k$ are submatrices of those returned by PCA with rank $l$ for $l > k$, we can compute the factors as submatrices of the $m$-by-$n$ matrices returned by PCA with full rank $\min(m, n)$ [49]. The pseudocode is shown as Algorithm 3.

## 2.4 Python implementation

Code for the Oboe system is at `https://github.com/udellgroup/oboe`. We build the system on top of scikit-learn [101] methods, and design the Oboe

**Algorithm 3** Online Stage

**Input:** error matrix $E$, runtime matrix $T$, meta-test dataset $\mathcal{D}$, total time budget $\tau$, fitted runtime predictors $\{f_j\}_{j=1}^n$, initial time target $\tilde{\tau}_0$, initial approximate rank $k_0$

**Output:** ensemble learner $\tilde{A}$

$x_i, y_j \leftarrow \arg\min \sum_{i=1}^m \sum_{j=1}^n (E_{ij} - x_i^\top y_j)^2$, $x_i \in \mathbb{R}^{\min(m,n)}$ for $i \in [M]$ , $y_j \in \mathbb{R}^{\min(m,n)}$ for $j \in [N]$

$\mathcal{D}_{\text{tr}}, \mathcal{D}_{\text{val}}, \mathcal{D}_{\text{te}} \leftarrow$ training, validation and test folds of $\mathcal{D}$

$\tilde{\tau} \leftarrow \tilde{\tau}_0$

$k \leftarrow k_0$

**while** $\tilde{\tau} \le \tau/2$ **do**

  $\{\tilde{y}_j\}_{j=1}^n \leftarrow k$-dimensional subvectors of $\{y_j\}_{j=1}^n$

  $\tilde{A} \leftarrow$ `fit_one_round`$(\{\tilde{y}_j\}_{j=1}^n, \{f_j\}_{j=1}^n, \mathcal{D}_{\text{tr}}, \tilde{\tau})$

  $e'_{\tilde{A}} \leftarrow \tilde{A}(\mathcal{D}_{\text{val}})$

  **if** $e'_{\tilde{A}} < e_{\tilde{A}}$ **then**

    $k \leftarrow k + 1$

  **end if**

  $\tilde{\tau} \leftarrow 2\tilde{\tau}$

  $e_{\tilde{A}} \leftarrow e'_{\tilde{A}}$

**end while**

classes and methods to have the same style as scikit-learn API. For example, here is the basic building block to create an `AutoLearner` instance for AutoML, fit on the training features `x_train` and training labels `y_train`, and predict on the test features `x_test`.

```
from oboe import AutoLearner

m = AutoLearner(runtime_limit=60)

m.fit(x_train, y_train)

y_test_pred = m.predict(x_test)
```

The designs are:

- The line of class instantiation `m = AutoLearner()` finishes the offline stage (Algorithm 1).
- Same as scikit-learn, the feature matrices `x_train` and `x_test` should

have shapes $n_{\text{train}}$-by-$d$ and $n_{\text{test}}$-by-$d$, respectively: each row corresponds to a data point and each column corresponds to a feature. The label vector `y_train` should be a one-dimensional array.

- The method `AutoLearner.fit()` finishes the online stage (multiple rounds of Algorithm 3) with the time limit for the online stage specified in the `AutoLearner.runtime_limit` attribute.
- The method `AutoLearner.predict()` uses the fitted ensemble to predict on each test point in `x_test`.

API documentations of more advanced funtionalities (e.g., stacking algorithm, verbose level, whether to build an ensemble) can be find at the public GitHub repository.

## 2.5 Experiments and discussions

We ran all experiments on a server with 128 Intel® Xeon® E7-4850 v4 2.10GHz CPU cores. The process of running each system on a specific dataset is limited to a single CPU core.

We test different AutoML systems on midsize OpenML and UCI datasets, using standard machine learning models shown in Appendix A.1, Table A.1. Since data pre-processing is not our focus, we pre-process all datasets in the same way: one-hot encode categorical features and then standardize all features to have zero mean and unit variance. These pre-processed datasets are used in all the experiments.

### 2.5.1 Performance comparison across AutoML systems

We compare AutoML systems that are able to select among different algorithm types under time constraints: Oboe (with error matrix generated from midsize OpenML datasets), auto-sklearn [42], probabilistic matrix factorization (PMF) [46], and a *time-constrained* random baseline. The time-constrained random baseline selects models to observe randomly from those predicted to take less time than the remaining time budget until the time limit is reached.

**Comparison with PMF.** PMF and Oboe differ in the surrogate models they use to explore the model space: PMF incrementally picks models to observe using Bayesian optimization, with model latent meta-features from probabilistic matrix factorization as features, while Oboe models algorithm performance as bilinear in model and dataset meta-features.

PMF does not limit runtime, hence we compare it to Oboe using either QR or ED (number) to decide the set $S$ of models (see Section 2.2.3). Figure 2.4 compares the performance of PMF and Oboe (using QR and ED (number) to decide the set $S$ of models) on our collected error matrix to see which is best able to predict the smallest entry in each row. We show the regret: the difference between the minimal entry in each row and the one found by the AutoML method. In PMF, $N_0 = 5$ models are chosen from the best algorithms on similar datasets (according to dataset meta-features shown in Appendix A.2, Table A.2) are used to warm-start Bayesian optimization, which then searches for the next model to observe. Oboe does not require this initial information before beginning its exploration. However, for a fair comparison, we show both "warm" and "cold" versions. The warm version observes both the models chosen by meta-features and those chosen by QR or ED; the number of observed entries in Figure 2.4 is

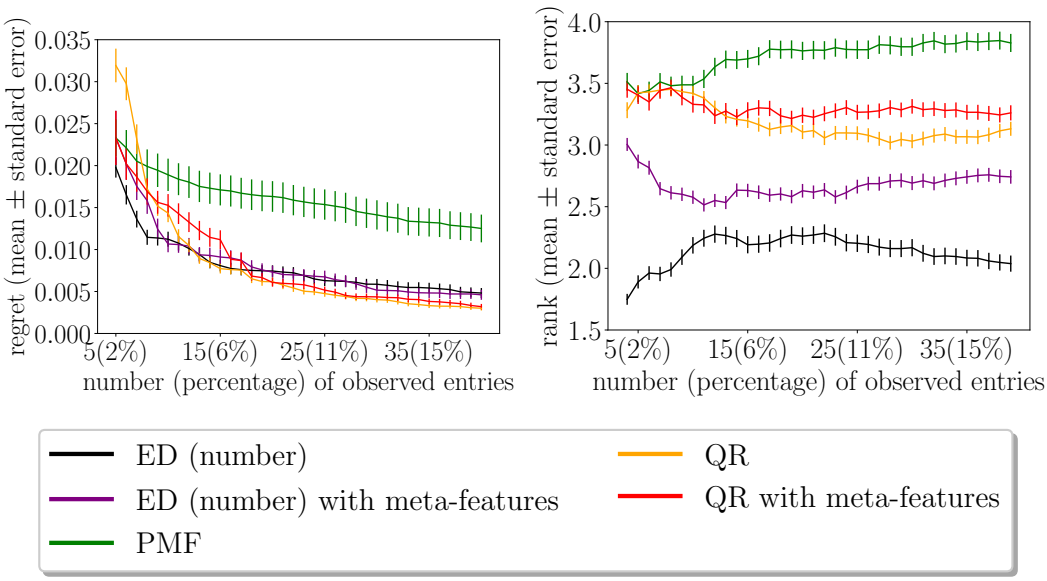



Figure 2.4: Comparison of sampling schemes (QR or ED) in Oboe and PMF. "QR" denotes QR decomposition with column pivoting; "ED (number)" denotes experiment design with number of observed entries constrained. The left plot shows the regret of each AutoML method as a function of number of entries; the right shows the relative rank of each AutoML method in the regret plot (1 is best and 5 is worst).

the sum of all observed models. The cold version starts from scratch and only observes models chosen by QR and ED.

(Standard ED also performs well; see Appendix A.4, Figure A.2.)

Figure 2.4 shows the surprising effectiveness of the low rank model used by Oboe:

1 Meta-features are of marginal value in choosing new models to observe. For QR, using models chosen by meta-features helps when the number of observed entries is small. For ED, there is no benefit to using models chosen by meta-features.

2 The low rank structure used by QR and ED seems to provide a better guide to which models will be informative than the Gaussian process prior used by

PMF: the regret of PMF does not decrease as fast as Oboe using either QR or ED.

**Comparison with auto-sklearn.** The comparison with PMF assumes we can use the labels for every point in the entire dataset for model selection, so we can compare the performance of every model selected and pick the one with lowest error. In contrast, our comparison with auto-sklearn takes place in a more challenging, realistic setting: when doing cross-validation on the meta-test dataset, we do not know the labels of the validation fold until we evaluate performance of the ensemble we built within time constraints on the training fold.

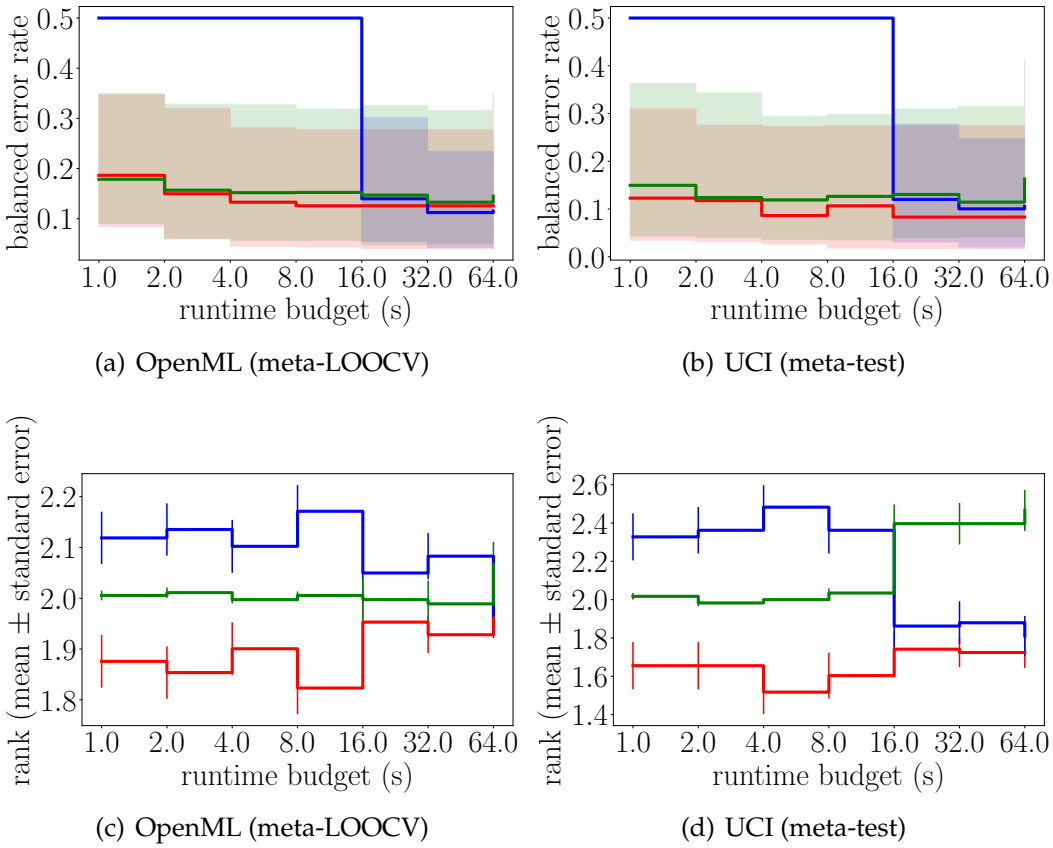
Figure 2.5 shows the error rate and ranking of each AutoML method as the runtime repeatedly doubles. Again, Oboe's simple bilinear model performs surprisingly well[2]:

1 Oboe on average performs as well as or better than auto-sklearn (Figures 2.5(c) and 2.5(d)).

2 The quality of the initial models computed by Oboe and by auto-sklearn are comparable, but Oboe computes its first nontrivial model more than 8× faster than auto-sklearn (Figures 2.5(a) and 2.5(b)). In contrast, auto-sklearn must first compute meta-features for each dataset, which requires substantial computational time, as shown in Appendix A.3, Figure A.1.

3 Interestingly, the rate at which the Oboe models improves with time is also faster than that of auto-sklearn: the improvement Oboe makes before 16 seconds

[2]Auto-sklearn's GitHub Issue #537 says "Do not start auto-sklearn for time limits less than 60s". These plots should not be taken as criticisms of auto-sklearn, but are used to demonstrate Oboe's ability to select a model within a short time.




(a) OpenML (meta-LOOCV) (b) UCI (meta-test)

(c) OpenML (meta-LOOCV) (d) UCI (meta-test)

Figure 2.5: Comparison of AutoML systems in a time-constrained setting, including Oboe with experiment design (red), auto-sklearn (blue), and Oboe with time-constrained random initializations (green). OpenML and UCI denote mid-size OpenML and UCI datasets. "meta-LOOCV" denotes leave-one-out cross-validation across datasets. In 2.5(a) and 2.5(b), solid lines represent medians; shaded areas with corresponding colors represent the regions between 75th and 25th percentiles. Until the first time the system can produce a model, we classify every data point with the most common class label. Figures 2.5(c) and 2.5(d) show system rankings (1 is best and 3 is worst).

matches that of auto-sklearn from 16 to 64 seconds. This indicates that the large time budget may be better spent in fitting more models than optimizing over hyperparameters, to which auto-sklearn devotes the remaining time.

4 Experiment design leads to better results than random selection in almost all cases.

### 2.5.2 Why does Oboe work?

Oboe performs well in comparison with other AutoML methods despite making a rather strong assumption about the structure of model performance across datasets: namely, bilinearity. It also requires effective predictions for model runtime. In this section, we perform additional experiments on components of the Oboe system to elucidate why the method works, whether our assumptions are warranted, and how they depend on detailed modeling choices.

**Low rank under different metrics.** Oboe uses balanced error rate to construct the error matrix, and works on the premise that the error matrix can be approximated by a low rank matrix. However, there is nothing special about the balanced error rate metric: most metrics result in an approximately low rank error matrix. For example, when using the AUC metric to measure error, the 418-by-219 error matrix from midsize OpenML datasets has only 38 eigenvalues greater than 1% of the largest, and 12 greater than 3%.

**Runtime prediction performance.** Runtimes of linear models are among the most difficult to predict, since they depend strongly on the conditioning of the problem. Our runtime prediction accuracy on midsize OpenML datasets is shown in Table 2.1 and in Figure 2.6. We can see that our empirical prediction of model runtime is roughly unbiased. Thus the sum of predicted runtimes on multiple models is a roughly good estimate.

**Cold-start.** Oboe uses D-optimal experiment design to cold-start model selection. In Figure 2.7, we compare this choice with A- and E-optimal design and nonlinear regression in Alors [93], by means of leave-one-out cross-validation on midsize OpenML datasets. We measure performance by the relative RMSE

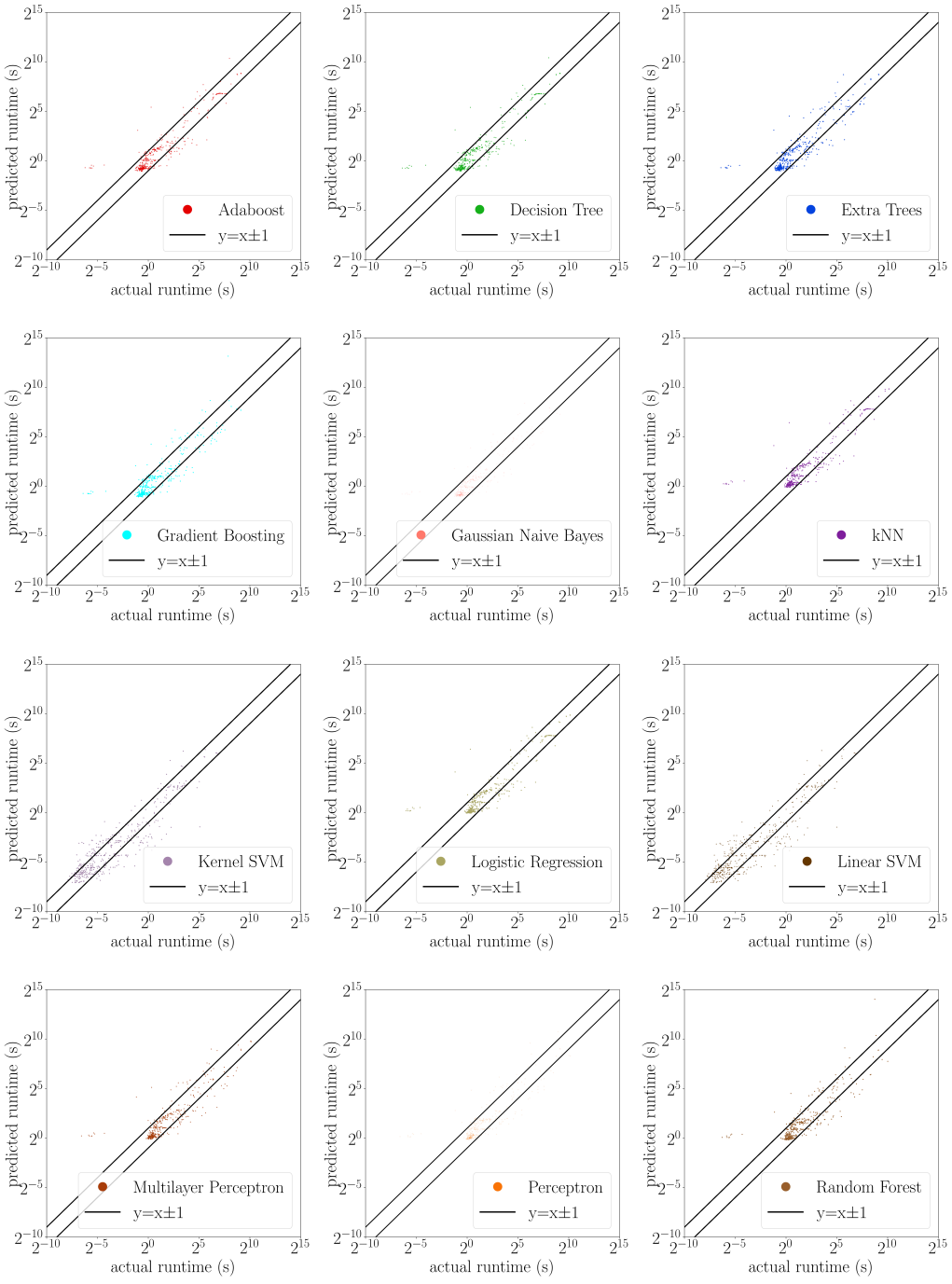



Figure 2.6: Runtime prediction performance on different machine learning algorithms, on midsize OpenML datasets.

Table 2.1: Runtime prediction accuracy on OpenML datasets (Oboe)

| Algorithm type | Runtime prediction accuracy | |
|---|---|---|
| | within factor of 2 | within factor of 4 |
| Adaboost | 83.6% | 94.3% |
| Decision tree | 76.7% | 88.1% |
| Extra trees | 96.6% | 99.5% |
| Gradient boosting | 53.9% | 84.3% |
| Gaussian naive Bayes | 89.6% | 96.7% |
| kNN | 85.2% | 88.2% |
| Logistic regression | 41.1% | 76.0% |
| Multilayer perceptron | 78.9% | 96.0% |
| Perceptron | 75.4% | 94.3% |
| Random Forest | 94.4% | 98.2% |
| Kernel SVM | 59.9% | 86.7% |
| Linear SVM | 30.1% | 73.2% |

$\|e - \hat{e}\|_2 / \|e\|_2$ of the predicted performance vector and by the number of correctly predicted best models, both averaged across datasets. The approximate rank of the error matrix is set to be the number of eigenvalues larger than 1% of the largest, which is 38 here. The time limit in experiment design implementation is set to be 4 seconds; the nonlinear regressor used in Alors implementation is the default `RandomForestRegressor` in scikit-learn 0.19.2 [101].

The horizontal axis is the number of models selected; the vertical axis is the percentage of best-ranked models shared between true and predicted performance vectors. D-optimal design robustly outperforms.

**Ensemble size.** As shown in Figure 2.8, more than 70% of the ensembles constructed on midsize OpenML datasets have no more than 5 base learners. This parsimony makes our ensembles easy to implement and interpret.

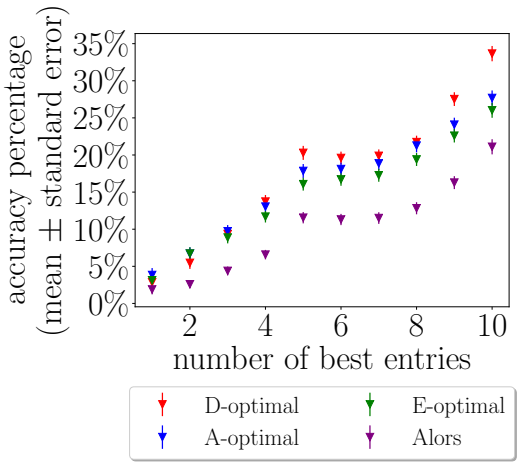



Figure 2.7: comparison of cold-start methods

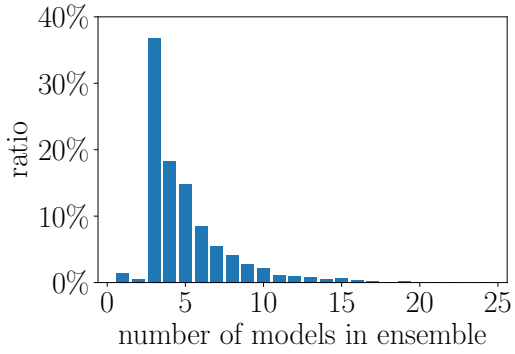



Figure 2.8: Histogram of Oboe ensemble size. The ensembles were built in executions on midsize OpenML datasets in Section 2.5.1.

CHAPTER 3

# TENSOROBOE: COLLABORATIVE FILTERING FOR FAST MACHINE LEARNING PIPELINE SELECTION

This chapter presents TensorOboe [141], an AutoML framework that builds on Oboe in Chapter 2 to select promising machine learning pipelines in the meta-learning setting. Since a pipeline has many components, each with various candidates to choose from, we use tensors with low multilinear ranks to model pipeline performance on datasets. In meta-test, the convexification method in Oboe is too expensive for the task of choosing informative pipelines from a larger number of candidates with constrained experiment design. We use a faster greedy method to directly find a solution to the mix-integer experiment design problem, and demonstrate in experiments that this method works better for the task.

Collecting pipeline performance data in the meta-training stage is often expensive. The hope is to collect good-enough meta-training data by only performing part of the meta-training pipeline evaluations. Since some evaluations may be more informative to the AutoML system, an abstraction of the problem is: how can we complete a partially observed tensor, with entry-wise observation probabilities possibly uneven and depend on the entries being missing? Towards this goal, TenIPS [143] presents a provable tensor completion method that completes a missing-not-at-random tensor with low multilinear rank structure. We may use such an approach to provably sample runs of experimental evaluations in the meta-training of TensorOboe.

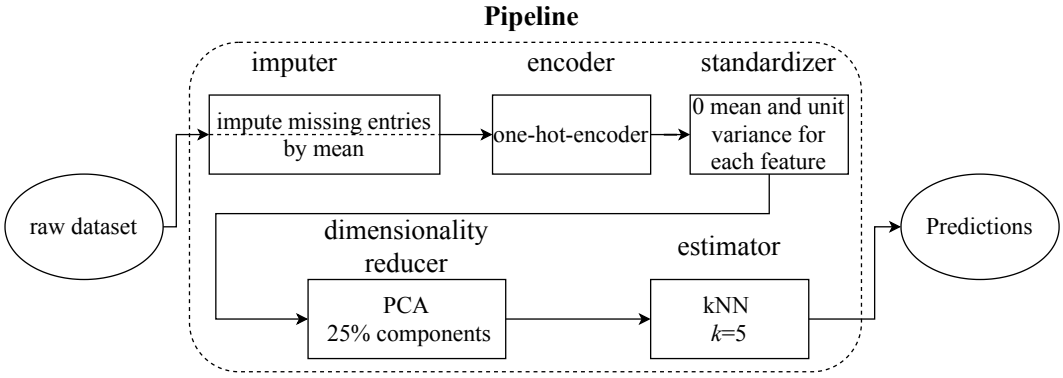



Figure 3.1: An example pipeline.

## 3.1 Introduction

A machine learning pipeline is a directed graph of learning components including imputation, encoding, standardization, dimensionality reduction, and estimation, that together define a function mapping input data to output predictions. Each component may also include hyperparameters, such as the output dimension of PCA, or the number of trees in a random forest. Simple pipelines may consist of sequences of these components; more complex pipelines may combine inputs to form pipelines with more complex topologies. An example pipeline is shown as Figure 3.1.

The job of a data scientist facing a new supervised learning problem is to choose the pipeline that yields a low out-of-sample error from among all possible pipelines. This task is challenging. First, no component dominates all others: there is "no free lunch" [138]. Rather, each performs well on certain data distributions. For example, the PCA dimensionality reducer works well on data points in $\mathbb{R}^d$ that roughly lie in a low rank subspace $\mathbb{R}^k$ with $k < d$; the feature selector that keeps features with large variances works well on datasets if such features are more informative; the Gaussian naive Bayes classifier works

well on features with normally distributed values in each class. However, it is difficult to check these distributional assumptions without running the component on the data: an expensive proposition! The second is the dependence of these choices: for example, standardizing the data may help some estimators, and harm others. Moreover, as the number of possible machine learning components grows, the number of possibilities grows exponentially, defying enumeration. Automating the selection of a pipeline is thus an important problem, which has received attention both from academia and industry [97, 42, 36, 87].

Human experts tackle this difficulty by choosing the right combination according to their domain knowledge. However, finding the right combination takes substantial expertise, and still requires several model fits to find the right combination of components and hyperparameters. An automated pipeline construction system, like a human expert, first forms a *surrogate model* to predict which pipelines are likely to work well. Surrogate models are meta-models that map dataset and machine learning model properties to quantities that characterize performance or informativeness.

A good surrogate model enables efficient search through the pipeline space. "All models are wrong, but some are useful [17]": a good surrogate model makes predictions that guide the search for pipelines without the need for many model fits. Auto-sklearn [42] and Alpine Meadow [111] use meta-learning [126, 2, 78, 129] to choose promising pipelines from those that perform the best on neighboring datasets, and use Bayesian optimization to fine-tune hyperparameters. TPOT [97] uses genetic programming to search over pipeline topologies. Alpine Meadow [111] uses multi-armed bandit to balance the exploration and exploitation of pipeline structures. In this work, we use a low multilinear

rank tensor as our surrogate model. This model makes explicit use of the combinatorial structure of the problem: as a result, the number of pipeline evaluations required to fit the surrogate model on a new dataset is modest, and independent of the number of pipeline components.

Our system learns the surrogate model for a new dataset by fitting a few pipelines on it. The problem of which pipelines to evaluate first, in order to predict the effectiveness of others, is called the *cold-start problem* in the literature on recommender systems. This problem is also of great interest to the AutoML community. Proximity in meta-features, "simple, statistical or landmarking metrics to characterize datasets [141]", are used by many AutoML systems [102, 43, 42, 46, 111] to select models that work well on neighboring datasets, with the belief that models perform similarly on datasets with similar characteristics. Probabilistic matrix factorization has been used to extract dataset latent representations from pipeline performance [46]. Other dataset and pipeline embeddings have also been proposed that use pipeline performance or even textual dataset or algorithm descriptions to build surrogate models [137, 141, 38].

In this work, we build pipeline embeddings by fitting a tensor decomposition to the (incompletely observed) tensor of pipeline performance on a set of training datasets. The tensor model is easy to extend to a new dataset by fitting a constant number of pipelines on it. We describe a simple rule to select which pipelines to observe by solving a constrained version of the classical experiment design [132, 68, 104, 18] problem using a greedy heuristic [90].

We consider the following concrete challenge in this work: select several pipelines that perform the best within a given time limit for a new dataset, in the case that we already know or have time to collect pipeline performance on some

existing datasets. We focus on small data and traditional supervised machine learning pipelines in our experiments, although the methodology can be generalized to a wider range of disciplines. Our main technical contributions are: a new tensor model to exploit the combinatorial pipeline performance structure, and a new pipeline search mechanism that builds on ideas from greedy experiment design. Together, these ideas yield a new state-of-the-art system for AutoML pipeline selection. Since Oboe 2 selects machine learning models by matrix factorization, we name our system in this work TensorOboe: the AutoML system that uses tensor decomposition to select pipelines.

## 3.2 Methodology

### 3.2.1 Overview

TensorOboe has two phases. In the offline phase, we compute the performance of pipelines on meta-training datasets to build a tensor surrogate model. In the online phase, we run a small number of pipelines on the new meta-test dataset to specialize the surrogate model and identify promising pipelines.

**Offline Stage.** We collect a partially observed error tensor using the approach described in Section 3.2.2 to limit the total runtime of the offline phase. We complete and decompose the error tensor $\mathcal{E}$ using the EM-Tucker algorithm, shown as Algorithm 4, with dataset and estimator ranks empirically chosen to be the ones that give low reconstruction error, described in Section 3.4.2.

**Online Stage.** Online, given a new dataset $\mathcal{D}$ with $n^{\mathcal{D}}$ data points and $p^{\mathcal{D}}$ fea-

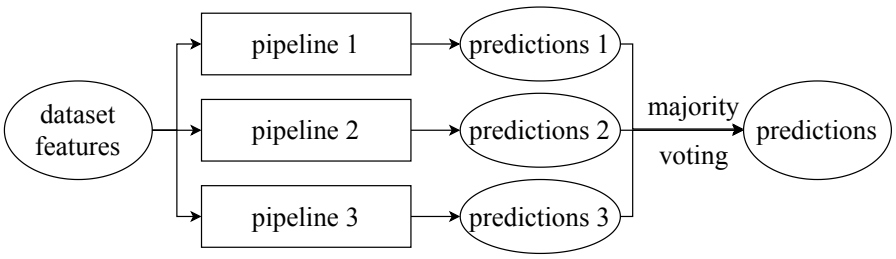



Figure 3.2: A pipeline ensemble with 3 base learners.

tures, we first predict the running time of each pipeline by a simple model: order-3 polynomial regression on $n^{\mathcal{D}}$ and $p^{\mathcal{D}}$ and their logarithms. This simple model works well because the time to fit the estimator dominates the time to fit the pipeline, and the theoretical complexities of estimators we use have no higher order terms [66, 141].

The initial dataset and estimator ranks are set to the number of principal components that capture 97% of the energy in the respective tensor matricizations. We double the runtime budget at each iteration and increment the estimator rank if the performance improves. In each iteration, we build ensembles whose base learners are the 5 pipelines with the best cross-validation error. An ensemble can improve on the performance of the best base pipeline. An example is shown as Figure 3.2.

### 3.2.2 Tensor collection for meta-training

In the meta-training phase of meta-learning, meta-training data is generally assumed to be already available or cheap to collect. Given the large number of possible pipeline combinations, though, collecting meta-training data can be prohibitively expensive. As an example, even if it takes one minute on average to evaluate each pipeline on each dataset, evaluating $20,000$ pipelines on 200

meta-training datasets would take more than 7 years of CPU time. This motivates us to use tensor completion to limit the time spent on the collection of meta-training data, while preserving accuracy of our surrogate model.

We collect pipeline performance in a biased way: using 3-fold cross-validation, we only evaluate pipelines that complete within 120 seconds. This rule gives a missing ratio of 3.3%. Notice that the entries are not missing uniformly at random: for example, some datasets are large and expensive to evaluate; our training data systematically lacks data from these large datasets. Nevertheless, we will show how to infer these entries using tensor completion in Section 3.2.3, and demonstrate in Section 3.4.2 that the method performs well despite bias.

### 3.2.3 Tensor decomposition and rank

The meta-training phase constructs the error tensor $\mathcal{E}$. In the meta-test phase, we see a new dataset, corresponding to a new slice of $\mathcal{E}$. To learn about the slice efficiently, we use a low rank tensor decomposition to predict all the entries in this slice from a subset of its informative entries.

Unlike matrices, there are many incompatible notions of tensor ranks and low rank tensor decompositions, including CANDECOMP/PARAFAC (CP) [22, 54], Tucker [127], and tensor-train [99]. Each emphasizes a different aspect of the tensor low rank property. In this work, we use Tucker decomposition; an illustration on an order-3 tensor is shown as Figure 3.3. As a form of higher-order PCA, Tucker decomposes a tensor into the product of a *core tensor* and several *factor matrices*, one for each mode [75]. A tensor with low multilinear

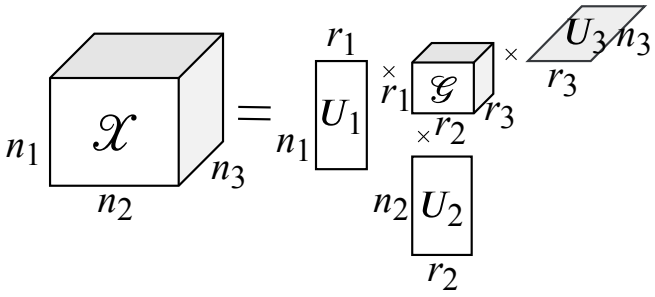



Figure 3.3: Tucker decomposition on an order-3 tensor.

rank has a low rank Tucker decomposition. In our setting of order-6 tensors, Tucker decomposition of $\mathcal{E}$ is

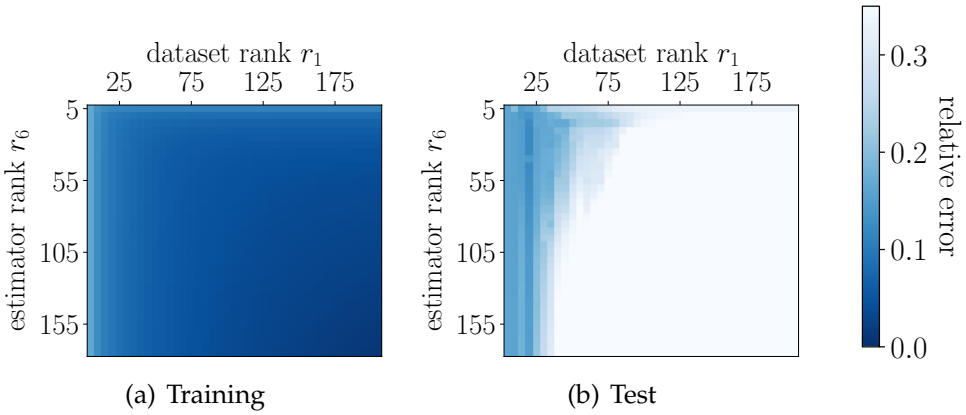
$$\mathcal{E} \approx \hat{\mathcal{E}} = \mathcal{G} \times_1 U_1 \times \cdots \times_6 U_6, \tag{3.1}$$

with core tensor $\mathcal{G} \in \mathbb{R}^{r_1 \times r_2 \times \cdots \times r_6}$ and column-orthonormal factor matrices $U_i \in \mathbb{R}^{n_i \times r_i}$, $i \in \{1, 2, \cdots, 6\}$. $\hat{\mathcal{E}}$ is linear in the factor matrices. Each factor matrix can thus be viewed as embedding the corresponding dataset or pipeline component, with pipeline embeddings as columns of $Y = (\mathcal{G} \times_2 U_2 \times \cdots \times_6 U_6)^{(1)} \in \mathbb{R}^{r_1 \times (\Pi_{i=2}^{6} n_i)}$, the mode-1 matricization of the product. We can use this observation to approximately factor the error matrix $E$, using Equation 3.1, as

$$X^\top Y \approx E \in \mathbb{R}^{n_1 \times (\Pi_{i=2}^{6} n_i)}, \tag{3.2}$$

in which $X \in \mathbb{R}^{r_1 \times n_1}$ and $Y \in \mathbb{R}^{r_1 \times (\Pi_{i=2}^{6} n_i)}$ are dataset and pipeline embeddings, respectively.

Figure 3.4 shows the low rank Tucker decomposition fits the error tensor well.




(a) Training (b) Test

Figure 3.4: Relative error heatmaps when varying ranks in dataset and estimator dimensions. Here, training entries are the ones with runtime less than 90 seconds; the test entries are the ones with runtime between 90 and 120 seconds.

### 3.2.4 Tensor completion

To infer missing entries in the error tensor we collected, namely the entries that take more than the time threshold to evaluate, we use the expectation-maximization (EM) [32, 119] approach together with Tucker decomposition in each step, which we call EM-Tucker and present as Algorithm 4.

**Algorithm 4** EM-Tucker algorithm for tensor completion

**Input:** order-$n$ error tensor $\mathcal{E}$ with missing entries, target multilinear rank $[r_1, \cdots, r_n]$
**Output:** imputed error tensor $\mathcal{E}$

$\mathcal{E}_{\text{obs}} \leftarrow \mathcal{E}$
$\mathbf{\Omega} \leftarrow$ observed entries in $\mathcal{E}_{\text{obs}}$
**do**
  $\mathcal{G}, \{U_i\}_{i=1}^n \leftarrow \text{Tucker}(\mathcal{E}, \text{ranks}=[r_1, \cdots, r_n])$
  $\mathcal{E}_{\text{pred}} \leftarrow \mathcal{G} \times_1 U_1 \times \cdots \times_n U_n$
  $\mathcal{E} \leftarrow \mathbf{\Omega} \odot \mathcal{E}_{\text{obs}} + (1 - \mathbf{\Omega}) \odot \mathcal{E}_{\text{pred}}$
**while** not converged

In Algorithm 4, $\mathbf{\Omega}$ is a binary tensor that denotes whether each entry of the error tensor $\mathcal{E}$ is observed or not. $\mathbf{\Omega}$ has the same shape as the original error tensor, with the corresponding entry $\mathbf{\Omega}_{i_1,i_2,\ldots,i_n} = 1$ if the $(i_1, i_2, \ldots, i_n)$-th entry of

the error tensor is observed, and 0 otherwise. The algorithm is regarded to have converged when the decrease of relative error is less than 0.1%.

Why bother with tensor completion? To recover the missing entries of a tensor, we can also perform matrix completion after matricization or perform matrix completion on every slice separately. Tensors are more constrained and so provide better fits to sparse and noisy data. Consider a tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \cdots I_N}$ with multilinear rank $[r_1, r_2, \cdots, r_n]$, where $I_1 = I_2 = \cdots = I_n = I$ and $r_1 = r_2 = \cdots = r_n = r$. The number of degrees of freedom of $\mathcal{X}$, which is the minimum number of entries required to recover $\mathcal{X}$, is $r^n + n(rI - r^2) =: m_0$. If we unfold $\mathcal{X}$ to $X \in \mathbb{R}^{I \times I^{n-1}}$, the number of degrees of freedom of $X$ is $(I + I^{n-1} - r)r =: m_1$. If we treat every slice of $\mathcal{X}$ separately, the number of degrees of freedom is $I^{n-2}(2rI - r^2) =: m_2$. Therefore, when $r < I$, we have $m_0 < m_1 < m_2$, which means we need fewer parameters to determine $\mathcal{X}$, compared to the matricization and union of slices. Thus, tensor completion may outperform matrix completion on $\mathcal{X}$ with the same number of observed entries.

### 3.2.5 Fast and accurate resource-constrained active learning

Given a new dataset, we first select a subset of pipelines to fit, so that we may estimate the performance of other pipelines. We use ideas from linear experiment design, which picks a subset of low-cost statistical trials to minimize the variance of the resulting estimator, to make this selection.

Concretely, we estimate the embedding $x$ of the new dataset by linear regression. Given the linear model as Equation 3.2, with known performance $e_S$ of a

subset $S \subseteq [n]$ of pipelines on the new dataset, we have

$$e_S = (Y_{:S})^\top x + \epsilon, \tag{3.3}$$

in which $Y$ collects the latent embeddings of pipeline performance, and $\epsilon$ is the error in this linear model. An example of the source of error is the misspecification of target multilinear rank for the Tucker decomposition. We estimate $x$ by linear regression and denote the result as $\hat{x}$. Then we estimate the performance of pipelines in $[n] \backslash S$ by the corresponding entries in $\hat{e} = Y^\top \hat{x}$.

Now we consider which $S$ to choose to accurately estimate $x$. We will motivate the use of the experiment design model and its greedy approach by first showing how to constrain the *number* of pipelines sampled in Section 3.2.5, and then develop a *time*-constrained version that we use in practice in Section 3.2.5.

**Greedy method for size-constrained experiment design.** Suppose the error $\epsilon \sim \mathcal{N}(0, \sigma^2 I)$. Using the linear regression model, Equation 3.3, we want to minimize the expected $\ell_2$ error $E_\epsilon \|\hat{x} - x\|^2 = E_\epsilon \|\hat{x} - E_\epsilon \hat{x}\|^2 + \|E_\epsilon \hat{x} - x\|^2$. Here, the second term is 0 since linear regression is unbiased, and the first term is the covariance $\sigma^2 (YY^\top)^{-1}$ of the estimated embedding $\hat{x}$, which is straightforward to compute.

Imagine we have enough time to run at most $m$ pipelines (and all pipelines run equally slowly). Given pipeline embeddings $\{y_j\}_{j=1}^n$ (which we call *design vectors* or *designs*), in which each $y_j \in \mathbb{R}^k$, we minimize a scalarization of the covariance to obtain the (number-constrained) $D$-optimal experiment design problem

$$\begin{array}{ll} \text{maximize} & \log\det\left(\sum_{j \in S} y_j y_j^\top\right) \\ \text{subject to} & |S| \leq m \\ & S \subseteq [n]. \end{array} \tag{3.4}$$

Here, $\sum_{j \in S} y_j y_j^\top$, the inverse of (scaled) covariance matrix, is called the Fisher information matrix.

Obtaining an exact solution for a mixed-integer nonlinear combinatorial optimization problem like Problem 3.4 is prohibitively expensive. Convexification is commonly used to solve such a problem [18, 104, 141]. However, we have more than $20,000$ pipelines to select from, making convex relaxations also too slow. Moreover, we can find better solutions with the greedy heuristic we present next.

Greedy methods form another popular approach to combinatorial optimization problems like Problem 3.4. Importantly, the objective function of Problem 3.4, $f(S) = \log\det\left(\sum_{j \in S} y_j y_j^\top\right)$, is submodular. (Recall a set function $g : 2^V \to \mathbb{R}$ defined on a subset of $V$ is submodular if for every $A \subseteq B \subseteq V$ and every element $s \in V \backslash B$, we have $g(A \cup \{s\}) - g(A) \geq g(B \cup \{s\}) - g(B)$. This characterizes a "diminishing return" property.) Given a size constraint, the submodular function maximization problem

$$\begin{aligned} \text{maximize} \quad & g(S) \\ \text{subject to} \quad & S \subseteq V \\ & |S| \leq m \end{aligned} \tag{3.5}$$

can be solved with a $1 - \frac{1}{e}$ approximation ratio [96] by the greedy approach: in every step, add the single design vector that maximizes the increase in function value. In $D$-optimal experiment design, we can compute this increase efficiently using Lemma 1.

**Lemma 1 (Matrix Determinant Lemma [55, 90])** *For any invertible matrix $A \in \mathbb{R}^{k \times k}$ and $a, b \in \mathbb{R}^k$,*

$$\det(A + ab^\top) = \det(A)(1 + b^\top A^{-1} a).$$

At the $t$-th step in our setting, with an already constructed Fisher information matrix $X_t = \sum_{j \in S} y_j y_j^\top$, we have

$$\operatorname{argmax}_{j \in [n] \setminus S} \det(X_t + y_j y_j^\top) = \operatorname{argmax}_{j \in [n] \setminus S} y_j^\top X_t^{-1} y_j.$$

Here, $y_j^\top X_t^{-1} y_j$ can be seen as the payoff for adding pipeline $j$. From the $t$-th to the $(t+1)$-th step, with the selected design vector at the $t$-th step as $y_t$, we update $X_t$ to $X_{t+1} = X_t + y_t y_t^\top$ by Lemma 2:

**Lemma 2 (Sherman-Morrison formula [113, 53])** *For any invertible matrix $A \in \mathbb{R}^{k \times k}$ and $a, b \in \mathbb{R}^k$,*

$$(A + ab^\top)^{-1} = A^{-1} - \frac{A^{-1} a b^\top A^{-1}}{1 + b^\top A^{-1} a}.$$

Pseudocode for the greedy algorithm to solve Problem 3.4 is shown as Algorithm 5, with per-iteration time complexity $O(k^3 + nk^2)$: it takes $O(k^3)$ (for a naive matrix multiplication algorithm) to update $X_t^{-1}$ and $O(nk^2)$ to choose the best pipeline to add.

**Algorithm 5** Greedy algorithm for size-constrained $D$-design

**Input:** design vectors $\{y_j\}_{j=1}^n$, in which $y_j \in \mathbb{R}^k$; maximum number of selected pipelines $m$; initial set of designs $S_0 \subseteq [n]$, s.t. $X_0 = \sum_{j \in S_0} y_j y_j^\top$ is non-singular
**Output:** The selected set of designs $S \subseteq [n]$

```
function GREEDY_ED_NUMBER
    S ← S_0
    do
        i ← argmax_{j∈[n]\S} y_j^⊤ X_t^{-1} y_j
        S ← S ∪ {i}
        X_{t+1} ← X_t + y_i y_i^⊤
    while |S| ≤ m
    return S
end function
```

There remains the problem of how to select an initial set of designs $S$ to start from, such that $X_0 = \sum_{j \in S} y_j y_j^\top = Y_{:S} Y_{:S}^\top$ is non-singular. This is equivalent to

the problem of finding a subset of vectors in $\{y_j\}_{j=1}^n$ that can span $\mathbb{R}^k$. We select this sized-$k$ subset $S_0$ to be the first $k$ pivot columns from QR factorization with column pivoting [49, 51] on $Y$, with time complexity $O((n+k)k^2)$.

**Greedy method for time-constrained experiment design.** We here move on to the realistic case in AutoML pipeline selection: which pipelines should we select to gain an accurate estimate of the entire pipeline space? In this setting, each pipeline is associated with a different cost. We characterize the cost as running time, and form the time-constrained version of experiment design as

$$\begin{aligned} \text{maximize} \quad & \log\det\left(\textstyle\sum_{j\in S} y_j y_j^\top\right) \\ \text{subject to} \quad & \textstyle\sum_{j\in S} \hat{t}_j \le \tau \\ & S \subseteq [n], \end{aligned} \tag{3.6}$$

in which $\{\hat{t}_i\}_{i=1}^n$ are the estimated pipeline running times. The payoff of adding design $i$ in the $t$-th step can thus be formulated as $\frac{y_i^\top X_t^{-1} y_i}{\hat{t}_i}$, giving Algorithm 6 the greedy method to solve Problem 3.6.

**Algorithm 6** Greedy algorithm for time-constrained $D$-design

**Input:** design vectors $\{y_j\}_{j=1}^n$, in which $y_j \in \mathbb{R}^k$; estimated running time of pipelines $\{\hat{t}_i\}_{i=1}^n$; maximum running time $\tau$; initial set of designs $S_0 \subseteq [n]$, s.t. $X_0 = \sum_{j\in S_0} y_j y_j^\top$ is non-singular

**Output:** The selected set of designs $S \subseteq [n]$

```
function GREEDY_ED_TIME
    S ← S_0
    do
        i ← argmax_{j∈[n]\S} (y_j^⊤ X_t^{-1} y_j) / t̂_j
        S ← S ∪ {i}
        X_{t+1} ← X_t + y_i y_i^⊤
    while Σ_{i∈S} t̂_i ≤ τ
    return S
end function
```

The initialization problem is solved similarly by the QR method. Given runtime limit $\tau$, we select among columns with corresponding pipelines predicted

to finish within $\frac{\tau}{2k}$. Pseudocode for this initialization algorithm is shown as Algorithm 7.

**Algorithm 7** Initialization of the greedy algorithm for time-constrained $D$-design, by QR factorization with column pivoting

**Input:** design vectors $\{y_j\}_{j=1}^n$, in which $y_j \in \mathbb{R}^k$; (predicted) running time of all pipelines $\{\hat{t}_i\}_{i=1}^n$; maximum running time $\tau$
**Output:** A subset of designs $S_0 \subseteq [n]$ for Algorithm 6 initialization

**function** QR_INITIALIZATION
  $S_{\text{valid}} \leftarrow \{i \in [n] : \hat{t}_i \leq \frac{\tau}{2k}\}$
  $S_0 \leftarrow \emptyset$, $\hat{t}_{\text{sum}} \leftarrow 0$
  **if** $|S_{\text{valid}}| < k$ **then** ▷ Case 1
    **do**
      $i \leftarrow \text{argmin}_{j \in [n] \setminus S} \hat{t}_j$
      $S_0 \leftarrow S_0 \cup \{i\}$
      $\hat{t}_{\text{sum}} \leftarrow \hat{t}_{\text{sum}} + \hat{t}_i$
    **while** $\hat{t}_{\text{sum}} \leq \tau$
  **else** ▷ Case 2
    $S_0 \leftarrow \text{QR\_with\_column\_pivoting}(Y_{:S_{\text{valid}}})[:k]$
  **end if**
  **return** $S_0$
**end function**

A corner case of Algorithm 7, shown as Case 1, is that there are not enough pipelines predicted to be able to finish within time limit. This corresponds to the case that the runtime limit is relatively small compared to the time of fitting pipelines on current dataset. In this case we greedily select the fast pipelines and do not run Algorithm 6 afterwards.

As a side note, the assumption that performance of different pipelines are predicted with equal variance is not quite realistic, especially when some components have much more pipelines than others. If the variance is known (but unequal), we obtain a weighted least squares problem. In the error matrix $E$, we can estimate the variance of prediction error of each pipeline $j \in [n]$ by the sample variance of $e_j - X^\top y_j$ and select the promising pipelines with the goal of

minimizing the rescaled covariance. Practically, however, this rescaled method does not systematically improve on the standard least squares approach in our experiments (shown in Appendix B.2), so we retrench to the simpler approach.

## 3.3 Python implementation

The TensorOboe system follows Oboe, thus we have the implementation in the same GitHub repository at `https://github.com/udellgroup/oboe`. The basic API for class instantiation, meta-training and meta-test is the same as in Oboe (introduced in Chapter 2, Section 2.4). We use the `sklearn.pipeline.Pipeline` class to define pipelines.

## 3.4 Experiments and discussions

We use a Linux machine with 128 Intel® Xeon® E7-4850 v4 2.10GHz CPU cores and 1056GB memory. Offline, we collect cross-validated pipeline performance on meta-training datasets: 215 OpenML [130, 44] classification datasets with number of data points between 150 and 10,000, listed in Appendix B.1.1. The 215 datasets are chosen alphabetically. Pipelines are combinations of the machine learning components shown in Appendix B.1.3, Table B.1, which lists 4 data imputers, 2 encoders, 2 standardizers, 8 dimensionality reducers and 183 estimators, resulting in 23,424 linear pipeline candidates in total.

### 3.4.1 Comparison with time-Constrained AutoML pipeline build systems

In this section, we demonstrate the performance of TensorOboe as an AutoML system for pipeline selection.

A naive approach for pipeline selection is to choose the one that on average performs the best among all meta-training datasets, which we call the baseline pipeline. Given the pipeline selection problem, it is common for human practitioners to try out the best pipeline at the very beginning. On our meta-training datasets, the baseline pipeline is: impute missing entries with the mode, encode categorical features as integers, standardize each feature, remove features with 0 variance, and classify by gradient boosting with learning rate 0.25 and maximum depth 3. The baseline pipeline has an average ranking of 1568 among all 23,424 pipelines across all 215 meta-training datasets.

Human practitioners may also reduce the number of trials by choosing certain pipeline components to be the type that performs the best on average. Figure 3.5, however, shows that although some estimator types (gradient boosting and multilayer perceptron) are commonly seen among the best pipelines, no estimator type uniformly dominates the rest.

We compare TensorOboe with auto-sklearn [42], TPOT [97], and the baseline pipeline in Figure 3.6. To ensure fair comparisons, we use a single CPU core for each AutoML system. We allow each to choose from the same primitives. We can see that:

1 All AutoML frameworks are able to construct pipelines that outperform the

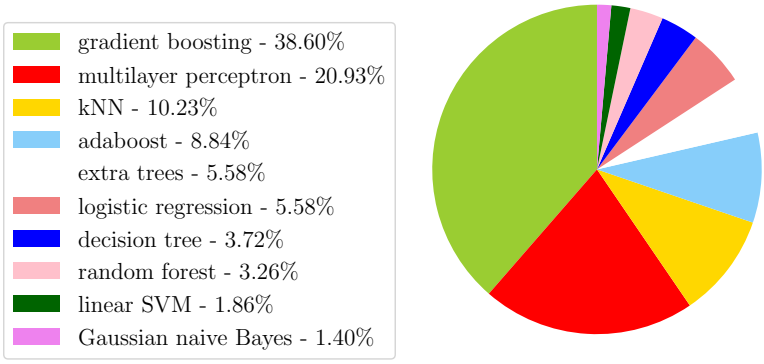
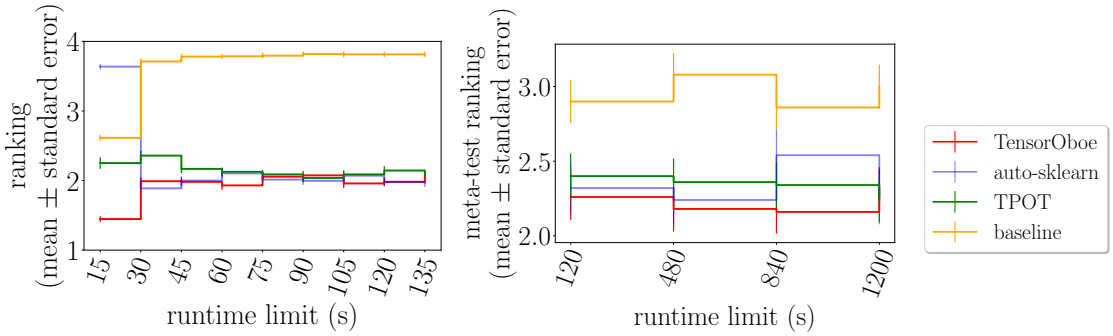



Figure 3.5: Which estimators work best? Distribution of estimator types in best pipelines on meta-training datasets.




(a) OpenML (meta-LOOCV) (b) UCI (meta-test)

Figure 3.6: Rankings of AutoML systems for pipeline search in a time-constrained setting, vs the baseline pipeline. We meta-train on OpenML classification datasets and meta-test on UCI classification datasets [39]. Until the first time the systems can produce a pipeline, we classify every data point with the most common class label. Lower ranks are better.

baseline on average once the method returns a pipeline (for auto-sklearn, this takes 30 seconds).

2 TensorOboe on average outperforms the competing methods and produces meaningful pipeline configurations fastest.

3 With the longer running time in Figure 3.6(b), TensorOboe still outperforms in most cases.

These results show that TensorOboe is able to accurately approximate the hyper-

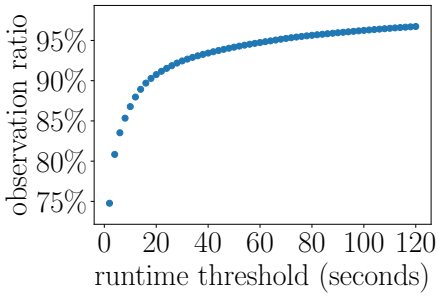



Figure 3.7: CDF of pipeline runtime on meta-training datasets.

parameter landscape. We discuss these results in greater detail in Section 3.4.5.

### 3.4.2 Tensor completion vs matrix completion for error tensor completion

Given meta-training data $\{\mathcal{D}, \mathcal{P}, \mathcal{P}(\mathcal{D})\}$ on a subset of dataset-pipeline combinations, a good surrogate model should accurately predict the performance of new dataset-pipeline combinations.

Figure 3.7 shows that most pipelines run quickly on most datasets: for example, over 90% finish in less than 20 seconds and over 95% finish in less than 80 seconds.

Figure 3.8 compares relative errors of predictions by tensor and matrix surrogate models. For each runtime threshold, we treat pipeline-dataset combinations with running time less than the threshold as training data, and those that take longer than threshold and less than 120 seconds as test. We compute relative errors on test data, hence the name "runtime generalization". To ensure a fair comparison, we set the dataset and estimator ranks to be equal in the tensor model, which is required for the matrix model, since column rank equals row

rank for a matrix. We can see that:

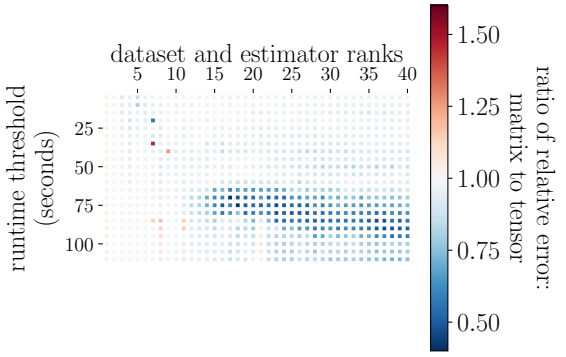
1. The tensor model outperforms the matrix model in nearly all cases, demonstrating that the additional combinatorial structure provided by the tensor model helps recover the combinatorial relationships among different pipeline components.

2. Figure 3.8(b) shows the U-shaped error curve as we increase the dataset and estimator ranks for both matrix and tensor models, moving from underfitting (decreasing error) to overfitting (increasing error). Informed by these results, we select both ranks to be 20, the rank in the middle, in the tensor surrogate model.

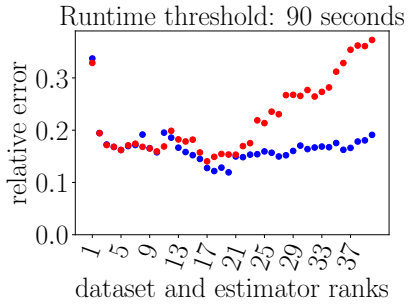
### 3.4.3 Cold-start performance by greedy experiment design

We compare the performance of different approaches to solve the experiment design problem, so as to choose which pipelines we should sample. Recall that there are two approaches:

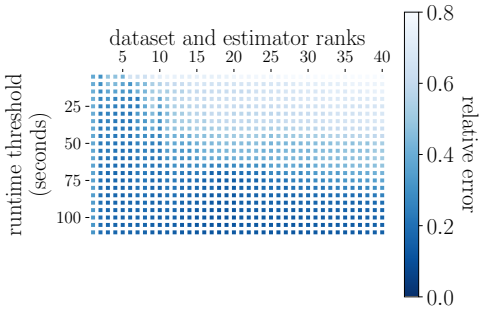
- **Convexification**: Solve the relaxed problem (Equation 3.6 with $v_i \in [0, 1]$, $\forall i \in [n]$) with an SLSQP solver, sort the entries in the optimal solution $v^*$, and greedily add the pipeline with large $v_i^*$ until the runtime limit is reached.
- **Greedy**: Solve the original integer programming problem (Equation 3.6) by the greedy algorithm (Algorithm 6), initialized by time-constrained QR (Algorithm 7).



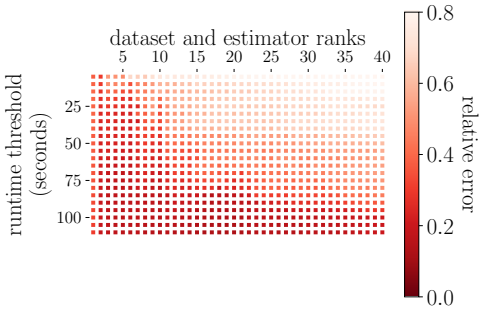

(a) Runtime generalization for tensor vs matrix models. Blue means the tensor model achieves lower error; red means the matrix model does. The tensor model outperforms for longer running times.




(b) Tensor and matrix completion errors when runtime threshold = 90 seconds (3.3% of entries in error tensor missing).




(c) Runtime generalization error by tensor model. Darker colors mean smaller errors.




(d) Runtime generalization error by matrix model. Darker colors mean smaller errors.


Figure 3.8: Tensor completion vs matrix completion for inferring pipeline performance.

For our problem, the greedy approach is superior, since the convexification method is prohibitive on our large $215 \times 23424$ error matrix. Hence we compare these methods on a subset of pipelines that only differ by estimators, 183 in total. This setting matches an experiment in [141]. Shown in Figure 3.9, we can see that:

1 The greedy method performs better for cold-start than convexification (Fig-

ure 3.9(a)): it selects informative designs that better predict the high-performing pipelines (Figure 3.9(b)).

2 The greedy method is more than 30× faster than convexification, which allows TensorOboe to devote its runtime budget to fitting pipelines instead of searching for the informative pipelines.

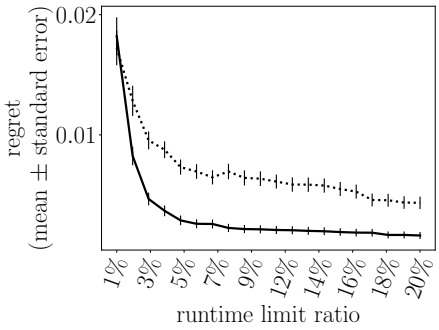
3 Shown in Figure 3.9(d), the greedy algorithm still takes a fair amount of time if the number of designs we select is large; however, the dataset ranks we choose are less than 50, so it generally takes less than 10 seconds to choose informative pipelines. This time can be further reduced using Lemma 2.

### 3.4.4 Pipeline runtime prediction performance

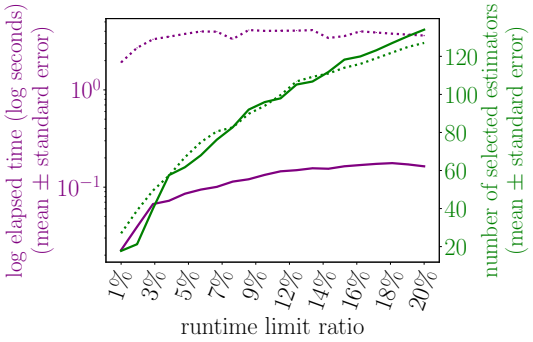
Runtime prediction accuracy is critical for the performance of our time-constrained pipeline selection system. Recall that our predictions use order-3 polynomial regression on $n^{\mathcal{D}}$ and $p^{\mathcal{D}}$, the numbers of data points and features in $\mathcal{D}$, and their logarithms. We shown in Table 3.1 that this runtime predictor performs well.

### 3.4.5 Learning the hyperparameter landscapes

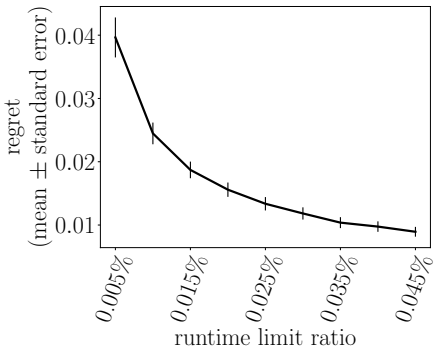
Hyperparameter landscapes plot pipeline performance with respect to hyperparameter values. While parameter landscapes have been extensively studied, especially in the deep learning context (for example, [72, 82, 47]), hyperparameter landscapes are less studied. The previous sections focus on how we can choose among different pipeline component types. In this section, we show



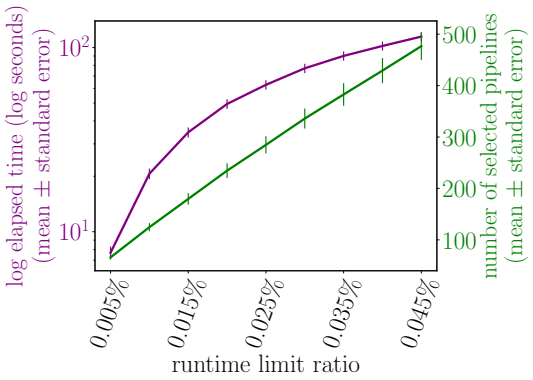

(a) Regret on the small error matrix (215×183) for estimator search.




(b) ED runtime and selected estimators on small error matrix (215×183) for estimator search.




(c) Regret on the full error matrix (215×23424) for pipeline search (greedy method only).




(d) ED runtime and selected pipelines on full error matrix (215×23424) for pipeline search (greedy method only).


Figure 3.9: Comparison of time-constrained experiment design methods across meta-training datasets. The y-axes in 3.9(a) and 3.9(c) are regrets: the difference between minimum pipeline error found by each method and the true minimum. The x-axes are runtime limit ratios: ratios of the runtime limit to the total runtime of all pipelines on each dataset.

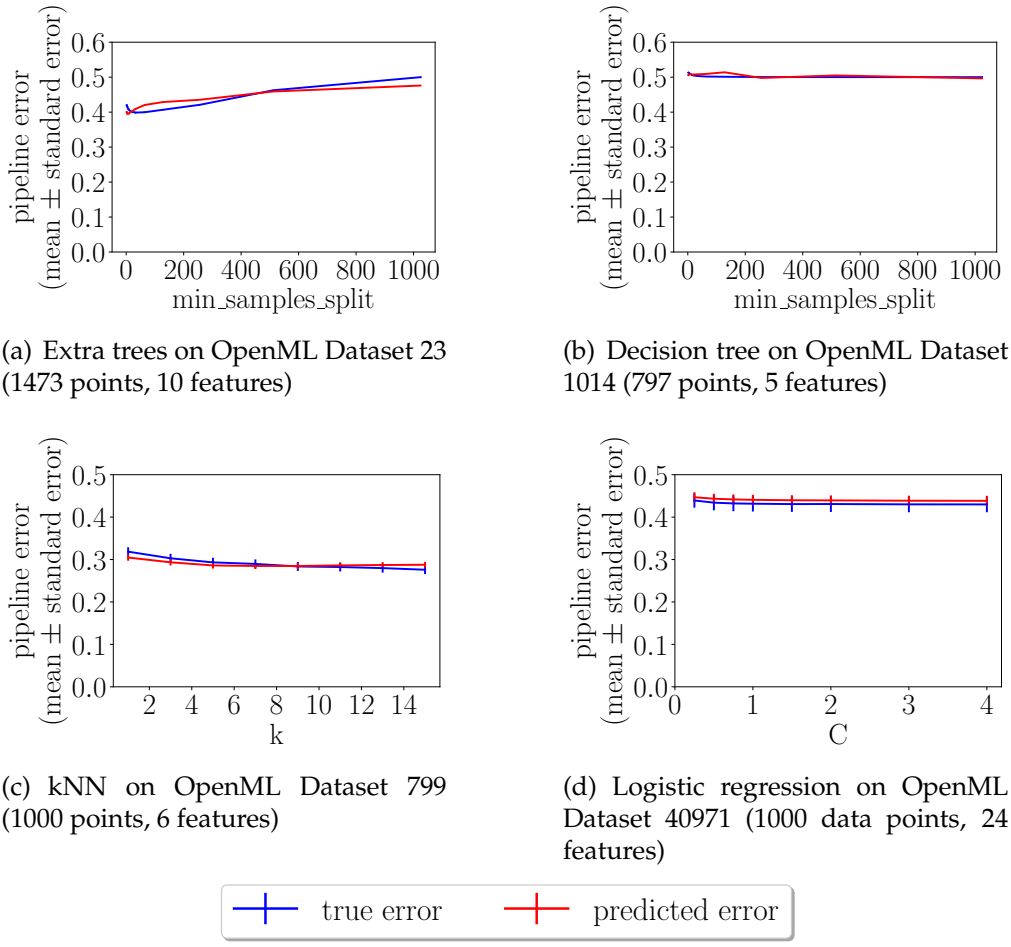
Table 3.1: Runtime prediction accuracy on OpenML datasets (TensorOboe)

| Pipeline estimator type | Runtime prediction accuracy | |
|---|---|---|
| | within factor of 2 | within factor of 4 |
| Adaboost | 73.6% | 86.9% |
| Decision tree | 62.7% | 78.9% |
| Extra trees | 71.0% | 83.8% |
| Gradient boosting | 53.4% | 77.5% |
| Gaussian naive Bayes | 67.3% | 82.3% |
| kNN | 68.7% | 84.4% |
| Logistic regression | 53.6% | 76.1% |
| Multilayer perceptron | 74.5% | 88.9% |
| Perceptron | 64.5% | 82.2% |
| Random Forest | 69.5% | 84.9% |
| Linear SVM | 56.8% | 79.5% |

that our tensor surrogate model is able to learn hyperparameter landscapes of different estimator types that exhibit qualitatively different behaviors.

Figure 3.10 shows some examples of both real and predicted hyperparameter landscapes after running our system for 135 seconds. We can see that our predictions match the overall tendencies of the curves. Larger plots (Figure B.3 in Appendix B.3) show our predictions also capture most of the small variations in these landscapes.

Note TensorOboe does not use a subroutine for hyperparameter optimization: it chooses the hyperparameter for each estimator from a predefined grid of values instead of optimizing hyperparameters by, for example, Bayesian optimization. The hyperparameter landscapes visualized here give confidence that grid search effectively samples performant hyperparameter settings within the range of hyperparameters: a coarse grid suffices.




(a) Extra trees on OpenML Dataset 23 (1473 points, 10 features)

(b) Decision tree on OpenML Dataset 1014 (797 points, 5 features)

(c) kNN on OpenML Dataset 799 (1000 points, 6 features)

(d) Logistic regression on OpenML Dataset 40971 (1000 data points, 24 features)

Figure 3.10: Hyperparameter landscape prediction examples.

## 3.5 Overfitting analysis

Two types of overfitting are of concern in AutoML systems: traditional overfitting (overfitting of models on training folds) and meta-overfitting (overfitting of AutoML surrogate models).

Traditional overfitting may happen in any machine learning system, and is often mitigated by controlling model complexity, cross validation on training set, etc. In TensorOboe, we always evaluate pipelines by k-fold cross validation, and build an ensemble since the pipeline with lowest cross-validation error may

not be the one with lowest test error.

Meta-overfitting happens when meta-training datasets are biased in some sense, and when the surrogate model is so complex that it captures noise in addition to model performance. We mitigate meta-overfitting in the following ways: The OpenML meta-training datasets we collect have diverse topics ranging among multiple science and sociology disciplines. The surrogate model we use is low rank tensor decomposition, a model with low complexity. It denoises cross-validated pipeline error, as discussed in Chapter 2, Section 2.2.

Meta-overfitting still presents many perils. The surrogate model may lack training instances. For example, the perceptron algorithm never performs the best on any meta-training dataset, as shown in Figure 3.5. Hence TensorOboe is unlikely ever to choose a perceptron pipeline. To mitigate this problem, we must collect pipeline performance in a larger space, or consider if the perceptron algorithm (for example) is truly dominated. Another possible source of meta-overfitting is that our meta-training datasets have no more than 10,000 points and smaller number of features. Order-3 polynomial runtime predictors may not generalize well to larger problems.

CHAPTER 4

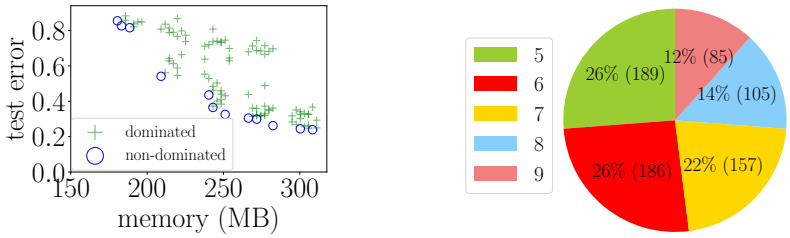
# PEPPP: COLLABORATIVE FILTERING TO SELECT NETWORK PRECISION

In this chapter, we present PEPPP [145], a system that uses meta learning to automatically select a promising low-precision configuration for a new dataset in the memory-constrained setting.

## 4.1 Introduction

Training modern-day neural networks is becoming increasingly expensive as task and model sizes increase. The energy consumption of the corresponding computation has increased alarmingly, and raises doubts about the sustainability of modern training practices [109]. Low-precision training can reduce consumption of both computation [30] and memory [117], thus minimizing the cost and energy to train larger models and making deep learning more accessible to resource-limited users.

Low-precision training replaces 32-bit or 64-bit floating point numbers with fixed or floating point numbers that allocate fewer bits for the activations, optimizers, and weights. A rich variety of methods appear in recent literature, including bit-centering [30], loss scaling [92] and mixed-precision training [154]. There is however a fundamental tradeoff: lowering the number of bits of precision increases the *quantization error*, which may disrupt convergence and increase downstream error [150, 28, 52]. The *low-precision configuration* is thus a hyperparameter to be chosen according to the resource constraints of the practitioner.




(a) error vs memory on CIFAR-10 [77]

(b) # activation bits in non-dominated configurations

Figure 4.1: Test error vs memory for ResNet-18 across 99 low-precision floating point configurations. Figure (a) shows the tradeoff on CIFAR-10. (Non-dominated points are blue circles.) Figure (b) shows that the best precision to use varies depending on the memory budget, on 87 image datasets. See Section 4.4 for experimental details.

How should we choose this hyperparameter? Many believe the highest allowable precision (given a memory budget) generally produces the lowest error model. However, there are typically many ways to use the same amount of memory. As shown in Figure 4.1(a), some of these configurations produce much lower error than others! Figure 4.1(b) shows that no one configuration dominates all others. We might consult previous literature to choose a precision; but this approach fails for new applications.

**Our goal is to efficiently pick the best low-precision configuration under a memory budget.** Efficiency is especially important for resource-constrained practitioners, such as individual users or early-stage startups. To promote efficiency, we use a meta-learning [81, 129, 61] approach: we train a small number of cheap very-low-precision models on the dataset to choose the perfect precision. The gains from choosing the right low-precision format can offset the cost of this extra training — but each precision must be chosen carefully to realize the benefits of low-precision training.

We use ideas from multi-objective optimization to characterize the trade-

off between memory and error and identify the *Pareto frontier*: the set of non-dominated solutions. Users will want to understand the tradeoff between error and memory so they can determine the resources needed to adequately train a model for a given machine learning task. This tradeoff may also influence the design of application-specific low-precision hardware, with profound implications for the future [60]. For example, among all 99 low-precision configurations we tried, we identified some configurations that are Pareto optimal across many different tasks (listed in Appendix C.4). These results could help hardware manufacturers decide which precision settings are worth manufacturing.

Computing the Pareto frontier by training models for all low-precision configurations is expensive and unnecessary. Cloud computing platforms like Google Cloud and Amazon EC2 charge more for machines with the same CPU and double memory: 25% more on Google Cloud[1] and 100% more on Amazon EC2[2] as of mid-September 2021. We use techniques from meta-learning to leverage the information from other low-precision training runs on related datasets. This approach allows us to *estimate* the Pareto frontier without evaluating all of the low-precision configurations.

Our system, Pareto Estimation to Pick the Perfect Precision (PEPPP), has two goals. The first goal is to find the Pareto frontiers of a collection of related tasks, and is called *meta-training* in the meta-learning literature. Meta-training requires a set of *measurements*, each collected by training and testing a neural network with a given precision on a dataset. This information can be gathered offline with a relatively large resource budget, or by crowdsourcing amongst the academic or open-source community. Still, it is absurdly expensive to exhaustively

[1]https://cloud.google.com/ai-platform/training/pricing
[2]https://aws.amazon.com/ec2/pricing/on-demand

evaluate all measurements: that is, every possible low-precision configuration on every task. Instead, we study how to choose a subset of the possible measurements to achieve the best estimate. The second goal we call *meta-test*: using the information learned on previous tasks, how can we transfer that information to a new task to efficiently estimate its Pareto frontier? This goal corresponds to a resource-constrained individual or startup who wants to determine the best low-precision configuration for a new dataset.

Both meta-training and meta-test rely on matrix completion and active learning techniques to avoid exhaustive search: we make a subset of all possible measurements and predict the rest. We then estimate the Pareto frontier to help the user make an informed choice of the best configuration. We consider two sampling schemes: uniform and non-uniform sampling. Uniform sampling is straightforward. Non-uniform sampling estimates the Pareto frontier more efficiently by making fewer or even no high-memory measurements.

To the best of our knowledge, PEPPP is the first to study the error-memory tradeoff in low-precision training and inference without exhaustive search. Some previous works show the benefit of low-precision arithmetic in significant energy reduction at the cost of a small accuracy decrease [56], and propose hardware-software codesign frameworks to select the desired model [79]. Many other papers focus on low-precision inference only and use it for model compression [139, 21]. Another line of work seeks state-of-the-art (SOTA) performance with low-precision training, using carefully designed learning schedules [122, 123]. Our work imagines low-precision as a way to **reduce training or deployment costs of customized models** trained on proprietary datasets (not SOTA models on ML benchmarks). Even so, we show in Section 4.4 that our

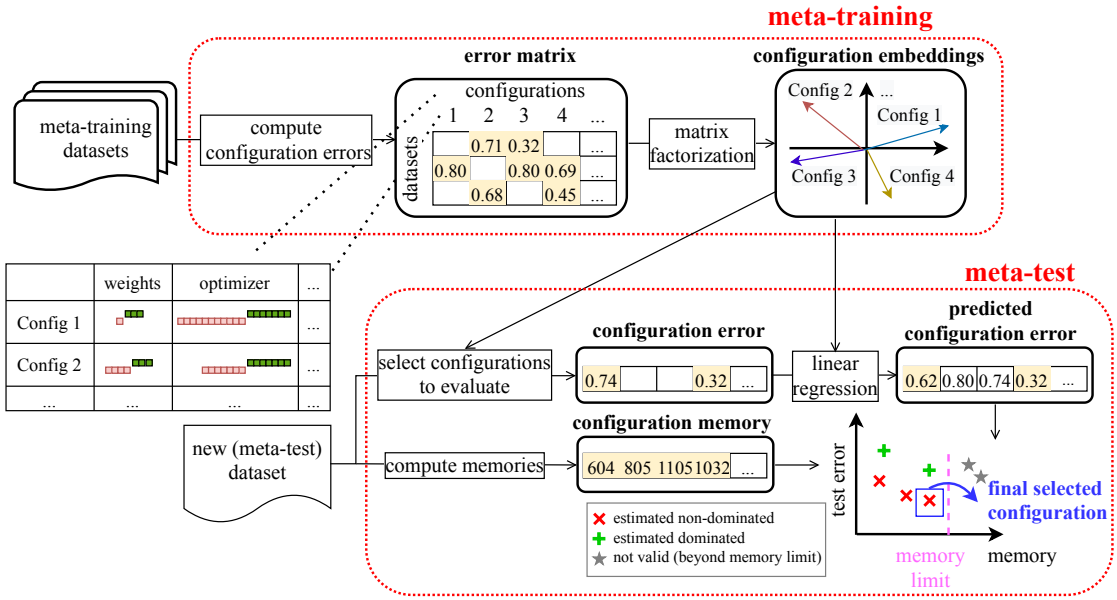



Figure 4.2: The PEPPP workflow. We begin with a collection of (meta-) training datasets and low precision configurations. In the meta-training phase, we sample dataset-configuration pairs to train, and compute the misclassification error. We use matrix factorization to compute a low dimensional embedding of every configuration. In the meta-test phase, our goal is to pick the perfect precision (within our memory budget) for the meta-test dataset. We compute the memory required for each configuration, and we select a subset of fast, informative configurations to evaluate. By regressing the errors of these configurations on the configuration embeddings, we find an embedding for the meta-test dataset, which we use to predict the error of every other configuration (including more expensive ones) and select the best subject to our memory budget.

work picks promising models for every memory budget, which enables near-SOTA results on CIFAR-10.

Figure 4.2 shows a flowchart of PEPPP. The rest of this work is organized as follows. Section 4.2 describes the main ideas we use to actively sample configurations and approximate Pareto frontiers. Section 4.4 shows experimental results.

## 4.2 Methodology

PEPPP operates in two phases: meta-training and meta-test. First in meta-training, we learn configuration embeddings by training a neural network of a specific architecture at different low-precision configurations on different datasets. As listed in Appendix C.1, we study image classification tasks and a range of low-precision formats that vary in the number of bits for the exponent and mantissa for the activations, optimizer, and weights. Our hope is to avoid enumerating every possible configuration since exhaustive search is prohibitive in practice. To this end, we make a few measurements and then predict the other test errors by active learning techniques. We also compute the memory matrix to understand the memory consumption of each measurement.

Then in meta-test, we assume that we have completed meta-training, and hence know the configuration embeddings. The meta-test goal is to predict the tradeoff between memory usage and test error on a new (meta-test) dataset. This step corresponds to inference in traditional machine learning; it must be quick and cheap to satisfy the needs of resource-constrained practitioners. To select the configuration that has smallest test error and takes less than the memory limit, we measure a few selected (informative and cheap) configurations on the meta-test dataset, and use the information of their test errors to predict the rest. The problem of choosing measurements is known as the "cold-start" problem in the literature on recommender systems. Then we use the model built on the meta-training datasets to estimate values for the other measurements on the meta-test dataset.

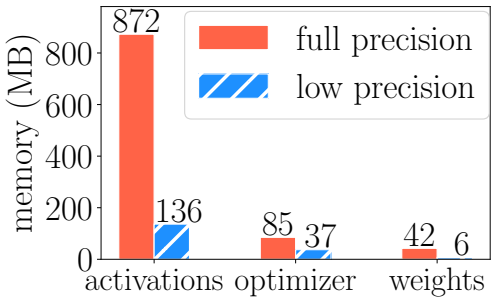
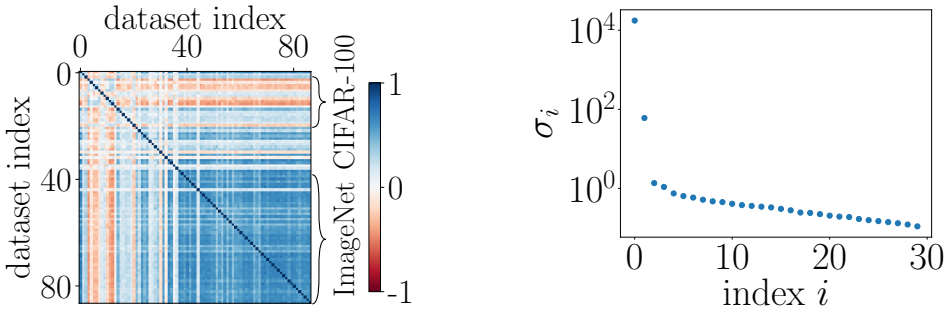



Figure 4.3: Memory usage under two training paradigms. Both train a ResNet-18 on CIFAR-10 with batch size 32.

### 4.2.1 Meta-training

On the $n$ meta-training datasets, we first theoretically compute the memory needed to evaluate each of the $d$ low-precision configurations to obtain the full memory matrix $M \in \mathbb{R}^{n \times d}$. The total memory consists of memory needed for model weights, activations, and the optimizer (gradients, gradient accumulators, etc.) [118]. Among these three types, activations typically dominate, as shown in Figure 4.3. Thus using lower precision for activations drastically reduces memory usage. Additionally, empirical studies [154] have shown that adopting higher precision formats for the optimizer can substantially improve the accuracy of the trained model. Since the optimizer usually requires a relatively small memory, it is often the best to use a higher precision format for this component. Thus for each low-precision configuration, it is typical to use a lower precision for network weights and activations, and a higher precision for the optimizer, resulting in combinatorially many choices in total. An example of the memory usage in this low-precision scenario is also shown in Figure 4.3.

In practice, ML tasks are often correlated: for example, meta-training datasets might be subsets of a large dataset like CIFAR-100 or ImageNet. The ranking of different configurations tends to be similar on similar tasks. For




(a) dataset correlation

(b) singular value decay

Figure 4.4: Kendall tau correlation of test error of all configurations between all pairs of datasets, and singular value decay of corresponding error matrix. Strong correlations allow PEPPP to succeed with a few measurements. Details in Appendix C.1.

example, we computed the test errors of 99 low-precision configurations on 87 datasets (both listed in Appendix C.1) to form a performance vector in $\mathbb{R}^{99}$ for each dataset. We use the Kendall tau correlation to characterize the alignment between the ranking of errors incurred by different configurations on two datasets: the Kendall tau correlation is 1 if the order is the same and -1 if the order is reversed. As shown in Figure 4.4(a), similar datasets have larger correlations: for example, datasets with indices 38 – 87 correspond to ImageNet subproblems such as distinguishing types of fruit or boat. Notice also that some dataset pairs have configuration performance rankings that are negatively correlated. The corresponding error matrix $E$ concatenates the performance vector for each dataset. It is not low rank, but its singular values decay rapidly, as shown in Figure 4.4(b). Hence we expect low rank approximation of this matrix from a few measurements to work well: it is not necessary to measure every configuration.

In the uniform sampling scheme, we sample measurements uniformly at random to obtain a partially observed error matrix $P_\Omega(E)$. We then estimate the

full error matrix $E$ using a low rank matrix completion method to form estimate $\widehat{E}$. Here, we use SOFTIMPUTE [91, 57] (Algorithm 9, Appendix C.2). Using the estimated error matrix $\widehat{E}$ and computed memory matrix $M$, we compute the Pareto frontier for each dataset to understand the (estimated) error-memory tradeoff. Section 4.4 Figure 4.8 shows an example.

In the non-uniform sampling scheme, we sample measurements with non-uniform probabilities, and use a weighted variant of SOFTIMPUTE, weighting by the inverse sampling probability, to complete the error matrix [88]. Then we estimate the Pareto frontier in the same way as above. Non-uniform sampling is useful to reduce the memory needed for measurements. To this end, we construct a probability matrix $P$ by entry-wise transformation $P_{ij} = \sigma(1/M_{ij})$, in which the monotonically increasing $\sigma : \mathbb{R} \to [0, 1]$ maps inverse memory usage to a sampling probability. In this way, we make more low-memory measurements, and thus reduce the total memory usage.

### 4.2.2 Meta-test

Having estimated the embedding of each configuration, our goal is to quickly compute the error-memory tradeoff on the meta-test dataset, avoiding the exhaustive search of all possible measurements.

We first compute the memory usage of each low-precision configuration on the meta-test dataset. With this information and guided by meta-training, we evaluate only a few (cheap but informative) configurations and predict the rest. Users then finally select the non-dominated configuration with highest allowable memory. An example of the process is shown in Figure 4.5.

PEPPP uses Experiment Design with Matrix Factorization (ED-MF) described below to choose informative configurations. With more time, ED-MF evaluates more configurations, improving our estimates. We may also set a hard cutoff on memory: we do not evaluate low-precision configurations exceeding the memory limit. This setting has been studied in different contexts, and is called *active learning* or *sequential decision making*.

ED-MF picks measurements to minimize the variance of the resulting estimate. Specifically, we factorize the true (or estimated) error matrix $E \in \mathbb{R}^{n\times d}$ (or $\widehat{E}$) into its best rank-$k$ approximation, and get dataset embeddings $X \in \mathbb{R}^{n\times k}$ and configuration embeddings $Y \in \mathbb{R}^{d\times k}$ as $E \approx X^\top Y$ [46, 141]. On a meta-test dataset, we denote the error and memory vectors of the configurations as $e^{\text{new}} \in \mathbb{R}^d$ and $m^{\text{new}} \in \mathbb{R}^d$, respectively. We model the error on the new dataset as $e^{\text{new}} = Y^\top x^{\text{new}} + \epsilon \in \mathbb{R}^d$, where $x^{\text{new}} \in \mathbb{R}^k$ is the embedding of the new dataset and $\epsilon \in \mathbb{R}^d$ accounts for the errors from both measurement and low rank decomposition. We estimate the embedding $x^{\text{new}}$ from a few measurements (entries) of $e^{\text{new}}$ by least squares. Note that this requires at least $k$ measurements on the meta-test dataset to make meaningful estimations, in which $k$ is the rank for matrix factorization.

If $\epsilon \sim \mathcal{N}(0, \sigma^2 I)$, the variance of the estimator is $\left(\sum_{j\in S} y_j y_j^\top\right)^{-1}$, in which $y_j$ is the $j$th column of $Y$. D-optimal experiment design selects measurements on the new dataset by minimizing (a scalarization of) the variance,

$$\begin{array}{ll} \text{minimize} & \log\det\left(\sum_{j\in S} y_j y_j^\top\right)^{-1} \\ \text{subject to} & |S| \le l \\ & S \subseteq [d], \end{array} \tag{4.1}$$

to find $S$, the set of indices of configurations to evaluate. The positive integer

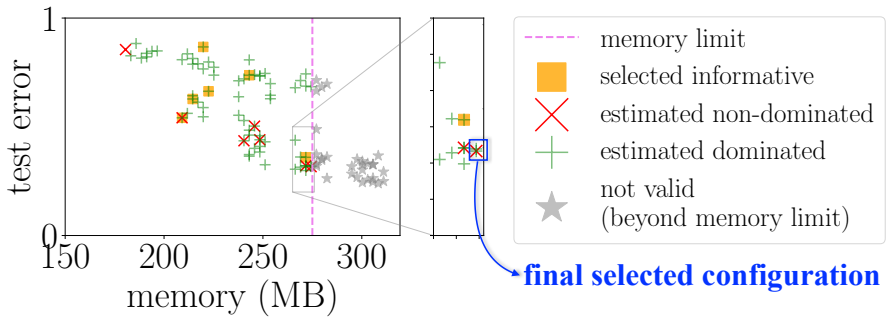



Figure 4.5: Meta-test on CIFAR-10. After meta-training on all other datasets in Appendix C.1 Table C.1, we use ED-MF to choose six informative measurements (orange squares) with a 275MB memory limit for each measurement on CIFAR-10. Then we estimate test errors of other configurations by ED-MF, and restrict our attention to configurations that we estimate to be non-dominated (red x's). Note some of these are in fact dominated, since we plot true (not estimated) test error! Finally we select the estimated non-dominated configuration with highest allowable memory (blue square).

$l$ bounds the number of measurements. Given a memory cap $m_{\max}$, we replace the second constraint above by $S \subseteq T$, in which $T = \{j \in [d] \mid m_j^{\text{new}} \leq m_{\max}\}$ is the set of feasible configurations. Since Problem 4.1 is combinatorially hard, we may either relax it to a convex optimization problem by allowing decision variables to have non-integer values between 0 and 1, or use a greedy method to incrementally choose measurements (initialized by measurements chosen by the column-pivoted QR decomposition [51, 49]). We compare these two approaches in Appendix C.3.

In Section 4.4.2, we compare ED-MF with competing techniques and show it works the best to estimate the Pareto frontier and select the final configuration.

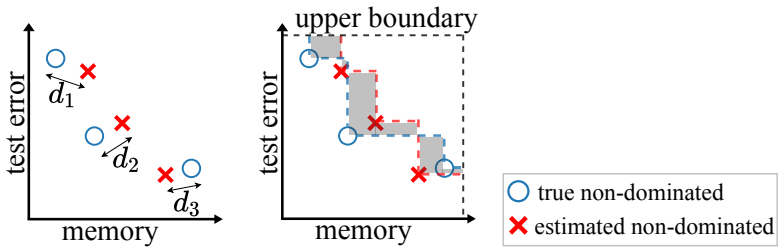
## 4.3 Python implementation

The code for PEPPP and experiments is in the GitHub repository at `https://github.com/chengrunyang/peppp`. We use QPyTorch [151] to simulate low-precision formats on standard hardware. The lower and higher precision formats are implemented as two quantization layers that are added to within the forward pass of the regular architecture classes that inherit `torch.nn.Module`.

## 4.4 Experiments and discussions

The 87 datasets and 99 low-precision configurations in experiments are listed in Appendix C.1. The datasets consist of natural and medical images from various domains. Apart from CIFAR-10, the datasets include 20 CIFAR-100 partitions from mutually exclusive subsets, based on the superclass labels. They also include 50 subsets of ImageNet [33], which contains over 20,000 classes grouped into multiple major hierarchical categories like fungus and amphibian; each of the 50 datasets come from different hierarchies. Finally, we use 16 datasets from the visual domain benchmark [133] to increase the diversity of domains.

The 99 low-precision configurations use mixed-precision as described in Section 4.2.1. Format A (for the activations and weights) uses 5 to 9 bits; Format B (for the optimizer) ranges from 14 to 20 bits. For each, these bits may be split arbitrarily between the exponent and mantissa. Across all configurations, we use ResNet-18, ResNet-34 and VGG [114] variants (11, 13, 16, 19) with learning rate 0.001, momentum 0.9, weight decay 0.0005 and batch size 32. Each training uses only 10 epochs as an early stopping strategy [146] to prevent overfitting.




(a) convergence (b) HyperDiff

Figure 4.6: Illustration of Pareto frontier metrics. (a) Convergence is the average distance from each estimated Pareto optimal point to its closest true point: average$(d_1, d_2, d_3)$. (b) HyperDiff is the absolute difference in area of feasible regions given by the true and estimated Pareto optimal points: the shaded area between Pareto frontiers.

All the measurements take 35 GPU days on NVIDIA® GeForce® RTX 3090.

Developing a method to select the above hyperparameters at the same time as the low-precision format is an important topic for future research, but not our focus here. In Section 4.4.3 we demonstrate how PEPPP can naturally extend to efficiently select both optimization and low-precision hyperparameters.

A number of metrics may be used to evaluate the quality of the obtained Pareto frontier in a multi-objective optimization problem [140, 83, 3]: the distance between approximated and true frontiers (convergence), the uniformity of the distances between neighboring Pareto optimal points (uniformity), how well the frontier is covered by the approximated points (spread), etc. In our setting, the memory is accurate and the test error is estimated, so we do not have full control of the uniformity and spread of the 2-dimensional frontiers between test error and memory. As illustrated in Figure 4.6, we evaluate the quality of our estimated Pareto frontiers by the following two metrics:

**Convergence** [31] between the sets of true and estimated Pareto optimal points $\mathcal{P}$, $\widehat{\mathcal{P}}$ is $\frac{1}{|\widehat{\mathcal{P}}|}\sum_{v\in\widehat{\mathcal{P}}}\text{dist}(v,\mathcal{P})$, where the distance between point $v$ and set $\mathcal{P}$ is

$\text{dist}(v, \mathcal{P}) = \min\{\|v - w\| : w \in \mathcal{P}\}$. This is a surrogate for the distance between Pareto frontiers.

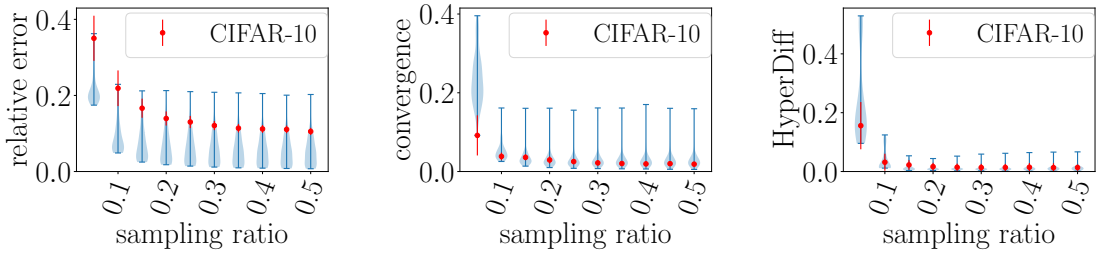
**Hypervolume difference (HyperDiff)** [155, 27, 140] is the absolute difference between the volumes of solution spaces dominated by the true and estimated Pareto frontiers. This metric improves with better convergence, uniformity, and spread. Its computation requires an upper boundary for each resource.

When computing these metrics, we normalize the memories by proportionally scaling them to between 0 and 1. To evaluate the matrix completion performance, we use the relative error defined as $\|\widehat{v} - v\|/\|v\|$, in which $\widehat{v}$ is the predicted vector and $v$ is the true vector.

### 4.4.1 Meta-training

We study the effectiveness of uniform and non-uniform sampling schemes. For simplicity, we regard all 87 available datasets as meta-training in this section. In SOFTIMPUTE, we use $\lambda = 0.1$ (chosen by cross-validation from a logarithmic scale); and we choose a rank 5 approximation, which accounts for 78% of the variance in the error matrix (as shown in Figure 4.4(b)).

In the uniform sampling scheme, we investigate how many samples we need for accurate estimates: we sample the error matrix at a number of different ratios, ranging from 5% to 50%, and complete the error matrix from each set of sampled measurements. In uniform sampling, the sampling ratio is also the percentage of memory we need to make the measurements in parallel, compared to exhaustive search. Hence we show the relationship between Pareto

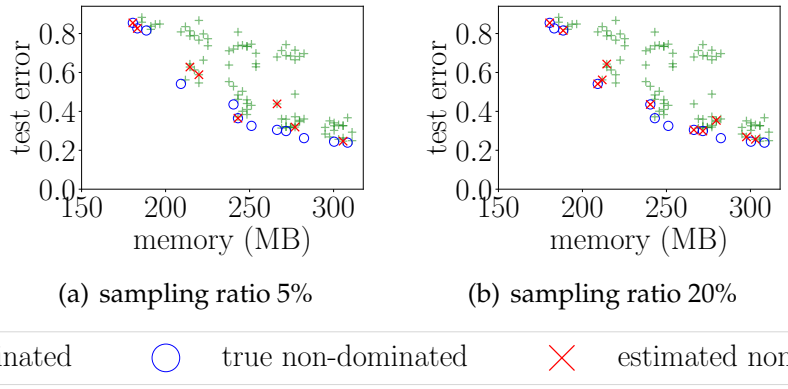



(a) relative error (b) convergence (c) HyperDiff

Figure 4.7: Pareto frontier estimation in PEPPP meta-training, with uniform sampling of configurations. The violins show the distribution of the performance on individual datasets, and the error bars (blue) show the range. The red error bars show the standard deviation of the error on CIFAR-10 across 100 random samples of the error matrix. Figure (a) shows the matrix completion error for each dataset; Figure (b) and (c) show the performance of the Pareto frontier estimates. Modest sampling ratios (around 0.1) already yield good performance.

frontier estimation performance and sampling ratio in Figure 4.7. We can see the estimates are more accurate at larger sampling ratios, but sampling 20% of the entries already suffices for good performance.

As a more intuitive example, Figure 4.8 shows the error-memory tradeoff on CIFAR-10. Compared to the estimated Pareto optimal points at sampling ratio 5%, the ones at 20% are closer to the true Pareto frontier (better convergence), lie more evenly (better uniformity) and cover the true Pareto frontier better (better spread), as shown by the respective convergence and HyperDiff values. The non-uniform sampling scheme has a similar trend and is shown in Appendix C.5.2.




(a) sampling ratio 5%

(b) sampling ratio 20%

Figure 4.8: Error vs memory on CIFAR-10 with true and estimated Pareto frontiers from uniform sampling in PEPPP meta-training. A 20% uniform sample of entries yields a better estimate of the Pareto frontier (convergence 0.03 and HyperDiff 0.02) compared to a 5% sample (convergence 0.09 and HyperDiff 0.16).

### 4.4.2 Meta-leave-one-out cross-validation (meta-LOOCV)

Now suppose that we have already collected measurements on the meta-training datasets to form a meta-training error matrix $E$ (or its low rank approximation $\widehat{E}$). On the meta-test dataset, PEPPP estimates the Pareto frontier by the active learning technique ED-MF. We compare ED-MF with a few other active learning techniques: Random selection with matrix factorization (RANDOM-MF), QR decomposition with column pivoting and matrix factorization (QR-MF) and two Bayesian optimization techniques (BO-MF and BO-FULL), to understand whether the strong assumptions in ED-MF (low rank and Gaussian errors) are a hindrance or a help. An introduction to these techniques can be found in Appendix C.5.1.

We use rank 3 for matrix factorization: $Y_{:,j} \in \mathbb{R}^3$ for each $j \in [d]$. In BO, we tune hyperparameters on a logarithmic scale and choose the RBF kernel with length scale 20, white noise with variance 1, and $\xi = 0.01$. Table 4.1 shows the meta-LOOCV settings for each acquisition technique. We compare the tech-

Table 4.1: Meta-LOOCV experiment settings

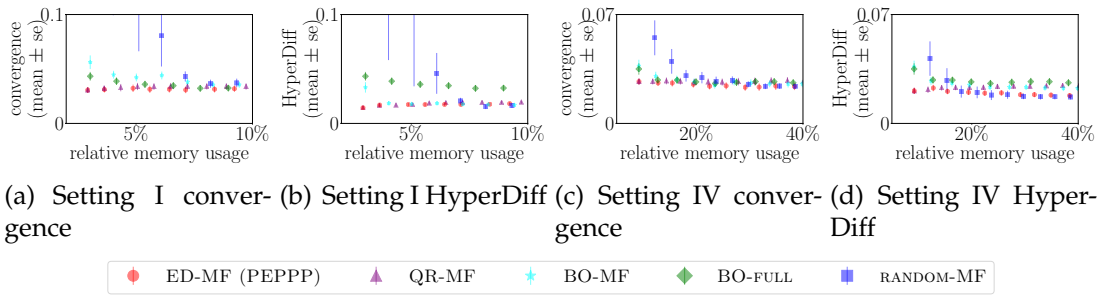
| meta-training error matrix | memory cap on meta-test? | |
|---|---|---|
| | no | yes |
| full | **I** | II |
| uniformly sampled | III | **IV** |
| non-uniformly sampled | V | VI |

niques at a range of number of configurations to measure in each meta-LOOCV split, resembling what practitioners do in hyperparameter tuning: evaluate an informative subset and infer the rest. Setting I is the most basic, Setting IV and VI are the most practical. We only show results of Setting I and IV in the main paper, and defer the rest to Appendix C.5.

In Setting I, we do meta-LOOCV with the full meta-training error matrix in each split and do not cap the memory for meta-test. This means we evaluate every configuration on the meta-training datasets. We can see from Figure 4.9(a) and 4.9(b) that:

- ED-MF stably outperforms under both metrics, especially with fewer measurements.
- QR-MF overall matches the performance of ED-MF.
- BO-MF, BO-FULL and RANDOM-MF underperform ED-MF and QR-MF at lower memory usage, but often match their performance with higher memory.

Practitioners may have only collected part of the meta-training performance, and desire or are limited by a memory cap when they do meta-test on the new dataset. In Setting IV, we cap the single-configuration memory usage for meta-test at 816MB, the median memory of all possible measurements across config-




(a) Setting I convergence (b) Setting I HyperDiff (c) Setting IV convergence (d) Setting IV HyperDiff

Figure 4.9: Pareto frontier estimates in meta-LOOCV Setting I and IV (with a 20% meta-training sampling ratio and an 816MB meta-test memory cap). Each error bar is the standard error across datasets. The x axis measures the memory usage relative to exhaustively searching the permissible configurations. ED-MF consistently picks the configurations that give the best PF estimates.

urations and datasets. Additionally, we uniformly sample 20% configurations from the meta-training error matrix in each split. In Figure 4.9(c) and 4.9(d), we can see similar trends as Setting I, except that QR-MF is slightly worse.

Ultimately, users would want to select a configuration that both achieves a small error and takes lower memory than the limit. As shown in Figure 4.2, PEPPP offers users the predicted Pareto frontier and chooses the non-dominated configuration with the highest allowable memory and hence the lowest error. We compare the acquisition techniques with the "random high-memory" baseline that randomly chooses a configuration that takes the highest allowable memory: an approach that follows the "higher memory, lower error" intuition. Figure 4.10 shows an example.

Overall, among all approaches and across all memory usage, ED-MF outperforms, especially at a smaller number of measurements. Compared to BO techniques, ED-MF also enjoys the benefit of having less hyperparameters to tune. Although techniques like QR-MF and RANDOM-MF are easier to implement, the additional cost of ED-MF is much smaller than the cost of making measurements: neural network training and testing.

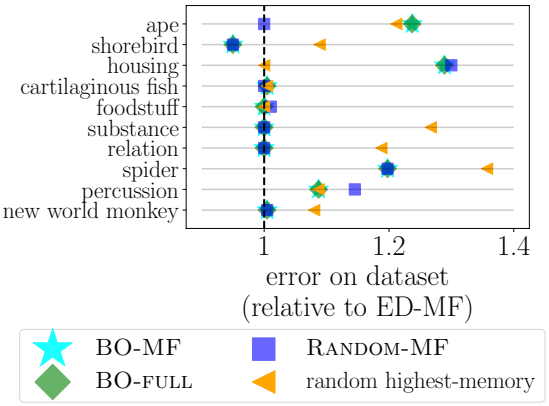



Figure 4.10: Relative performance with respect to ED-MF in meta-test Setting IV when making 3 measurements (memory usage ~10%) on 10 ImageNet partitions. ED-MF outperforms in most cases.

### 4.4.3 Tuning optimization hyperparameters

Previously, we have shown that PEPPP can estimate the error-memory tradeoff and select a promising configuration with other hyperparameters fixed. In practice, users may also want to tune hyperparameters like learning rate to achieve the lowest error. In Appendix C.5.4, we tune the number of epochs and learning rate in addition to precision, and show that the methodology can be used in broader settings.

### 4.4.4 Meta-learning across architectures

The meta-learning in previous sections were conducted on ResNet-18. In Appendix C.5.5, we show that on 10 ImageNet partitions, PEPPP is also capable of estimating the error-memory tradeoff of ResNet-34, VGG-11, VGG-13, VGG-16 and VGG-19 competitively. Moreover, the meta-learning across architectures works better than considering each architecture separately.

## 4.5 Conclusion

We proposes PEPPP, a meta-learning system to select low-precision configurations that leverages training information from related tasks to efficiently pick the perfect precision given a memory budget. Built on low rank matrix completion with active learning, PEPPP estimates the Pareto frontier between memory usage and model performance to find the best low-precision configuration at each memory level. By reducing the cost of hyperparameter tuning in low-precision training, PEPPP allows practitioners to efficiently train accurate models.

CHAPTER 5

# TABNAS: RESOURCE-CONSTRAINED NEURAL ARCHITECTURE SEARCH ON TABULAR DATASETS

This chapter presents TabNAS [142], a resource-constrained neural architecture search method to search for feedforward networks on tabular datasets.

## 5.1 Introduction

It is often observed that to make a machine learning model better, one can scale it up. However this is not always possible when machine learning models are deployed since larger networks are also more expensive as measured by inference time, memory, energy, etc. These costs limit the application of large models: training these models is unsustainable, and inference is often too slow to satisfy end user requirements.

One of the most widespread applications of machine learning in industry is tabular data in, e.g., finance, advertising and medicine. It was only recently that in these applications deep learning was able to outperform classical tree-based models [50, 70].

For vision, optimizing the models to make them suitable for practical deployment often relies on Neural Architecture Search (NAS). Most NAS literature targets these convolutional networks on vision benchmarks [86, 20, 62, 147]. Despite the practical importance of tabular data, NAS research on this topic is quite limited [41, 40].

Weight-sharing allows us to reduce the cost of NAS by training a *SuperNet*

that is the superset of all architecture candidates [10]. This trained SuperNet is then used to estimate the quality of the individual architectures, the so-called *child networks*, by only activating a subset of components of that architecture and running an evaluation. To efficiently find the most promising child networks, Reinforcement Learning (RL) has shown to be effective [103, 20, 11] on vision problems.

In our experiments, we observe that a direct application of the vision approaches for tabular data is suboptimal. We started from the TuNAS [11] approach from vision and observed that this struggled to find the optimal architectures for tabular datasets. The failure is caused by the interaction of the search space and the RL controller. In vision, a popular approach is to use a factorized RL controller, which assumes that all choices can be made independently. The search space consists of a limited number of options per layer. In tabular data, we need more options per layer, but there are fewer layers overall. Feedforward networks with *bottleneck structures* often outperform other feed-forward networks of similar size on tabular data. In such a *bottleneck architecture*, there exists at least one hidden layer that is much narrower than its preceding and following layers. A popular hypothesis is that its weights resemble the low-rank factors of a wider network, and thus mimics the behavior of the latter with less cost [74, 26]. These bottleneck structures often have a a very good tradeoff between cost and quality (more examples in Appendix D.2.2, Table D.3), but finding these bottleneck structures is difficult for a factorized RL controller. To understand why, we consider the following toy example with 2 layers, illustrated in Figure 5.1. For the each layer, we can choose a layer size of $2, 3$ or $4$ and the maximum compute budget is set to 25. The optimal solution is to set the size of layer 1 to 4 and layer 2 to 2. Finding this solution is difficult with

a cost penalty. The RL controller is initialized with uniform probabilities. As a result, it is quite likely that the RL controller will initially be penalized heavily when choosing option 4 for the first layer, since two thirds of the choices for the second layer will result in a model that is too expensive. As a result, option 4 for the first layer is quickly discarded by the RL controller and we get stuck in a local optimum.

To circumvent this problem, one could attempt to learn a non-factorized probability distribution. However, this requires a more complicated model, e.g., an LSTM, that is often more difficult to tune. We propose a different solution inspired by rejection sampling. We only update the RL controller when the sampled model satisfies our cost constraint. The RL controller is then discouraged from sampling poor models within the cost constraint and encouraged to sample the high quality models. Rather than penalizing models which violates the cost constraints, the controller silently discards them, thereby circumventing the local optimum.

Our contributions can be summarized as follows:

- We first identify failure cases of existing resource-aware NAS methods on tabular data, and link these cases to the cost penalty in the reward.
- We then propose and evaluate an alternative: a rejection mechanism which ensures that the RL controller can only select architectures that satisfy the user-specified resource constraint. Instead of reward shaping, this extra rejection step allows the RL controller to explore parts of the search space which would otherwise be overlooked.
- The rejection mechanism also introduces a systematic bias into the RL gradient updates, which can skew the search results. To compensate for this

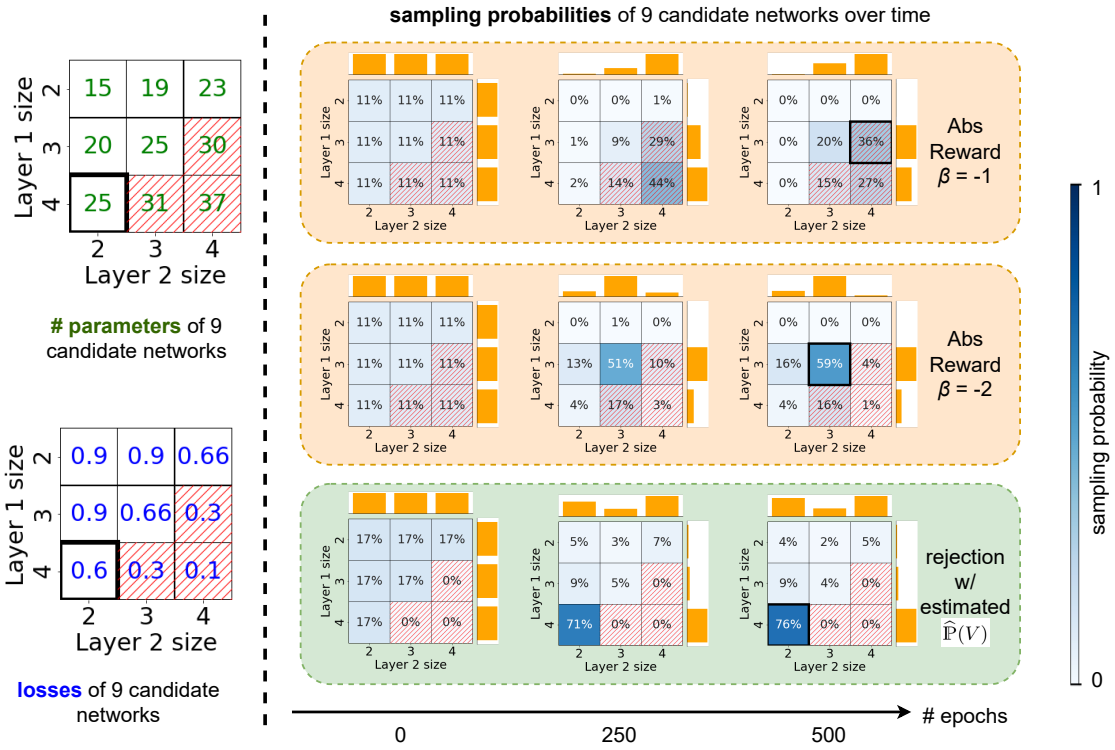



Figure 5.1: **A toy example for tabular NAS** in the 2-layer search space with a 2-dimensional input and a limit of 25 parameters. The left half shows the number of parameters and loss of each candidate architecture in the search space. The infeasible architectures have striped patch in the corresponding cells. The bottom left cell squared in bold is the global optimal architecture with hidden size 1 = 4 and hidden size 2 = 2. The right half shows the change of sampling probabilities in weight-sharing NAS with different RL rewards. Each cell represents an architecture; the sampling probability value is shown both as a percentage in the cell, and with the color intensity indicated by the right colorbar. The orange bars on the top and right sides show the sampling probability distribution among size candidates for each layer. With the Abs Reward, the sampling probability of each architecture is the product of sampling probabilities of its layer sizes; with the rejection-based reward, the sampling probability of an infeasible architecture is 0, and that of a feasible architecture gets reweighted by the sum of probabilities of all feasible architectures. At epoch 500, the cell squared in bold shows the architecture picked by the corresponding RL controller. RL with the Abs Reward $Q(x) + \beta|T(x)/T_0 - 1|$ proposed in TuNAS [11] either converges to a feasible but suboptimal architecture ($\beta = -2$, middle row) or violates the resource constraint ($\beta = -1$, top row). Other latency-aware reward functions show similar failures. In contrast, our new rejection-based controller converges to the optimal solution (bottom row).

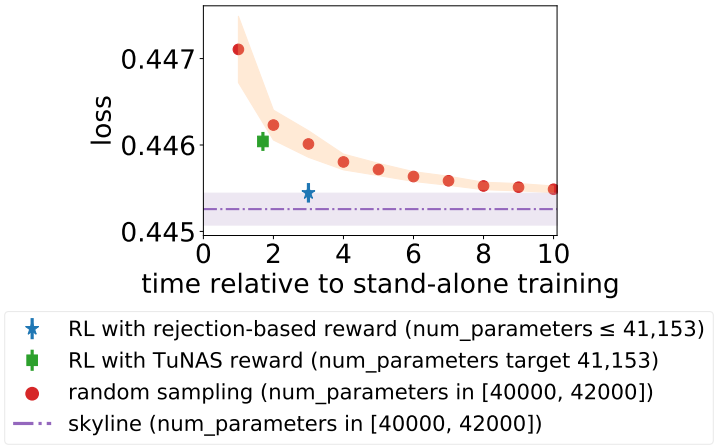



Figure 5.2: Rejection-based reward distributionally outperforms random search and resource-aware Abs Reward on the Criteo dataset within a 3-layer search space. All error bars and shaded regions are 95% confidence intervals. The x axis is the time relevant to training a single architecture in the search space. Results of random sampling comes from 100 independent runs on 50 architectures within the number of parameters range. The result of each RL method comes from 5 independent runs. The skyline is the performance of 3 independent retrains of the best architecture that is found by 3 independent exhaustive searches. More details in Appendix D.2.2.

bias, we introduce a theoretically motivated and empirically effective correction into our gradient updates. This correction can be computed exactly for small search spaces, and we show how to efficiently approximate it with Monte-Carlo sampling when the space is large.

These contributions form TabNAS, our RL-based weight-sharing NAS with the rejection-based reward that can robustly and efficiently find a feasible architecture that has optimal performance within the resource constraint. Figure 5.2 shows an example.

## 5.2 More notations and terminologies

In addition to the notations and terminologies defined in Section 1.2, we use more defined as below for this chapter.

**Math basics.** We use stop_grad($f$) to denote the constant value (with gradient 0) corresponding to a differentiable quantity $f$. This is equivalent to `tensorflow.stop_gradient(f)` in TensorFlow [1] or `f.detach()` in PyTorch [100]. $\nabla$ denotes the gradient with respect to the variable in the context.

**Weight, architecture, and hyperparameter.** We use *weights* to refer to the parameters of the neural network and are trained in the neural network training. The *architecture* of a neural network is the structure of how nodes are connected; examples of architectural choices are hidden layer sizes and activation types. *Hyperparameters* are the the non-architectural parameters that control the training process of either stand-alone training or RL, including learning rate, optimizer type, optimizer parameters, etc.

**Neural architecture.** A neural network with specified architecture and hyperparameters is called a *model*. We only consider fully-connected feedforward networks (FFNs) in this work, since they can already achieve SOTA performance on tabular datasets [70]. The number of hidden nodes after each weight matrix and activation function is called a *hidden layer size*. We denote a single network in our search space with hyphen-connected choices. For example, when searching for hidden layer sizes, in the space of 3-hidden-layer ReLU networks, 32-144-24 denotes the candidate where the sizes of the first, second and third hidden layers are 32, 144 and 24, respectively. We only search for ReLU networks; for brevity, we will not mention the activation function type in the sequel.

**Loss-resource tradeoff and reference architectures.** As shown in the tradeoff plots in Figure 5.3, within the hidden layer size search space, the validation loss in general decreases with the increase of the number of parameters, giving the loss-resource tradeoff. Loss and number of parameters can be understood as two *costs* for the NAS problem. Thus there are Pareto-optimal models that achieve the smallest loss among all models with a given bound on the number of parameters. With an architecture that outperforms others with a similar or fewer number of parameters, we do resource-constrained NAS with the number of parameters of this architecture as the resource target or constraint. We call this architecture the *reference architecture* (or *reference*) of NAS, and its performance the *reference performance*. We do NAS with the goal of *matching* (the size and performance of) the reference. Note that the RL controller only has knowledge of the number of parameters of the reference, and is not informed of its hidden layer sizes.

**Search space.** When searching $L$-layer networks, we use capital letters like $X = X_1\text{-}\cdots\text{-}X_L$ to denote the random variable of sampled architectures, in which $X_i$ is the random variable for the size of the $i$-th layer. We use lowercase letters like $x = x_1\text{-}\cdots\text{-}x_L$ to denote an architecture sampled from the distribution over $X$, in which $x_i$ is an instance of the $i$-th layer size. When there are multiple samples drawn, we use a bracketed superscript to denote the index over samples: $x^{(k)}$ denotes the $k$-th sample. The search space $S = \{s_{ij}\}_{i\in[L],j\in[C_i]}$ has $C_i$ choices for the $i$-th hidden layer, in which $s_{ij}$ is the $j$-th choice for the size of the $i$-th hidden layer: for example, when searching for a one-hidden-layer network with size candidates $\{5, 10, 15\}$, we have $s_{13} = 15$.

**Reinforcement learning.** The RL algorithm learns the set of logits $\{\ell_{ij}\}_{i\in[L],j\in[C_i]}$,

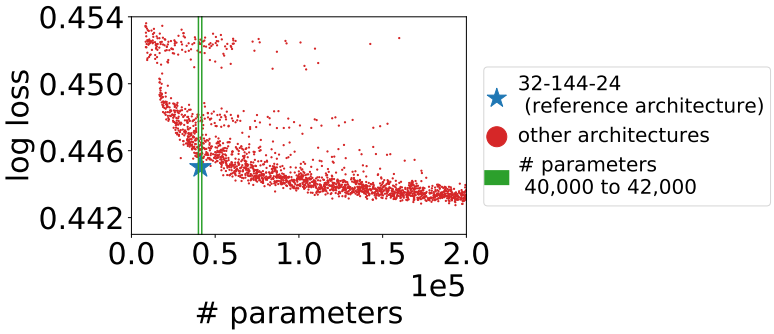



Figure 5.3: Tradeoff between loss and number of parameters on Criteo within a 3-layer search space. The search space and Pareto-optimal architectures are shown in Appendix D.2.2. We use logistic loss as the loss metric. When training each architecture 5 times, the standard deviation (std) across different runs is 0.0002, meaning that the architectures whose performance difference is larger than 2× std are qualitatively different with high probability.

in which $\ell_{ij}$ is the logit associated with the $j$-th choice for the $i$-th hidden layer. With a fully factorized distribution of layer sizes (we learn a separate distribution for each layer), the probability of sampling the $j$-th choice for the $i$-th layer $p_{ij}$ is given by the SoftMax function: $p_{ij} = \exp(\ell_{ij}) / \sum_{j \in [C_i]} \exp(\ell_{ij})$. In each RL step, we sample an architecture $y$ to compute the single-step RL objective $J(y)$, and update the logits with $\nabla J(y)$: an unbiased estimate of the gradient of the RL value function.

**Resource metric and number of parameters.** We use the number of parameters, which can be easily computed for neural networks, as a cost metric in this work. Our approach does not depend on the specific cost used.

## 5.3 Methodology

The methodologies we use for NAS can be decomposed into three main components: weight-sharing with layer warmup, REINFORCE with one-shot search,

and Monte Carlo (MC) sampling with rejection.

As an overview, our method starts with a SuperNet, which is a network that layer-wise has width to be the largest choice within the search space. We first stochastically update the weights of the entire SuperNet to "warm up" over the first 25% of search epochs. Then we alternate between updating the shared model weights (which are used to estimate the quality of different child models) and the RL controller (which focuses the search on the most promising parts of the space). In each iteration, we first sample a child network from the current layer-wise probability distributions and update the corresponding weights within the SuperNet (weight update), then sample another child network to update the layerwise logits that give the probability distributions (RL update). The latter RL update is only performed if the sampled network is feasible, in which case we use rejection with MC sampling to update the logits with a sampling probability conditional on the feasible set.

To avoid overfitting, we split the labelled portion of a dataset into training and validation splits. Weight updates are carried out on the training split; RL updates are performed on the validation split.

### 5.3.1 Weight sharing with layer warmup

The weight-sharing approach has shown success on various computer vision tasks and NAS benchmarks [103, 10, 20, 11]. To do search for an FFN on tabular datasets, we build a SuperNet where the size of each hidden layer is the largest value in the search space. Figure 5.4 shows an example. When we sample a child network with a hidden layer size $\ell_i$ smaller than the SuperNet, we only

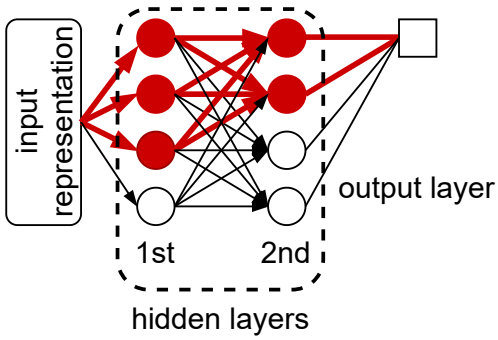



Figure 5.4: Illustration of weight-sharing on two-layer FFNs for a binary classification task. The edges denote weights, and arrows at the end of lines denote activations. The circles denote hidden nodes, and the two squares in the output layer denote the output logits. The search space of the size of each hidden layer is $\{2, 3, 4\}$, thus the SuperNet is a two-layer network with size 4-4. At this moment, the controller picks the child network 3-2 in the SuperNet 4-4, thus only the first 3 hidden nodes in the first hidden layer and the first 2 hidden nodes in the second hidden layer, together with the connected edges (in red), are enabled to compute the output logits.

use the first $\ell_i$ hidden nodes in that layer to compute the output in the forward pass. In weight updates, only the weights in the child network are updated in the backward pass. In RL updates, only the weights of the child network are used to estimate the quality reward that is used to update logits.

In weight-sharing NAS, warmup helps to ensure that the SuperNet weights are sufficiently trained to properly guide the RL updates [11]. With probability $p$, we train all weights of the SuperNet, and with probability $1 - p$ we only train the weights of a random child model. When we run architecture searches for FFNs, we do warmup in the first 25% epochs, during which the probability $p$ linearly decays from 1 to 0 (Figure 5.5(a)). The RL controller is disabled during this period.

### 5.3.2 One-shot training and REINFORCE

We do NAS on FFNs with a REINFORCE-based algorithm. Previous works have used this type of algorithm to search for convolutional networks on vision tasks [124, 20, 11].

When searching for $L$-layer FFNs, we learn a separate probability distribution over $C_i$ size candidates for each layer. The distribution is given by $C_i$ logits via the SoftMax function. Each layer has its own independent set of logits. With $C_i$ choices for the $i$th layer, where $i = 1, 2, \cdots, L$, there are $\prod_{i\in[L]} C_i$ candidate networks in the search space but only $\sum_{i\in[L]} C_i$ logits to learn. This technique significantly reduces the difficulty of RL and make the NAS problem practically tractable [20, 11].

The REINFORCE-based algorithm trains the SuperNet weights and learns the logits $\{\ell_{ij}\}_{i\in[L],j\in[C_i]}$ that give the sampling probabilities $\{\ell_{ij}\}_{i\in[L],j\in[C_i]}$ over size candidates by alternating between weight and RL updates. In each iteration, we first sample a child network $x$ from the SuperNet and compute its training loss in the forward pass. Then we update the weights in the child network with gradients of the training loss computed in the backward pass. This weight update step trains the weights of the sampled network. The weights in the architectures with larger sampling probabilities are sampled and thus trained more often. We then update the logits for the RL controller by sampling a child network $y$ that is independent of the network $x$ from the same layerwise distributions, compute the quality reward $Q(y)$ as $1 - loss(y)$ on the validation set, and then update the logits with the gradient of $J(y) = \text{stop\_grad}(Q(y) - \bar{Q}) \log \mathbb{P}(y)$: the product of the advantage of the current network's reward over past rewards (usually an exponential moving average) and the log-probability of the current sample.

The alternation creates a positive feedback loop that trains the weights and updates the logits of the large-probability child networks; thus the layer-wise sampling probabilities gradually converge to more deterministic distributions, under which one or several architectures are finally selected.

Details of a resource-oblivious version is shown as Algorithm 11 in Appendix D.1, which does not take into account a resource constraint. In Section 5.3.3, we show an algorithm that combines Monte-Carlo sampling with rejection sampling, which serves as a subroutine of Algorithm 11 by replacing the probability in $J(y)$ with a conditional version.

### 5.3.3 Rejection-based reward with MC sampling

Only a subset of the architectures in the search space $S$ will satisfy our resource constraints, $V$ denotes this set of feasible architectures. To find a feasible architecture, a resource target $T_0$ is often used in an RL reward. Given an architecture $y$, a latency-aware reward combines its quality $Q(y)$ and resource consumption $T(y)$ into a single reward. MnasNet [124] proposes the reward functions $Q(y) \times (T(y)/T_0)^\beta$ and $Q(y) \times \max\{1, (T(y)/T_0)^\beta\}$ while TuNAS [11] proposes the absolute value reward (or Abs Reward) $Q(y) + \beta|T(y)/T_0 - 1|$. In these approaches $\beta$ is a hyperparameter that needs careful tuning. The idea behind these reward functions is to encourage models with high quality with respect the resource target.

We found that in tabular data experiments, RL controllers using these resource-aware rewards above can struggle to discover *bottleneck structures* – where we select a large number of filters for the $i$th layer of the network but a

small number of filters for the $i+1$st layer. Figure 5.1 shows a toy example in the search space in Figure 5.4, in which we know the validation losses of each child network and only train the RL controller for 200 steps. The optimal network is 4-2 among architectures with number of parameters no more than 25, but the RL controller rarely chooses it. In Section 5.4.1, we show examples of this on real datasets.

Such a phenomenon reveals a gap between the true distribution we want to sample from and the distributions given by factorized search space that we are truly sampling from:

- We *only* want to sample from the set of feasible architectures $V$, whose distribution is $\{\mathbb{P}(y \mid y \in V)\}_{y \in V}$. The number of parameters (or another resource metric) of an architecture, and thus its feasibility, is determined jointly by the sizes of all layers.
- On the other hand, the factorized search space determines that we learn a separate (independent) probability distribution for the choices of each layer. While this distribution is efficient to learn, the independence assumption makes it difficult for a RL controller with a resource-aware reward to choose a bottleneck structure. A bottleneck requires the controller to select large sizes for some layers and small layer sizes for others. But decisions for different layers are made independently, and both very large and very small layer sizes, when selected independently of each other, have very negative expected rewards. Small layers are likely to have suboptimal quality, and large layers are likely to exceed the resource constraints.

To bridge the gap and efficiently learn layerwise distributions that take into account the architecture feasibility, we propose a rejection-based RL reward for

Algorithm 11. We next sketch the idea; detailed pseudocode is provided as Algorithm 12 in Appendix D.1.

REINFORCE optimizes a set of logits $\{\ell_{ij}\}_{i\in[L],j\in[C_i]}$ which define a probability distribution $p$ over architectures. In the original REINFORCE algorithm, we sample a random architecture $y$ from $p$ and then estimate its quality $Q(y)$. Updates to the logits $\ell_{ij}$ take the form $\ell_{ij} \leftarrow \ell_{ij} + \eta\frac{\partial}{\partial\ell_{ij}}J(y)$, where $\eta$ is the learning rate,

$$J(y) = \text{stop\_grad}(Q(y) - \overline{Q}) \cdot \log \mathbb{P}(y)$$

and $\overline{Q}$ is a moving average of recent rewards. If $y$ is better (resp. worse) than average then $Q(y) - \overline{Q}$ will be positive (resp. negative), so the REINFORCE update will increase (resp. decrease) the probability of sampling the same architecture in the future.

In our new REINFORCE variant, motivated by rejection sampling, we skip the REINFORCE update to the logits unless $y$ is feasible. And if $y$ is feasible, we replace the probability $\mathbb{P}(y)$ in the REINFORCE update equation with the conditional probability $\mathbb{P}(y \mid y \in V) = \mathbb{P}(y)/\mathbb{P}(y \in V)$. So $J(y)$ becomes

$$J(y) = \text{stop\_grad}(Q(y) - \overline{Q}) \cdot \log\left[\mathbb{P}(y)/\mathbb{P}(y \in V)\right].$$

We can compute the probability of sampling a feasible architecture $\mathbb{P}(V) := \mathbb{P}(y \in V)$ exactly when the search space is small, but that becomes prohibitively expensive when the space is large. In the latter case, we replace the exact probability $\mathbb{P}(y)$ with a differential approximation $\widehat{\mathbb{P}}(y)$ obtained using Monte-Carlo (MC) sampling. In each RL step, we sample $N$ architectures $\{z^{(k)}\}_{k\in[N]}$ within the search space with a proposal distribution $q$, and get

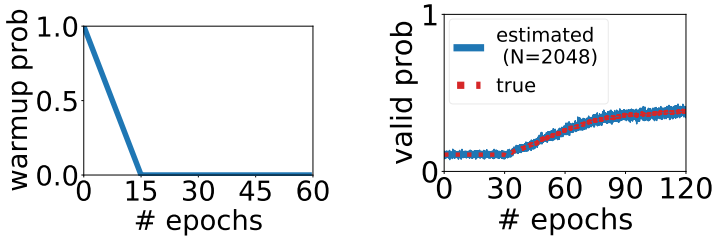
$$\widehat{\mathbb{P}}(V) = \frac{1}{N}\sum_{k\in[N]}\frac{p^{(k)}}{q^{(k)}} \cdot \mathbb{1}(z^{(k)} \in V)$$

as an estimate of $\mathbb{P}(V)$. For each $k \in [N]$, $p^{(k)}$ is the probability of sampling $z^{(k)}$ with the factorized layerwise distributions, and is thus differentiable with respect to the logits. In contrast, $q^{(k)}$ is the probability of sampling $z^{(k)}$ with the proposal distribution, and is therefore non-differentiable.

We show $\widehat{\mathbb{P}}(V)$ is an unbiased and consistent estimate of $\mathbb{P}(V)$, and $\nabla \log[\mathbb{P}(y)/\widehat{\mathbb{P}}(V)]$ is a consistent estimate of $\nabla \log[\mathbb{P}(y \mid y \in V)]$ (Appendix D.7). A larger $N$ gives better results (Section D.5); in our experiments, we need smaller than the size of the sample space to get a faithful estimate (Figure 5.5(b), Section 5.4.3 and Appendix D.3) because neighboring RL steps can correct the estimates of each other. We set $q = \text{stop\_grad}(p)$ in our experiments for convenience: use the current distribution over architectures for MC sampling. Other distributions that have a larger support on $V$ may be used to reduce the sampling variance (Appendix D.7).

At the end of NAS, we pick the layer sizes with largest sampling probabilities as the found architecture if the layerwise distributions are deterministic, or sample the distributions $m$ times and pick $n$ feasible architectures with the largest number of parameters if not. Appendix D.1 Algorithm 13 provides the full details of this procedure. Although it is cheap to use larger values, we find $m = 500$ and $n \leq 3$ suffice to find an architecture that can match the reference architecture in our experiments.

In practice, we find that the distributions often (almost) converge after 2× of the number of epochs used to train stand-alone child networks, while the distributions are often informative enough after 1× epochs, in the sense that the architectures found by Algorithm 13 are competitive.




(a) Warmup probability (b) Valid probability

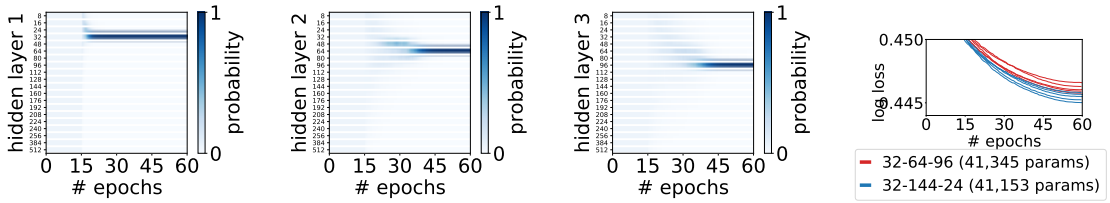
Figure 5.5: Example layer warmup and valid probabilities. Figure 5.5(a) shows our schedule of layer warmup probabilities: linearly decay from 1 to 0 in the first 25% epochs. Figure 5.5(b) shows an example of the change of true and estimated valid probabilities ($\mathbb{P}(V)$ and $\widehat{\mathbb{P}}(V)$) in a successful search, with 8,000 architectures in the search space and the number of Monte-Carlo samples $N = 1024$. Both probabilities are (nearly) constant during warmup before RL starts, then start to increase when the RL starts because of rejection sampling.

Figure 5.1 show that our rejection-based method finds the best feasible architecture, 4-2, in our toy example, when using the $\widehat{\mathbb{P}}(V)$ estimated by MC sampling.

## 5.4 Experiments and discussions

We ran all experiments using TensorFlow on a Cloud TPU v2 with 8 cores. We use a 1,027-dimensional input representation for Criteo, 180 features for Volkert and 128 features for Aloi[1]. More experiment setup details can be found in Appendix D.2.

[1]Our paper takes these features as given. It is worth noting that methods proposed in feature engineering works like [73] and [84] are complementary to and can work together with TabNAS.




(a) Layer 1 (final choice 32) (b) Layer 2 (final choice 64) (c) Layer 3 (final choice 96) (d) Retrain performance

Figure 5.6: **Failure case of the Abs Reward on Criteo** in a search space of 3-layer FFNs. The change of sampling probabilities and comparison of retrain performance between the 32-144-24 reference and the 32-64-96 architecture found with the $Q(x)+\beta|T(x)/T_0-1|$ Abs Reward, the target for the reward was 41,153 parameters. Repeated runs of the same search find the same architecture. Figure 5.6(d) shows the change of validation losses across 5 retrains of 32-64-96 (NAS-found) and 32-144-24 (reference).

### 5.4.1 When do previous RL rewards fail?

Section 5.3.3 discussed the resource-aware RL rewards and highlighted a potential failure case. In this section, we show several failure cases of the resource-aware rewards $Q(x)(T(x)/T_0)^\beta$, $Q(x)\max\{1,(T(x)/T_0)^\beta\}$ and $Q(x)+\beta|T(x)/T_0-1|$ on our tabular datasets; more failure cases can be found in Appendix D.3.

#### Criteo – 3 Layer Search Space

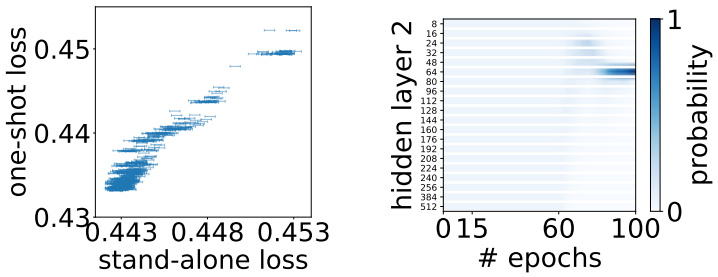
We use the 32-144-24 reference architecture, which has 41,153 parameters. Figure 5.3 gives an overview of the costs and the losses of all architectures in the search space. The search space requires us to choose one of 20 possible sizes for each hidden layer in the network; details are discussed in Appendix D.3. We set the maximum inference cost to 42,000 parameters. The search has 1.7× the cost of a stand-alone training run.

**Failure of latency rewards.** Figure 5.6 shows the sampling probabilities from

the search when using the Abs Reward $Q(x) + \beta|T(x)/T_0 - 1|$, and the retrain performance of the found architecture 32-64-96.

In Figures 5.6(a) – 5.6(c), we can see that the sampling probabilities for the different choices are uniform during warmup and then converge quickly. The final selected model (32-64-96) is much worse than the reference model (32-144-24) even though the reference model is actually less expensive. We also observed similar failures for the MnasNet rewards. With the MnasNet rewards, the RL controller also struggles to find a model within ±5% of the constraint despite a grid search of the RL parameters (details in Appendix D.2). In both cases, almost all found models are worse than the reference architecture.

**The RL controller is to blame.** To verify that a low quality SuperNet was not the culprit, we trained a SuperNet without updating the RL controller, and manually inspected the quality of the resulting SuperNet. The sampling probabilities for the RL controller remained uniform throughout the search; the rest of the training setup was kept the same. At the end of the training, we compare two sets of losses on each of the child networks: the validation loss from the SuperNet (*one-shot loss*), and the validation loss from training the child network from scratch. Figure 5.7(a) shows that there is a strong correlation between these accuracies; Figure 5.7(b) shows RL that starts from the sufficiently trained SuperNet weights in 5.7(a) still chooses the suboptimal choice 64. This suggests that the suboptimal search results on Criteo are likely due to issues with the RL controller, rather than issues with the one-shot model weights. In a 3 layer search space we can actually find good models without the RL controller, but in a 5 layer search space, we found an RL controller whose training is interleaved with the SuperNet is important to achieve good results.




(a) SuperNet calibration on Criteo, 3-layer networks

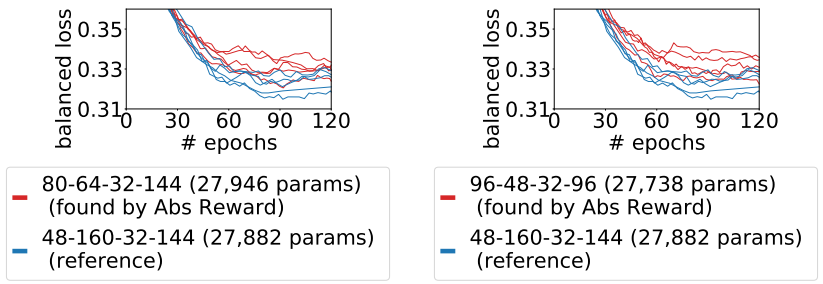
(b) Layer 2 with further delayed RL (final choice 64)

Figure 5.7: SuperNet calibration on Criteo among 3-layer networks (with search space in Appendix D.2), and the Layer 2 change of probabilities in a search with the same number of epochs for only SuperNet training. The y coordinates in Figure (a) are from a SuperNet trained with the same hyperparameters as the search in Figure 5.6, except that there are no RL updates in the first 60 epochs; the x coordinates are from stand-alone training of architectures with performance standard deviation 0.0003, with each errorbar spanning a range of 0.0006. Figure (a) has a 0.96 Pearson correlation coefficient.

**Volkert – 4 Layer Search Space**

We search for 4-layer and 9-layer networks on the Volkert dataset[2]; details are in Appendix D.3. For resource-aware RL rewards, we ran a grid search over the RL learning rate and $\beta$ hyperparameter. The reference architecture for the 4 layer search space is 48-160-32-144 with 27,882 parameters. Despite a hyperparameter grid search, it was difficult to find models with the right target cost reliably using the MnasNet rewards. Using the Abs Reward (Figure 5.8), searched models met the target cost but their quality was suboptimal, and the trend is similar to what has been shown in the toy example (Figure 5.1): a smaller $|\beta|$ gives an infeasible architecture that is beyond the reference number of parameters, and a larger $|\beta|$ gives an architecture that is feasible but suboptimal.

[2]https://www.openml.org/d/41166




(a) $\beta = -10$, RL learning rate $\eta = 0.001$

(b) $\beta = -25$, RL learning rate $\eta = 0.001$

Figure 5.8: On Volkert, the retrain performance of two $Q(x)+\beta|T(x)/T_0-1|$-found architectures versus the 48-160-32-144 reference. Each architecture is trained 5 times with the same setting. The plots of layer-wise sampling probabilities like Figure 5.6(a) – 5.6(c) are omitted for brevity.

**A Common Failure Pattern**

Looking back at the failure modes on Criteo and Volkert, we can see that the reference architectures on which the RL controller with soft constraint fails often have a bottleneck structure. For example, with a 1,027-dimensional input representation, the 32-144-24 reference on Criteo has bottleneck 32; with 180 features, the 48-160-32-144 reference on Volkert has bottleneck 32. More examples can be found in Appendix D.3. As the example in Section 5.3.3 shows, the wide hidden layers around the bottlenecks get penalized harder in the search, and it is thus more difficult for RL with the Abs Reward to find a model that can match the reference performance. Also, Appendix D.2.2 shows the Pareto-optimal architectures in the tradeoff points in Figure 5.3 often have bottleneck structures. This means resource-aware RL rewards in previous NAS practice may be more likely to fail than imagined.

Next in Section 5.4.2, we show the performance of RL with the rejection-based reward in matching these reference architectures.

### 5.4.2 NAS with the rejection-based reward

As introduced in Section 5.3.3, the rejection-based reward does not introduce an additional resource-aware bias in the RL reward, but rather uses conditional probabilities to update the logits in feasible architectures.

To match the 32-144-24 reference on Criteo, we run the search with the MC-sampling-with-rejection approach for 120 epochs, with RL learning rate 0.005 and number of MC samples $N = 3072$.[3]

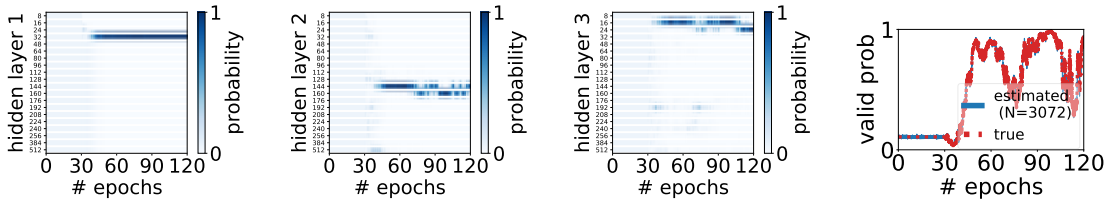
The RL controller converges to two architectures, 32-160-16 (40,769 parameters, with loss 0.4457 ± 0.0002) and 32-144-24 (41,153 parameters, with loss 0.4455 ± 0.0003), after around 50 epochs of NAS, then oscillates between these two solutions (Figure 5.9). At the end of the 120-epoch search, we sample from the layerwise distribution and pick the largest feasible architecture, causing us to select the reference architecture 32-144-24. It is clear that this approach does not get stuck in a local optimum immediately.

On the same hardware, the search takes 3× the time of a stand-alone training in Figure 5.6(d) to finish. As a result, as can be seen in Figure 5.2 the proposed architecture search method is much more efficient than a random baseline.

### 5.4.3 Ablation studies

We do the ablation studies on Criteo with the 32-144-24 reference. The behavior on other datasets with other reference architectures are similar.

[3]The 3-layer search space has $20^3 = 8000$ candidate architectures, which is small enough to compute $\mathbb{P}(V)$ exactly. However, MC can scale to larger spaces which are prohibitively expensive for exhaustive search (Appendix D.3).




(a) Layer 1 (final choice 32)

(b) Layer 2 (final choice 144)

(c) Layer 3 (final choice 24)

(d) Valid probabilities

Figure 5.9: **Success case:** on Criteo in a search space of 3-layer FFNs, Monte-Carlo sampling with rejection eventually finds 32-144-24, the reference architecture, with RL learning rate 0.005 and number of MC samples 3,072. Figure 5.9(d) shows the change of true and estimated valid probabilities.

**Whether to use $\widehat{\mathbb{P}}(V)$ instead of $\mathbb{P}(V)$.** The Monte-Carlo (MC) sampling estimates $\mathbb{P}(V)$ with $\widehat{\mathbb{P}}(V)$ to save resources. Such estimations are especially efficient when the sample space is large. Empirically, the $\widehat{\mathbb{P}}(V)$ estimated with enough MC samples (as described in Appendix D.5) enables the RL controller to find the same architecture as $\mathbb{P}(V)$, because the $\widehat{\mathbb{P}}(V)$ estimated with a large enough number of samples is accurate enough (e.g., Figure 5.5(b) and 5.9(d)).

**Whether to skip infeasible architectures in weight updates.** In each iteration of one-shot training and REINFORCE (Appendix D.1 Algorithm 11) with the rejection mechanism (Appendix D.1 Algorithm 12), we train the weights in the sampled child network $x$ regardless of whether $x$ is feasible. Instead, we may update the weights only when $x$ is feasible, in a similar rejection mechanism as the RL step. We find this mechanism may mislead the search because of insufficiently trained weights: the rejection-based RL controller can still find qualitatively the best architectures on Criteo with the 32-144-24 or 48-240-24-256-8 reference, but fails with the 48-128-16-112 reference. In the latter case, although the RL controller still finds architectures with bottleneck structures (e.g., 32-384-8-144), the first layer sizes of the found architectures are much smaller, leading to suboptimal performance.

**Whether to differentiate through $\widehat{\mathbb{P}}(V)$.** REINFORCE with rejection has the optimization objective:

$$J(y) = \text{stop\_grad}(Q(y) - \overline{Q}) \cdot \log [\mathbb{P}(y)/\mathbb{P}(V)]$$

To update the RL controller's logits, we compute $\nabla J(y)$, which requires a differentiable approximation of $\mathbb{P}(V)$. From a theoretical standpoint, omitting the extra term $\mathbb{P}(V)$ – or using a non-differentiable approximation – will result in biased gradient estimates. Empirically, we ran experiments with multiple variants of our algorithm where we omitted the term $\mathbb{P}(V)$, but found that the quality of the searched architectures was significantly worse.

**Strategy for choosing the final architecture after search.** When RL finishes, instead of biasing towards architectures with more parameters (Appendix D.1 Algorithm 13), we may also bias towards those that are feasible and have larger sampling probabilities. We find that when the final distributions are less deterministic, the architectures found by the latter strategy to perform worse: for example, the top 3 feasible architectures found with the final distribution in Figure 5.9 are 32-128-16, 32-160-16 and 32-128-8, and they are all inferior to 32-144-24.

APPENDIX A

# APPENDIX FOR OBOE

## A.1 Machine learning models

Shown in Table A.1, the hyperparameter names are the same as those in scikit-learn 0.19.2.

## A.2 Dataset meta-features

Dataset meta-features used throughout the experiments are listed in Table A.2 (next page).

## A.3 Meta-feature calculation time

On a number of not very large datasets, the time taken to calculate meta-features in the previous section are already non-negligible, as shown in Figure A.1. Each dot represents one midsize OpenML dataset.

Table A.1: Base Algorithm and Hyperparameter Settings

| Algorithm type | Hyperparameter names (values) |
|---|---|
| Adaboost | n_estimators (50,100), learning_rate (1.0,1.5,2.0,2.5,3) |
| Decision tree | min_samples_split (2,4,8,16,32,64,128,256,512,1024,0.01,0.001,0.0001,1e-05) |
| Extra trees | min_samples_split (2,4,8,16,32,64,128,256,512,1024,0.01,0.001,0.0001,1e-05), criterion (gini,entropy) |
| Gradient boosting | learning_rate (0.001,0.01,0.025,0.05,0.1,0.25,0.5), max_depth (3, 6), max_features (null,log2) |
| Gaussian naive Bayes | - |
| kNN | n_neighbors (1,3,5,7,9,11,13,15), p (1,2) |
| Logistic regression | C (0.25,0.5,0.75,1,1.5,2,3,4), solver (liblinear,saga), penalty (l1,l2) |
| Multilayer perceptron | learning_rate_init (0.0001,0.001,0.01), learning_rate (adaptive), solver (sgd,adam), alpha (0.0001, 0.01) |
| Perceptron | - |
| Random forest | min_samples_split (2,4,8,16,32,64,128,256,512,1024,0.01,0.001,0.0001,1e-05), criterion (gini,entropy) |
| Kernel SVM | C (0.125,0.25,0.5,0.75,1,2,4,8,16), kernel (rbf,poly), coef0 (0,10) |
| Linear SVM | C (0.125,0.25,0.5,0.75,1,2,4,8,16) |

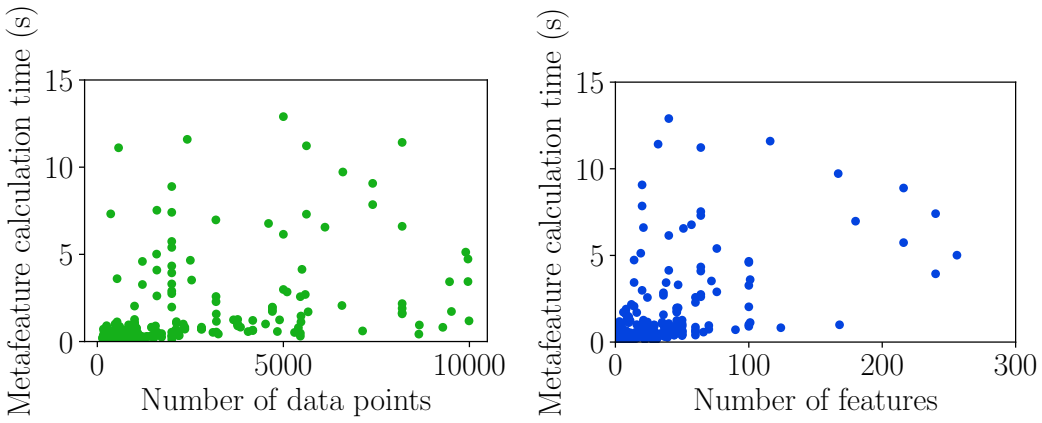



Figure A.1: Meta-feature calculation time and corresponding dataset sizes of the midsize OpenML datasets. The collection of meta-features is the same as that used by auto-sklearn [42]. We can see some calculation times are not negligible.

Table A.2: Dataset Meta-features

| Meta-feature name | Explanation |
|---|---|
| number of instances | number of data points in the dataset |
| log number of instances | the (natural) logarithm of number of instances |
| number of classes | |
| number of features | |
| log number of features | the (natural) logarithm of number of features |
| number of instances with missing values | |
| percentage of instances with missing values | |
| number of features with missing values | |
| percentage of features with missing values | |
| number of missing values | |
| percentage of missing values | |
| number of numeric features | |
| number of categorical features | |
| ratio numerical to nominal | the ratio of number of numerical features to the number of categorical features |
| ratio numerical to nominal | |
| dataset ratio | the ratio of number of features to the number of data points |
| log dataset ratio | the natural logarithm of dataset ratio |
| inverse dataset ratio | |
| log inverse dataset ratio | |
| class probability (min, max, mean, std) | the (min, max, mean, std) of ratios of data points in each class |
| symbols (min, max, mean, std, sum) | the (min, max, mean, std, sum) of the numbers of symbols in all categorical features |
| kurtosis (min, max, mean, std) | |
| skewness (min, max, mean, std) | |
| class entropy | the entropy of the distribution of class labels (logarithm base 2) |
| **landmarking [102] meta-features** | |
| LDA | |
| decision tree | decision tree classifier with 10-fold cross validation |
| decision node learner | 10-fold cross-validated decision tree classifier with `criterion="entropy", max_depth=1, min_samples_split=2, min_samples_leaf=1, max_features=None` |
| random node learner | 10-fold cross-validated decision tree classifier with `max_features=1` and the same above for the rest |
| 1-NN | |
| PCA fraction of components for 95% variance | the fraction of components that account for 95% of variance |
| PCA kurtosis first PC | kurtosis of the dimensionality-reduced data matrix along the first principal component |
| PCA skewness first PC | skewness of the dimensionality-reduced data matrix along the first principal component |

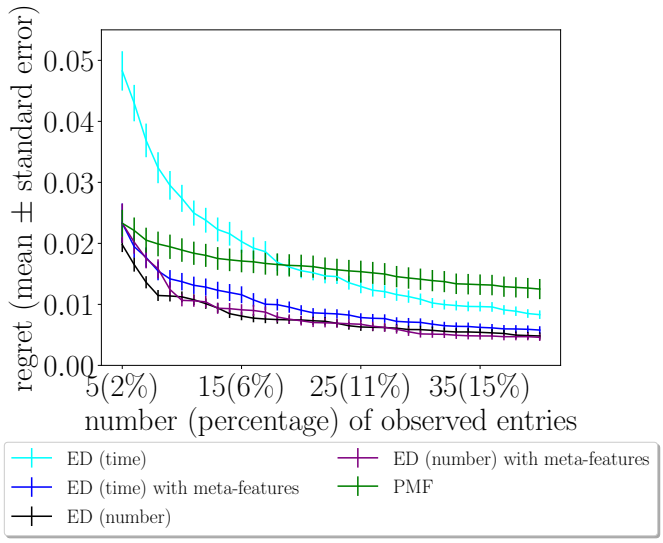



Figure A.2: Comparison of different versions of ED with PMF. "ED (time)" denotes ED with runtime constraint, with time limit set to be 10% of the total runtime of all available models; "ED (number)" denotes ED with the number of entries constrained.

## A.4 Comparison of experiment design with different constraints

In Section 2.5.1, we compared experiment design (ED) with constraint on the number of observed entries. This version is more comparable to QR and PMF than the version with runtime constraint (Equation 2.1). However, the time-constrained version has decent performance, as shown in Figure A.2.

APPENDIX B

# APPENDIX FOR TENSOROBOE

For reproducibility, refer to Section B.1 for datasets and the pipeline search space. All the code is in the GitHub repository at `https://github.com/udellgroup/oboe`.

## B.1 Reproducibility for meta-training

### B.1.1 Meta-training OpenML datasets

Indices of the OpenML datasets we use for meta-training: 2, 3, 5, 7, 9, 11, 12, 13, 14, 15, 16, 18, 20, 22, 23, 24, 25, 27, 28, 29, 30, 31, 35, 36, 37, 38, 39, 40, 41, 42, 44, 46, 48, 50, 53, 54, 59, 60, 181, 182, 183, 187, 285, 307, 313, 316, 329, 336, 337, 338, 375, 377, 389, 446, 450, 458, 463, 469, 475, 694, 715, 717, 718, 720, 721, 723, 725, 728, 730, 732, 733, 735, 737, 740, 742, 743, 744, 745, 746, 747, 748, 749, 750, 751, 753, 763, 769, 773, 776, 778, 779, 788, 792, 794, 796, 797, 799, 803, 805, 806, 807, 813, 818, 819, 820, 824, 825, 826, 830, 832, 837, 838, 847, 853, 855, 863, 866, 869, 870, 871, 873, 877, 880, 884, 888, 896, 900, 903, 904, 906, 907, 908, 909, 910, 911, 912, 913, 915, 917, 920, 923, 925, 926, 933, 934, 935, 936, 937, 941, 943, 952, 953, 954, 955, 958, 962, 970, 971, 973, 976, 978, 979, 980, 983, 987, 991, 994, 995, 996, 997, 1005, 1011, 1012, 1014, 1016, 1020, 1021, 1022, 1025, 1026, 1038, 1039, 1041, 1042, 1048, 1049, 1050, 1054, 1056, 1063, 1065, 1067, 1068, 1069, 1071, 1073, 1100, 1115, 1116, 1121, 4134, 40966, 40971, 40975, 40978, 40979, 40981, 40982, 40983, 40984, 40994, 40997, 41000, 41004, 41005.

### B.1.2 Meta-test UCI datasets

banknote-authentication, blood-transfusion-service-center, breast-cancer-wisconsin-diagnostic, breast-cancer-wisconsin-original, breast-cancer-wisconsin-prognostic, chess-king-rook-vs-king-pawn, cnae-9, congressional-voting-records, connectionist-bench, connectionist-bench-sonar, contraceptive-method-choice, cylinder-bands, haberman-survival, heart-disease-cleveland, heart-disease-hungarian, heart-disease-va, hepatitis, hill-valley, hill-valley-noise, horse-colic, image-segmentation, indian-liver-patient, iris, libras-movement, mammographic-mass, monks-problems-2, ozone-level-detection-eight, ozone-level-detection-one, parkinsons, pen-based-recognition-handwritten-digits, planning-relax, qsar-biodegradation, seeds, seismic-bumps, statlog-project-german-credit, statlog-project-landsat-satellite, thoracic-surgery, thyroid-disease-allbp, thyroid-disease-allhyper, thyroid-disease-allhypo, thyroid-disease-allrep, thyroid-disease-ann-thyroid, thyroid-disease-dis, thyroid-disease-new-thyroid, thyroid-disease-sick, thyroid-disease-sick-euthyroid, thyroid-disease-thyroid-0387, wall-following-robot-navigation-2, wall-following-robot-navigation-24, wall-following-robot-navigation-4.

### B.1.3 Pipeline search space

We build pipelines using scikit-learn [101] primitives. The available components are listed in Table B.1. "null" denotes a pass-through.

## B.2 Experiment design for weighted least squares

When factorizing the error matrix by SVD, we approximate performance of different pipelines to different accuracies. Different accuracies can be characterized

Table B.1: Pipeline search space

| Component | Algorithm type | Hyperparameter names (values) |
|---|---|---|
| Data imputer | Simple imputer | strategy (mean, median, most_frequent, constant) |
| Encoder | null | - |
| | OneHotEncoder | handle_unknown (ignore), sparse (0) |
| Standardizer | null | - |
| | StandardScaler | - |
| Dimensionality reducer | null | - |
| | PCA | n_components (25%, 50%, 75%) |
| | VarianceThreshold | - |
| | SelectKBest | k (25%, 50%, 75%) |
| Estimator | Adaboost | n_estimators (50,100), learning_rate (1.0,1.5,2.0,2.5,3) |
| | Decision tree | min_samples_split (2,4,8,16,32,64,128,256,512,1024,0.01,0.001,1e-4,1e-5) |
| | Extra trees | min_samples_split (2,4,8,16,32,64,128,256,512,1024,0.01,0.001,1e-4,1e-5), criterion (gini,entropy) |
| | Gradient boosting | learning_rate (0.001,0.01,0.025,0.05,0.1,0.25,0.5), max_depth (3, 6), max_features (null,log2) |
| | Gaussian naive Bayes | - |
| | Perceptron | - |
| | kNN | n_neighbors (1,3,5,7,9,11,13,15), p (1,2) |
| | Logistic regression | C (0.25,0.5,0.75,1,1.5,2,3,4), solver (liblinear,saga), penalty (l1,l2) |
| | Multilayer perceptron | learning_rate_init (1e-4,0.001,0.01), learning_rate (adaptive), solver (sgd,adam), alpha (1e-4, 0.01) |
| | Random forest | min_samples_split (2,4,8,16,32,64,128,256,512,1024,0.01,0.001,1e-4,1e-5), criterion (gini,entropy) |
| | Linear SVM | C (0.125,0.25,0.5,0.75,1,2,4,8,16) |

by different variances in the linear regression model, thus the weighted least squares (WLS) model that would theoretically give the best linear unbiased estimate to the new dataset embedding may perform better.

In detail, recall that the constrained $D$-optimal experiment design formulation relies on the assumption that given a low rank matrix multiplication model $X^\top Y = E$, the error term in linear regression $\epsilon \sim \mathcal{N}(0, \sigma^2 I)$, which means each pipeline is predicted to the same accuracy. In the WLS version of our pipeline performance estimation setting, the pipeline performance vector of the new dataset can be written as $e = Y^\top x + \epsilon$, in which $\epsilon \sim \mathcal{N}(0, \Sigma)$. $\Sigma = \text{diag}(\sigma_1^2, \sigma_2^2, \cdots, \sigma_n^2)$

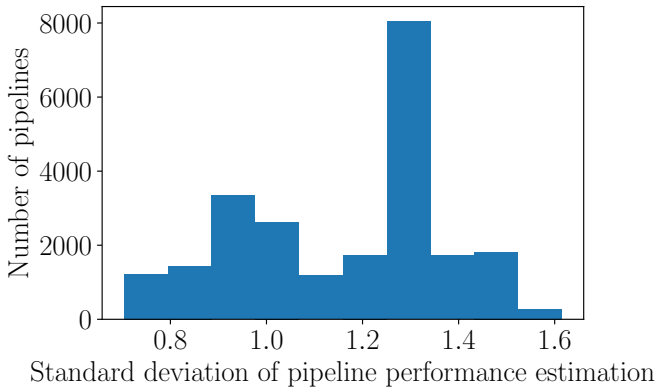



Figure B.1: Standard deviation of prediction accuracy of each pipeline, across meta-training datasets.

is a covariance matrix; diagonal in the weighted least squares setting. For each pipeline $j \in [n]$, we estimate the variance by the sample variance of $e_j - X^\top y_j$, and show a histogram in Figure B.1. In this case, the time-constrained $D$-experiment design problem to solve becomes

$$\begin{aligned} \text{minimize} \quad & \log\det\left(\textstyle\sum_{j=1}^{n} v_j \frac{y_j y_j^\top}{\sigma_j^2}\right)^{-1} \\ \text{subject to} \quad & \sum_{j=1}^{n} v_j \hat{t}_j \le \tau \\ & v_j \in \{0,1\}, \forall\, j \in [n]. \end{aligned} \tag{B.1}$$

The corresponding greedy approach, which we call *weighted-greedy*, is shown as Algorithm 8. It differs from the ordinary greedy approach in that each $y_j$ is scaled by $1/\sigma_j$. Figure B.2 shows its performance compared to convexification and greedy. We can see the weighted-greedy approach performs similarly to the ordinary greedy approach in our experiments.

**Algorithm 8** Greedy algorithm for time-constrained $D$-design in WLS setting, with QR initialization

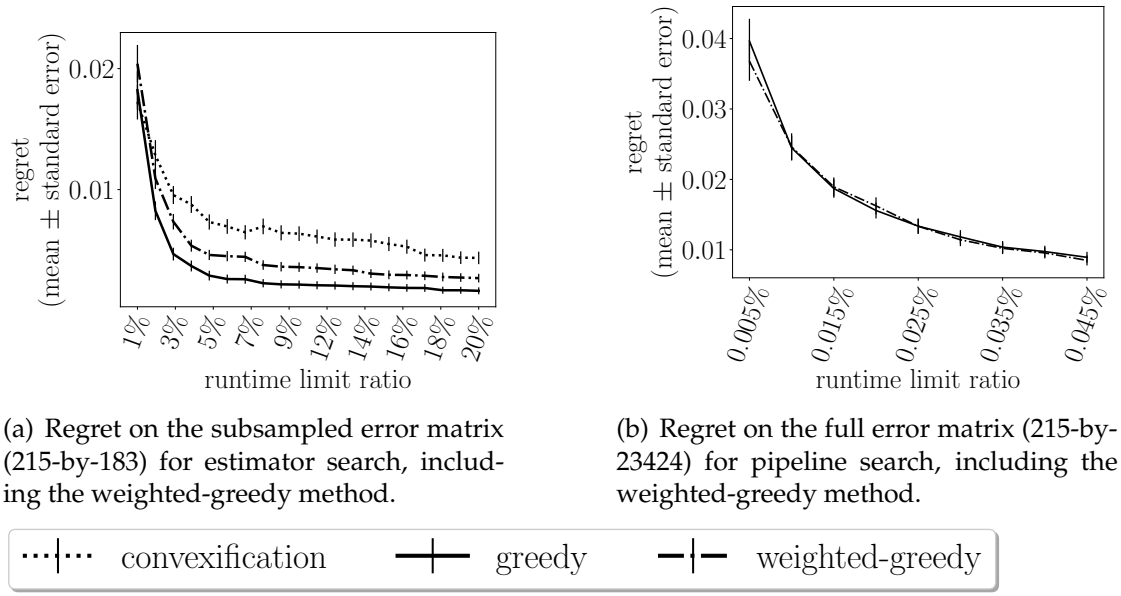
**Input:** design vectors $\{y_j\}_{j=1}^n$, in which $y_j \in \mathbb{R}^k$; pipeline estimation variances $\{\sigma_j^2\}_{j=1}^n$, (predicted) running time of all pipelines $\{\hat{t}_i\}_{i=1}^n$; maximum running time $\tau$

**Output:** The selected set of designs $S \subseteq [n]$

1 $y_j \leftarrow y_j/\sigma_j, \forall j \in [n]$

2 $S_0 \leftarrow$ `QR_initialization`$(\{y_j\}_{j=1}^n, \{\hat{t}_i\}_{i=1}^n, \tau)$

3 $S \leftarrow$ `Greedy_without_repetition`$(\{y_j\}_{j=1}^n, \{\hat{t}_i\}_{i=1}^n, \tau, S_0)$

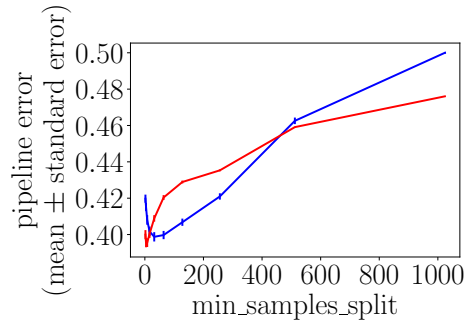



(a) Regret on the subsampled error matrix (215-by-183) for estimator search, including the weighted-greedy method.

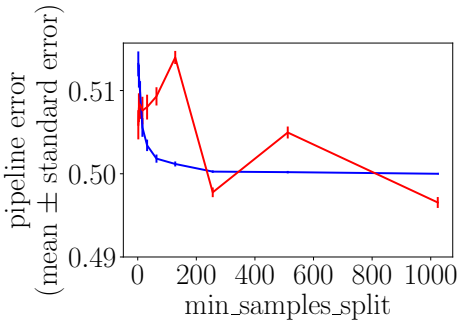
(b) Regret on the full error matrix (215-by-23424) for pipeline search, including the weighted-greedy method.

Figure B.2: Comparison of time-constrained experiment design methods, including the weighted-greedy method.

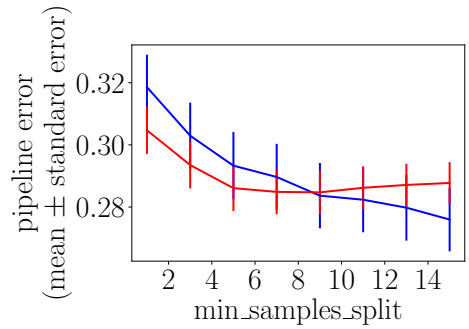
## B.3 Zoomed-in hyperparameter landscapes



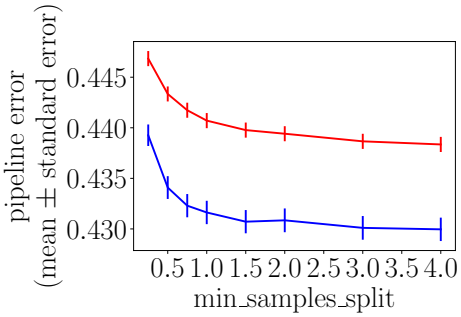

(a) Extra trees on Dataset 23 (1473 points, 10 features)




(b) Decision tree on Dataset 1014 (797 points, 5 features)




(c) kNN on Dataset 799 (1000 points, 6 features)




(d) Logistic regression on Dataset 40971 (1000 data points, 24 features)


Figure B.3: Zoomed-in hyperparameter landscapes in Figure 3.10. The y-axes here do not start from 0.

APPENDIX C

# APPENDIX FOR PEPPP

All the tables in this appendix are shown in the last page.

## C.1 Datasets and low-precision formats

The 87 datasets chosen in our experiment are listed in Table C.1. Note that the datasets contain images of two different resolutions: 32 denotes resolution $3 \times 32 \times 32$, and 64 denotes $3 \times 64 \times 64$. We list our choices of Format A and B in Table C.2 and C.3, respectively. In each evaluation, the low-precision configuration is composed of one Format A (for the activations and weights) and one Format B (for the optimizer), as detailed in Section 4.2.1. Therefore, the total number of low-precision configurations used in our experiment is 99.

## C.2 The SoftImpute algorithm

There are several versions of SoftImpute; [57] gives a nice overview. We use the version in [91].

At a high level, SoftImpute iteratively applies a soft-thresholding operator $\mathcal{S}_\lambda$ on the partially observed error matrix with a series of decreasing $\lambda$ values. Each $\mathcal{S}_\lambda$ replaces the singular values $\{\sigma_i\}$ with $\{(\sigma_i - \lambda)_+\}$. The regularization parameter $\lambda$ can be set in advance or ad hoc, by convergence dynamics.

The pseudocode for the general SoftImpute algorithm is shown as Algorithm 9, in which $P_\Omega(E)$ is a matrix with the same shape as $E$, and has the $(i, j)$-th

entry being $E_{ij}$ if $(i, j) \in \Omega$, and 0 otherwise. In our implementation, $\lambda_i = \lambda t_i$ for each step $i$, and $t_i$ is the step size from TFOCS backtracking [9].

**Algorithm 9** SOFTIMPUTE

**Input:** a partially observed matrix $P_\Omega(E) \in \mathbb{R}^{n \times d}$, number of iterations $I$, a series of decreasing $\lambda$ values $\{\lambda_i\}_{i=1}^I$
**Output:** an estimate $\widehat{E}$

```
for i = 1 to I do
    Ẽ ← P_Ω(E) + P_{Ω^C}(Ê)
    U, Σ, V ← svd(Ẽ)
    Ê ← U S_λ(Σ) V^⊤
end for
```

## C.3 Algorithms for experiment design

As mentioned in Section 4.2.2, there are mainly two algorithms to solve Problem 4.1, the D-optimal experiment design problem: convexification and greedy.

The convexification approach relaxes the combinatorial optimization problem to the convex optimization problem

$$\begin{aligned} \text{minimize} \quad & \log\det\left(\textstyle\sum_{j=1}^d v_j y_j y_j^\top\right)^{-1} \\ \text{subject to} \quad & \sum_{j=1}^d v_j \le l \\ & v_j \in [0,1], \forall j \in [d] \end{aligned} \tag{C.1}$$

that can be solved by a convex solver (like SLSQP). Then we sort the entries in the optimal solution $v^* \in \mathbb{R}^d$ and set the largest $l$ entries to 1 and the rest to 0.

The greedy approach [90, 144] maximizes the submodular objective function by first choosing an initial set of configurations by column-pivoted QR decomposition, and then greedily adding new configuration to the solution set $S$ in each step until $|S| = l$. The greedy stepwise selection algorithm is shown as

Algorithm 10, in which the new configuration is chosen by the Matrix Determinant Lemma[1] [55], and $X_t^{-1}$ is updated by the close form from the Sherman-Morrison Formula[2] [113]. The column-pivoted QR decomposition selects top $k$ pivot columns of $Y \in \mathbb{R}^{k \times d}$ to get the index set $S_0$, ensuring that $X_0 = \sum_{j \in S_0} y_j y_j^\top$ is non-singular.

**Algorithm 10** Greedy algorithm for D-optimal experiment design

**Input:** design vectors $\{y_j\}_{j=1}^d$, in which $y_j \in \mathbb{R}^k$; maximum number of selected configurations $l$; initial set of configurations $S_0 \subseteq [d]$, s.t. $X_0 = \sum_{j \in S_0} y_j y_j^\top$ is non-singular

**Output:** the selected set of designs $S \subseteq [d]$

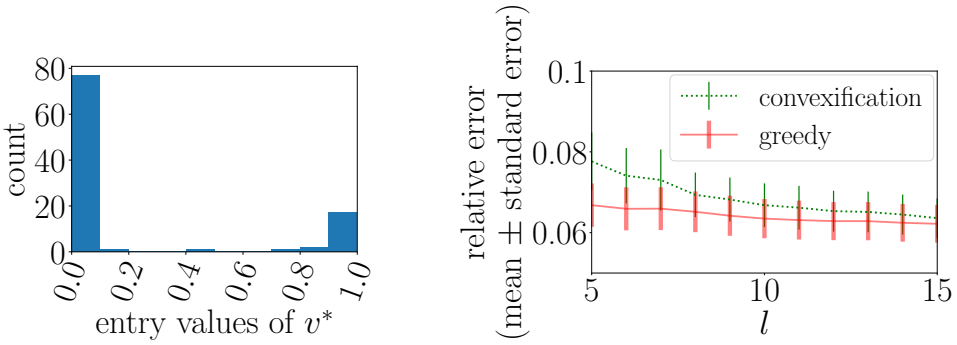
```
S ← S_0
while |S| ≤ l do
    i ← argmax_{j∈[d]\S} y_j^⊤ X_t^{-1} y_j
    S ← S ∪ {i}
    X_{t+1} ← X_t + y_i y_i^⊤
end while
```

The convexification approach empirically works because most entries of the optimal solution $v^*$ are close to either 0 or 1. The histogram in Figure C.1 shows an example when we use rank $k = 5$ to factorize the entire error matrix and set $l = 20$.

In terms of solution quality, the relative error plot in Figure C.1 shows that the greedy approach consistently outperforms in terms of the relative matrix completion error for each dataset in ED-MF solutions. Since the greedy approach is also more than 10× faster than convexification (implemented by `scipy`), we use the greedy approach throughout all experiments for the rest of this work.

[1]The Matrix Determinant Lemma states that for any invertible matrix $A \in \mathbb{R}^{k \times k}$ and $a, b \in \mathbb{R}^k$, $\det(A + ab^\top) = \det(A)(1 + b^\top A^{-1} a)$. Thus $\operatorname{argmax}_{j \in [d] \setminus S} \det(X_t + y_j y_j^\top) = \operatorname{argmax}_{j \in [d] \setminus S} y_j^\top X_t^{-1} y_j$.

[2]The Sherman-Morrison Formula states that for any invertible matrix $A \in \mathbb{R}^{k \times k}$ and $a, b \in \mathbb{R}^k$, $(A + ab^\top)^{-1} = A^{-1} - \frac{A^{-1} a b^\top A^{-1}}{1 + b^\top A^{-1} a}$.




(a) histogram of $v^*$ entries

(b) relative error

Figure C.1: Convexification vs greedy for ED.

## C.4 Information to hardware design

Among all 87 datasets (each has its own error-memory tradeoff), the promising configurations that we identify among all 99 configurations are as below (number of sign bits - number of exponent bits -number of mantissa bits).

- Format A: 1-4-1, Format B: 1-6-7 (appears on 60 out of 87 Pareto frontiers)
- Format A: 1-4-1, Format B: 1-7-7 (appears on 33 out of 87 Pareto frontiers)
- Format A: 1-3-1, Format B: 1-7-7 (appears on 30 out of 87 Pareto frontiers)
- Format A: 1-4-2, Format B: 1-6-7 (appears on 24 out of 87 Pareto frontiers)
- Format A: 1-5-2, Format B: 1-6-7 (appears on 21 out of 87 Pareto frontiers)

## C.5 More details on experiments

We first show the plot of explained variances of top principal components in Figure C.2: how much variance in our data do the first several principal components account for [16]. This quantity is computed by the ratio of sum of squares

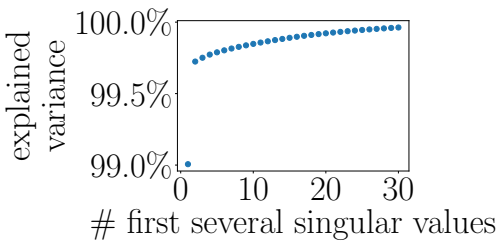
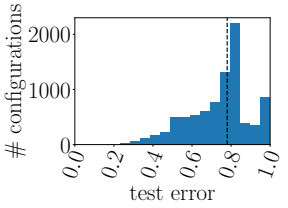



Figure C.2: Explained variance of the first several singular values in Figure 4.4(b).

of the first $k$ singular values to that of all singular values. In Figure C.2, we vary $k$ from 1 to 30, corresponding to the decay of singular values shown in Figure 4.4(b). We can see the first singular value already accounts for 99.0% of the total variance, and the first two singular values account for more than 99.5%. This means we can pick a small rank for PCA in meta-training and still keep the most information in our meta-training data.

We use the ratio of incorrectly classified images as our error metric. Figure C.3 shows histograms of error and memory values in our error and memory matrices from evaluating 99 configurations on 87 datasets. The vertical dashed lines show the respective medians: 0.78 for test error and 816MB for memory. We can see both the test error and memory values span a wide range. The errors come from training a wide range of low-precision configurations with optimization hyperparameters not fine-tuned, and are thus larger than SOTA results. In Section 4.4.3 of the main paper and Section C.5.4 here, we show that training for a larger number of epochs and with some other optimization hyperparameters yield the same error-memory tradeoff as Figure 4.1(a) in the main paper.

In meta-LOOCV settings with a meta-test memory cap at the median memory 816MB, Figure C.4 shows a histogram of number of feasible configurations on each of the 87 datasets. There are "cheap" (resolution 32) datasets on which



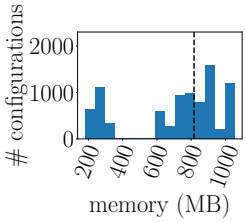




Figure C.3: Histograms of error and memory. The dashed lines are the respective medians.

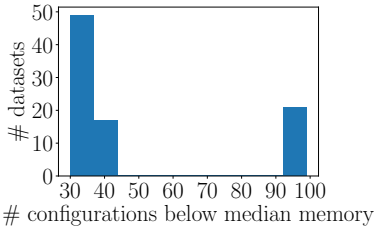
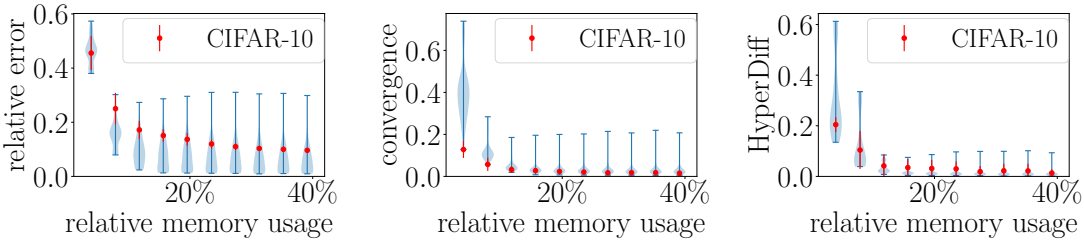



Figure C.4: Histogram of datasets by the number of configurations that take memories less than the overall median of 816MB.

each of the 99 configurations takes less than the cap, and "expensive" (resolution 64) datasets on which the feasible configurations are far less than 99.

### C.5.1 Introduction to RANDOM-MF, QR-MF and BO

- **Random selection with matrix factorization (RANDOM-MF).** Same as ED-MF, RANDOM-MF predicts the unevaluated configurations by linear regression, except that it selects the configurations to evaluate by random sampling.
- **QR decomposition with column pivoting and matrix factorization (QR-MF).** QR-MF first selects the configurations to evaluate by QR decomposition with column pivoting: $EP = QR$, in which the permutation matrix $P$ gives the most informative configurations. Then it predicts unevaluated configurations in the same way as ED-MF and RANDOM-MF.
- **Bayesian optimization (BO)**. Bayesian optimization is a sequential decision making framework that learns and optimizes a black-box function by incrementally building surrogate models and choosing new measurements [45]. It works best for black-box functions that are expensive to eval-




(a) relative error (b) convergence (c) HyperDiff

Figure C.5: Pareto frontier estimation performance in PEPPP meta-training with non-uniform sampling of configurations. The violins and scatters have the same meaning as Figure 4.7. The x axis measures the memory usage relative to exhaustive search.

uate and lack special structures or derivatives. We compare ED-MF with two BO techniques. The first technique, BO-MF, applies BO to the function $f_1 : \mathbb{R}^k \to \mathbb{R}$ that maps low-dimensional configuration embeddings $\{Y_{:,j}\}_{j=1}^d$ to the test errors of the configurations on the meta-test dataset $\{e_j^{\text{new}}\}_{j=1}^d$ [46]. The embeddings come from the same low-rank factorization of the error matrix $E$ as in ED-MF. The second, BO-FULL, applies BO to the function $f_2 : \mathbb{R}^n \to \mathbb{R}$ that directly maps columns of the error matrix $\{E_{:,j}\}_{j=1}^d$ to $\{e_j^{\text{new}}\}_{j=1}^d$. To learn either of these black-box functions, we start by evaluating a subset of configurations $S \subseteq [d]$ and then incrementally select new configurations that maximize the expected improvement [94, 69].

### C.5.2 Additional meta-training results: non-uniform sampling

In non-uniform sampling, we sample each entry of the error matrix $E$ with probabilities $P_{ij} = \sigma(1/W_{ij})$, in which $\sigma$ maps $\{1/W_{ij}\}$ into an interval $[0, p_{\max}] \subseteq [0, 1]$ according to the cumulative distribution function of $\{1/W_{ij}\}$. By varying $p_{\max}$, we change how we aggressively sample the configurations, how different the

sampling probabilities are between large and small memory configurations, and also the percentage of memory needed. In Figure C.5, we vary $p_{\max}$ from 0.1 to 1 and see that the quality of the estimated Pareto frontier improves with more memory.

### C.5.3 Additional meta-LOOCV results

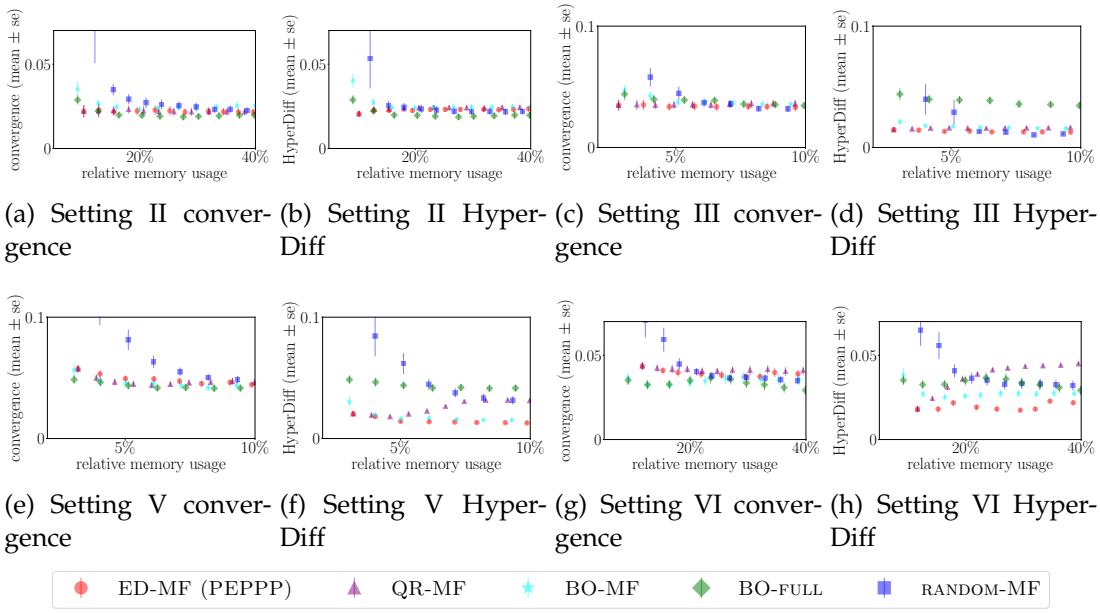
In the main paper, we have shown the performance of Pareto frontier estimates and configuration selection for Setting I and IV. For the rest of the settings in Table 4.1, we conduct meta-LOOCV in the same way and show the results of Pareto frontier estimates in Figure C.6.

**Setting II.** We have the 816MB memory cap for meta-test, and have the full meta-training error matrix in each split.

**Setting III.** We have no memory cap for meta-test, and uniformly sample 20% configurations for meta-training in each split.

**Setting V.** We have no memory cap for meta-test, and non-uniformly sample 20% configurations for meta-training in each split. The sampling method is the same as in meta-training (Figure C.5).

**Setting VI.** We have the 816MB memory cap for meta-test, and non-uniformly sample 20% meta-training configurations in the same way as Setting V above.




(a) Setting II convergence (b) Setting II HyperDiff (c) Setting III convergence (d) Setting III HyperDiff

(e) Setting V convergence (f) Setting V HyperDiff (g) Setting VI convergence (h) Setting VI HyperDiff

Figure C.6: Pareto frontier estimates in meta-LOOCV Setting II (full meta-training error matrix, a 816MB memory cap), Setting III (uniformly sample 20% meta-training measurements, no meta-test memory cap), Setting V (non-uniformly sample 20% meta-training measurements, no meta-test memory cap), and Setting VI (non-uniformly sample 20% meta-training measurements, an 816MB meta-test memory cap). Each error bar is the standard error across datasets. ED-MF is among the best in every setting and under both metrics.

### C.5.4 Tuning optimization hyperparameters

#### Number of epochs

The low-precision networks still underfit after 10 epochs of training. This situation is typical: underfitting due to budget constraints is unfortunately common in deep learning. Luckily, meta-learning the best precision will succeed so long as the validation errors are correctly *ordered*, even if they all overestimate the error of the corresponding fully trained model. Indeed, our validation errors correlate well with the errors achieved after further training: on CIFAR-10, the Kendall tau correlation between ResNet-18 errors at 10 epochs and 100 epochs is 0.73, shown in Figure C.7. The lowest error at 100 epochs is 9.7%, only ap-

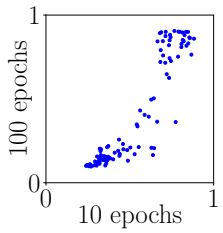
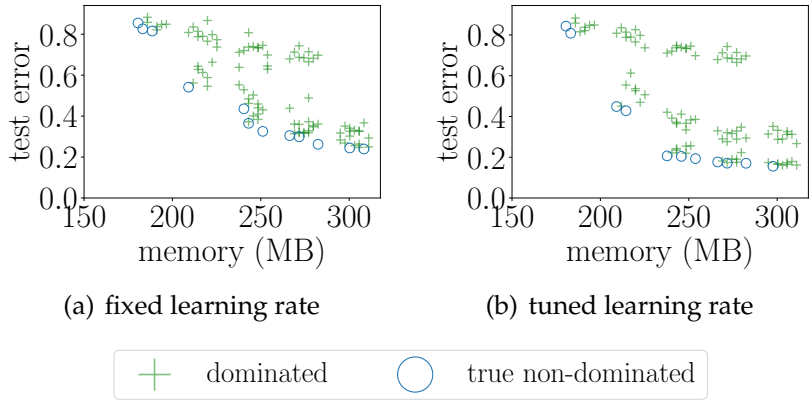



Figure C.7: Errors of 99 configurations trained for different numbers of epochs.




(a) fixed learning rate

(b) tuned learning rate

Figure C.8: CIFAR-10 error-memory tradeoff. Figure (a) has learning rate 0.001 for all low-precision configurations. Figure (b) shows the tradeoff with tuned learning rates: at each low-precision configuration, the lowest test error achieved by learning rates $\{0.01, 0.001, 0.0001\}$ is selected.

proximately 2% higher than SOTA [19].

### Learning rate

Previously, we have shown that PEPPP can estimate the error-memory tradeoff and select a promising configuration with other hyperparameters fixed. In practice, users may also want to tune hyperparameters like learning rate to achieve the lowest error. Here, we tune learning rate in addition to precision, and show that the methodology can be used in broader settings of hyperparameter tuning.

Figure C.8 shows that the error-memory tradeoff still exists with a fine-

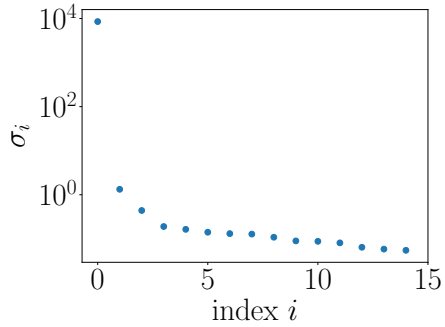
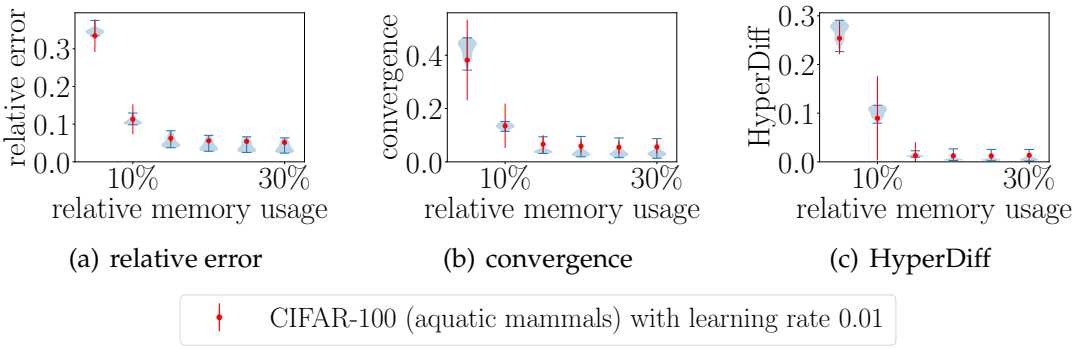



Figure C.9: Singular value decay of the LR-tuned error matrix.

tuned learning rate for each configuration. With the learning rate tuned across $\{0.01, 0.001, 0.0001\}$, the test errors are in general smaller, but high-memory configurations still achieve lower errors in general. Thus the need to efficiently select a low-precision configuration persists.

Our approach can naturally extend to efficiently selecting optimization *and* low-precision hyperparameters. We perform the meta-training and meta-LOOCV experiments on a subset of CIFAR-100 partitions with multiple learning rates $\{0.01, 0.001, 0.0001\}$. The error and memory matrices we use here have 45 rows and 99 columns, respectively. Learning rate and low-precision configuration are collapsed into the same dimension: each row corresponds to a combination of one CIFAR-100 subset and one of the learning rates $\{0.01, 0.001, 0.0001\}$, as shown in Table C.4. We say that these error and memory matrices are *LR-tuned*. Figure C.9 shows the LR-tuned error matrix also has a fast singular value decay. The other hyperparameters are the same as in LR-fixed experiments, except that we use batch size 128 and train for 100 epochs. The meta-training and meta-LOOCV results are consistent with those in Sections 4.4.1 and 4.4.2, respectively:

- In meta-training, we first uniformly sample the error matrix and study




(a) relative error (b) convergence (c) HyperDiff

CIFAR-100 (aquatic mammals) with learning rate 0.01

Figure C.10: The Pareto frontier estimation performance in meta-training, with uniform sampling of configurations on the LR-tuned error and memory matrices. Similar to Figure 4.7, the violins show the distribution of the performance on individual datasets, and the error bars (blue) show the range. The red error bars show the standard deviation of the error on CIFAR-100 aquatic mammals and learning rate 0.01, across 100 random samples of the error matrix. Figure (a) shows the matrix completion error for each dataset; Figure (b) and (c) show the performance of the Pareto frontier estimates in convergence and HyperDiff.

the performance of matrix completion and Pareto frontier estimation. Figure C.10 shows the matrix completion error and Pareto frontier estimation metrics. Then we do non-uniformly sampling and get Figure C.11, the LR-tuned version of Figure C.5 in the main paper.

- In meta-test, we evaluate settings in Table 4.1 in the main paper. We get the performance of Pareto frontier estimates in Figure C.12, the LR-tuned version of Figure 4.9 in the main paper. ED-MF steadily outperforms.

### C.5.5 Learning across architectures

We show that on 10 ImageNet partitions, PEPPP with ED-MF is able to estimate the error-memory tradeoff of the low-precision configurations on of ResNet-34, VGG-11, VGG-13, VGG-16 and VGG-19. The 10 ImageNet partitions have WordNet IDs {n02470899, n01482071, n02022684, n03546340, n07566340,

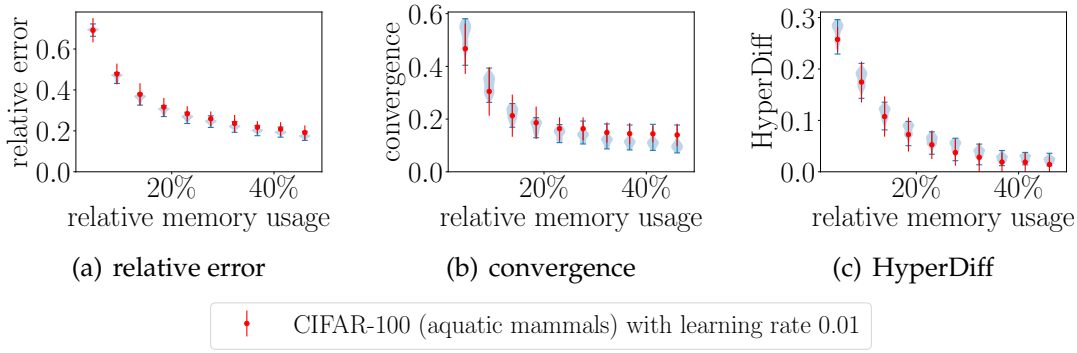



Figure C.11: The Pareto frontier estimation performance in meta-training, with non-uniform sampling of configurations on the LR-tuned error and memory matrices. The violins and scatters have the same meaning as Figure C.5 in the main paper.

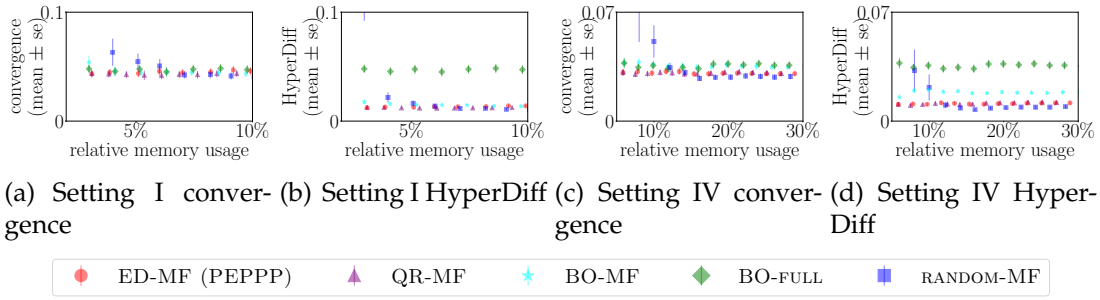



Figure C.12: Pareto frontier estimates in meta-LOOCV settings on the LR-tuned error and memory matrices. Each error bar is the standard error across datasets.

n00019613, n01772222, n03915437, n02489589, n02127808} and are randomly selected from the 50 ImageNet subsets on which we collected the error matrix. On these ImageNet partitions, we use the performance of ResNet-18 as meta-training data, and either the performance of ResNet-18 or VGG variants as meta-test data. In Figure C.13, we can see that ED-MF is steadily among the best in Pareto frontier estimation, and there is no statistical difference between the estimation performance on ResNet-34 and VGG variants.

Next, we compare the performance of the following two cases:

i. Meta-learning across datasets with performance from the same architec-

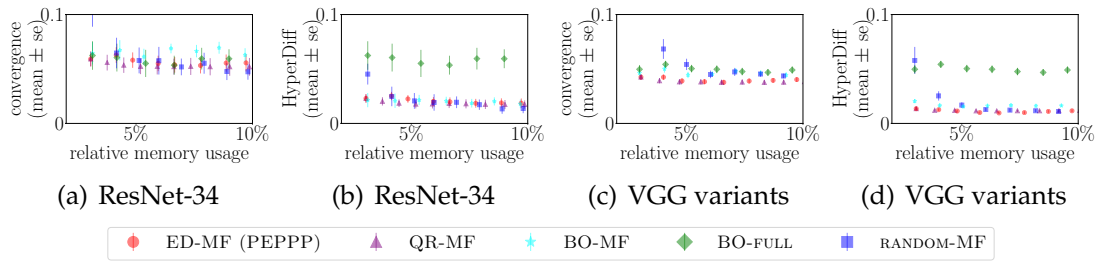



Figure C.13: Pareto frontier estimates in meta-LOOCV Setting I when learning across architectures: from ResNet-18 to either ResNet-34, or to VGG variants. Each error bar is the standard error across datasets. The x axis measures the memory usage relative to exhaustively searching the permissible configurations. ED-MF consistently picks the configurations that give the best PF estimates.

ture: For example, to learn the error-memory tradeoff of ResNet-18 on n02470899, we only use the tradeoffs of ResNet-18 on 9 other ImageNet partitions as the meta-training data.

ii. Meta-learning across datasets with performance from both the same and other architectures: For example, the error-memory tradeoff of ResNet-18 on n02470899, we not only use the tradeoffs of ResNet-18 on 9 other ImageNet partitions, but also use those of ResNet-34, VGG-11, VGG-13, VGG-16 and VGG-19 on the 9 partitions as the meta-training data.

Figure C.14 shows that Case ii outperforms Case i in better estimating the error-memory tradeoffs on different architectures and datasets.

Table C.1: Datasets

| index | dataset name | resolution | # points |
|---|---|---|---|
| 1 | CIFAR10 | 32 | 60000 |
| 2 | CIFAR100 (aquatic mammals) | 32 | 3000 |
| 3 | CIFAR100 (fish) | 32 | 3000 |
| 4 | CIFAR100 (flowers) | 32 | 3000 |
| 5 | CIFAR100 (food containers) | 32 | 3000 |
| 6 | CIFAR100 (fruit and vegetables) | 32 | 3000 |
| 7 | CIFAR100 (household electrical devices) | 32 | 3000 |
| 8 | CIFAR100 (household furniture) | 32 | 3000 |
| 9 | CIFAR100 (insects) | 32 | 3000 |
| 10 | CIFAR100 (large carnivores) | 32 | 3000 |
| 11 | CIFAR100 (large man-made outdoor things) | 32 | 3000 |
| 12 | CIFAR100 (large natural outdoor scenes) | 32 | 3000 |
| 13 | CIFAR100 (large omnivores and herbivores) | 32 | 3000 |
| 14 | CIFAR100 (medium-sized mammals) | 32 | 3000 |
| 15 | CIFAR100 (non-insect invertebrates) | 32 | 3000 |
| 16 | CIFAR100 (people) | 32 | 3000 |
| 17 | CIFAR100 (reptiles) | 32 | 3000 |
| 18 | CIFAR100 (small mammals) | 32 | 3000 |
| 19 | CIFAR100 (trees) | 32 | 3000 |
| 20 | CIFAR100 (vehicles 1) | 32 | 3000 |
| 21 | CIFAR100 (vehicles 2) | 32 | 3000 |
| 22 | aircraft | 64 | 6667 |
| 23 | cub | 64 | 10649 |
| 24 | dtd | 64 | 3760 |
| 25 | isic | 64 | 22802 |
| 26 | merced | 64 | 1890 |
| 27 | scenes | 64 | 14088 |
| 28 | ucf101 | 64 | 12024 |
| 29 | daimlerpedcls | 64 | 29400 |
| 30 | gtsrb | 64 | 26640 |
| 31 | kather | 64 | 4000 |
| 32 | omniglot | 64 | 25968 |
| 33 | svhn | 64 | 73257 |
| 34 | vgg-flowers | 64 | 2040 |
| 35 | bach | 64 | 320 |
| 36 | protein atlas | 64 | 12113 |
| 37 | minc | 64 | 51750 |
| 38 | ImageNet (bag) | 64 | 6519 |
| 39 | ImageNet (retriever) | 64 | 6668 |
| 40 | ImageNet (domestic cat) | 64 | 6750 |
| 41 | ImageNet (stick) | 64 | 6750 |
| 42 | ImageNet (turtle) | 64 | 6750 |
| 43 | ImageNet (finch) | 64 | 6750 |
| 44 | ImageNet (watchdog) | 64 | 6404 |
| 45 | ImageNet (footwear) | 64 | 6587 |
| 46 | ImageNet (salamander) | 64 | 6750 |
| 47 | ImageNet (anthropoid ape) | 64 | 6750 |
| 48 | ImageNet (elasmobranch) | 64 | 6750 |
| 49 | ImageNet (shorebird) | 64 | 6750 |
| 50 | ImageNet (housing) | 64 | 6605 |
| 51 | ImageNet (foodstuff) | 64 | 6750 |
| 52 | ImageNet (substance) | 64 | 6643 |
| 53 | ImageNet (spider) | 64 | 8100 |
| 54 | ImageNet (percussion instrument) | 64 | 8082 |
| 55 | ImageNet (New World monkey) | 64 | 8100 |
| 56 | ImageNet (big cat) | 64 | 8100 |
| 57 | ImageNet (box) | 64 | 7829 |
| 58 | ImageNet (fabric) | 64 | 7862 |
| 59 | ImageNet (kitchen appliance) | 64 | 7773 |
| 60 | ImageNet (mollusk) | 64 | 8100 |
| 61 | ImageNet (hand tool) | 64 | 8054 |
| 62 | ImageNet (butterfly) | 64 | 8100 |
| 63 | ImageNet (stringed instrument) | 64 | 8100 |
| 64 | ImageNet (boat) | 64 | 8006 |
| 65 | ImageNet (rodent) | 64 | 8006 |
| 66 | ImageNet (toiletry) | 64 | 7522 |
| 67 | ImageNet (computer) | 64 | 7696 |
| 68 | ImageNet (shop) | 64 | 9400 |
| 69 | ImageNet (musteline mammal) | 64 | 9450 |
| 70 | ImageNet (Old World monkey) | 64 | 9450 |
| 71 | ImageNet (bottle) | 64 | 9205 |
| 72 | ImageNet (fungus) | 64 | 9450 |
| 73 | ImageNet (truck) | 64 | 9309 |
| 74 | ImageNet (spaniel) | 64 | 9119 |
| 75 | ImageNet (sports equipment) | 64 | 9450 |
| 76 | ImageNet (game bird) | 64 | 9450 |
| 77 | ImageNet (seat) | 64 | 9126 |
| 78 | ImageNet (fruit) | 64 | 9450 |
| 79 | ImageNet (weapon) | 64 | 9450 |
| 80 | ImageNet (beetle) | 64 | 10800 |
| 81 | ImageNet (toy dog) | 64 | 9832 |
| 82 | ImageNet (decapod crustacean) | 64 | 10800 |
| 83 | ImageNet (fastener) | 64 | 10675 |
| 84 | ImageNet (timepiece) | 64 | 10164 |
| 85 | ImageNet (dish) | 64 | 10556 |
| 86 | ImageNet (mechanical device) | 64 | 10617 |
| 87 | ImageNet (colubrid snake) | 64 | 12150 |

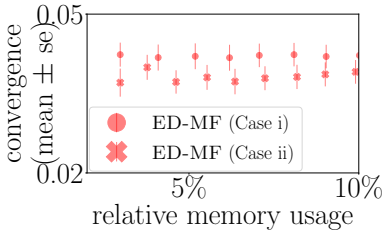



Figure C.14: Benefit of meta-learning across architectures. Each error bar is the standard error across architecture-dataset combinations (e.g., ResNet-18 + n02470899 is a combination). The x axis measures the memory usage relative to exhaustively searching the permissible configurations.

Table C.2: Format A
(for activations and weights)

| index | # exponent bits | # mantissa Bits | total bit width |
|---|---|---|---|
| 1 | 3 | 1 | 5 |
| 2 | 3 | 2 | 6 |
| 3 | 3 | 3 | 7 |
| 4 | 3 | 4 | 8 |
| 5 | 4 | 1 | 6 |
| 6 | 4 | 2 | 7 |
| 7 | 4 | 3 | 8 |
| 8 | 4 | 4 | 9 |
| 9 | 5 | 1 | 7 |
| 10 | 5 | 2 | 8 |
| 11 | 5 | 3 | 9 |

Table C.3: Format B (for optimizer)

| index | # exponent bits | # mantissa Bits | total bit width |
|---|---|---|---|
| 1 | 6 | 7 | 14 |
| 2 | 6 | 9 | 16 |
| 3 | 6 | 11 | 18 |
| 4 | 7 | 7 | 15 |
| 5 | 7 | 9 | 17 |
| 6 | 7 | 11 | 19 |
| 7 | 8 | 7 | 16 |
| 8 | 8 | 9 | 18 |
| 9 | 8 | 11 | 20 |

Table C.4: Datasets and learning rates in Section 4.4.3

| index | dataset name | learning rate |
|---|---|---|
| 1 | CIFAR100 (aquatic mammals) | 0.01 |
| 2 | CIFAR100 (fish) | 0.01 |
| 3 | CIFAR100 (flowers) | 0.01 |
| 4 | CIFAR100 (food containers) | 0.01 |
| 5 | CIFAR100 (fruit and vegetables) | 0.01 |
| 6 | CIFAR100 (household electrical devices) | 0.01 |
| 7 | CIFAR100 (household furniture) | 0.01 |
| 8 | CIFAR100 (insects) | 0.01 |
| 9 | CIFAR100 (large carnivores) | 0.01 |
| 10 | CIFAR100 (large man-made outdoor things) | 0.01 |
| 11 | CIFAR100 (large natural outdoor scenes) | 0.01 |
| 12 | CIFAR100 (large omnivores and herbivores) | 0.01 |
| 13 | CIFAR100 (medium-sized mammals) | 0.01 |
| 14 | CIFAR100 (non-insect invertebrates) | 0.01 |
| 15 | CIFAR100 (people) | 0.01 |
| 16 | CIFAR100 (reptiles) | 0.01 |
| 17 | CIFAR100 (aquatic mammals) | 0.001 |
| 18 | CIFAR100 (fish) | 0.001 |
| 19 | CIFAR100 (flowers) | 0.001 |
| 20 | CIFAR100 (food containers) | 0.001 |
| 21 | CIFAR100 (fruit and vegetables) | 0.001 |
| 22 | CIFAR100 (household electrical devices) | 0.001 |
| 23 | CIFAR100 (household furniture) | 0.001 |
| 24 | CIFAR100 (insects) | 0.001 |
| 25 | CIFAR100 (large carnivores) | 0.001 |
| 26 | CIFAR100 (large man-made outdoor things) | 0.001 |
| 27 | CIFAR100 (large natural outdoor scenes) | 0.001 |
| 28 | CIFAR100 (large omnivores and herbivores) | 0.001 |
| 29 | CIFAR100 (medium-sized mammals) | 0.001 |
| 30 | CIFAR100 (non-insect invertebrates) | 0.001 |
| 31 | CIFAR100 (people) | 0.001 |
| 32 | CIFAR100 (reptiles) | 0.001 |
| 33 | CIFAR100 (aquatic mammals) | 0.0001 |
| 34 | CIFAR100 (fish) | 0.0001 |
| 35 | CIFAR100 (flowers) | 0.0001 |
| 36 | CIFAR100 (food containers) | 0.0001 |
| 37 | CIFAR100 (fruit and vegetables) | 0.0001 |
| 38 | CIFAR100 (household electrical devices) | 0.0001 |
| 39 | CIFAR100 (household furniture) | 0.0001 |
| 40 | CIFAR100 (insects) | 0.0001 |
| 41 | CIFAR100 (large carnivores) | 0.0001 |
| 42 | CIFAR100 (large man-made outdoor things) | 0.0001 |
| 43 | CIFAR100 (large natural outdoor scenes) | 0.0001 |
| 44 | CIFAR100 (large omnivores and herbivores) | 0.0001 |
| 45 | CIFAR100 (medium-sized mammals) | 0.0001 |

APPENDIX D

# APPENDIX FOR TABNAS

## D.1 Algorithm pseudocode

We show psuedocode of the algorithms introduced in Section 4.2.

**Algorithm 11** (Resource-Oblivious) One-Shot Training and REINFORCE

**Input:** search space $S$, weight learning rate $\alpha$, RL learning rate $\eta$
**Output:** sampling probabilities $\{p_{ij}\}_{i\in[L],j\in[C_i]}$

initialize logits $\ell_{ij} \leftarrow 0, \forall i \in [L], j \in [C_i]$
initialize quality reward moving average $\bar{Q} \leftarrow 0$
layer warmup
**for** iter = 1 **to** max_iter **do**
  $p_{ij} \leftarrow \exp(\ell_{ij}) / \sum_{j\in[C_i]} \exp(\ell_{ij}), \forall i \in [L], j \in [C_i]$
  ▷ weight update
  **for** $i = 1$ to $L$ **do**
    $x_i \leftarrow$ the $i$-th layer size sampled from $\{s_{ij}\}_{j\in[C_i]}$ with distribution $\{p_{ij}\}_{j\in[C_i]}$
  **end for**
  $loss(x) \leftarrow$ the (training) loss of $x = x_1\text{-}\cdots\text{-}x_L$ on the training set
  $w \leftarrow w - \alpha\nabla loss(x)$, in which $w$ is the weights of $x$ ▷ can be replaced with optimizers other than SGD
  ▷ RL update
  **for** $i = 1$ to $L$ **do**
    $y_i \leftarrow$ the $i$-th layer size sampled from $\{s_{ij}\}_{j\in[C_i]}$ with distribution $\{p_{ij}\}_{j\in[C_i]}$
  **end for**
  $Q(y) \leftarrow 1 - loss(y)$, the quality reward of $y = y_1\text{-}\cdots\text{-}y_L$ on the validation set
  RL reward $r(y) \leftarrow Q(y)$ ▷ can be replaced with resource-aware rewards introduced in Section 5.3.3
  $J(y) \leftarrow \text{stop\_grad}(r(y) - \bar{Q}) \log \mathbb{P}(y)$ ▷ can be replaced with Algorithm 12 when resource-constrained
  $\ell_{ij} \leftarrow \ell_{ij} + \eta\nabla J(y), \forall i \in [L], j \in [C_i]$ ▷ can be replaced with optimizers other than SGD
  $\bar{Q} \leftarrow \frac{\gamma*\bar{Q}+(1-\gamma)*Q(y)}{\gamma*\bar{Q}+1-\gamma}$ ▷ update moving average with $\gamma = 0.9$
**end for**

**Algorithm 12** Rejection with Monte-Carlo (MC) Sampling

**Input:** number of MC samples $N$, feasible set $V$, MC proposal distribution $q$, quality reward moving average $\bar{Q}$, sampled architecture for RL in the current step $y = y_1\text{-}y_2\text{-}\cdots\text{-}y_L$, current layer size distribution over $\{s_{ij}\}_{j\in[C_i]}$ with probability $\{p_{ij}\}_{j\in[C_i]}$
**Output:** $J(y)$
**if** $y$ is feasible **then**
    $Q(y)$ = the quality reward of $y$
    $\mathbb{P}(y) := \prod_{i\in[L]} \mathbb{P}(Y_i = y_i)$
    **for** $i = 1$ to $L$ **do**
        $\{z_i^{(k)}\}_{k\in[N]} \leftarrow N$ samples of the $i$-th layer size, sampled from $\{s_{ij}\}_{j\in[C_i]}$ with distribution $\{p_{ij}\}_{j\in[C_i]}$
    **end for**
    $p_i^{(k)} := \mathbb{P}(Z_i = z_i^{(k)}), \forall i \in [L], k \in [N]$
    $p^{(k)} := \prod_{i\in[L]} p_i^{(k)}, \forall k \in [N]$
    $\widehat{\mathbb{P}}(V) \leftarrow \frac{1}{N} \sum_{k\in[N], z^{(k)}\in V} \frac{p^{(k)}}{q^{(k)}}$, in which $z^{(k)} := z_1^{(k)}\text{-}\cdots\text{-}z_L^{(k)}$
    $J(y) \leftarrow \text{stop\_grad}(Q(y) - \bar{Q}) \log \frac{\mathbb{P}(y)}{\widehat{\mathbb{P}}(V)}$
**else**
    $J(y) \leftarrow 0$
**end if**

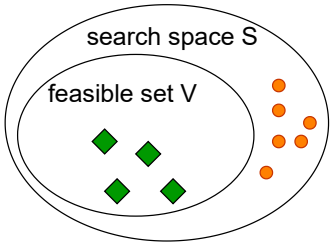



Figure D.1: Illustration of the feasible set $V$ within the search space $S$. Each green diamond or orange dot denotes a feasible or infeasible architecture, respectively.

**Algorithm 13** Sample to Return the Final Architecture

**Input:** sampling probabilities $\{p_{ij}\}_{i\in[L],j\in[C_i]}$ returned by Algorithm 11, number of desired architectures $n$, number of samples to draw $m$
**Output:** the set of $n$ selected architectures $A$
**for** $i = 1$ to $L$ **do**
    $\{x_i^{(k)}\}_{k\in[m]} \leftarrow m$ samples of the $i$-th layer size, sampled from $\{s_{ij}\}_{j\in[C_i]}$ with distribution $\{p_{ij}\}_{j\in[C_i]}$
**end for**
$F := \{k \in [m] \mid x_1^{(k)}\text{-}x_2^{(k)}\text{-}\cdots\text{-}x_L^{(k)} \in V\}$
$A \leftarrow n$ unique architectures in $F$ with largest numbers of parameters

Notice that in Algorithm 11, we show the weight and RL updates with the stochastic gradient descent (SGD) algorithm; in our experiments on the toy example and real datasets, we use Adam for both updates as in ProxylessNAS [20] and TuNAS [11], since it synchronizes convergence across different layer size choices, and slows down the learning which would otherwise converge too rapidly.

## D.2 Details of experiment setup

### D.2.1 Toy example

We use the Adam optimizer with $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 0.001$ to update the logits. When we use the Abs Reward, the results are similar when $\eta \geq 0.05$, while the RL controller with $\eta < 0.05$ converges too slow or is hard to converge. When we use the rejection-based reward, we use RL learning rate $\eta = 0.1$; other $\eta$ values with which RL converges give similar results.

### D.2.2 Real datasets

Table D.1 shows the datasets we use. Datasets other than Criteo come from the OpenML dataset repository [130]. For Criteo, we randomly split the labeled part (45,840,617 points) into 90% training (41,258,185 points) and 10% validation (4,582,432 points); for the other datasets, we randomly split into 80% training and 20% validation[1]. The representations we use for Criteo are inspired by DCN-V2 [134].

Table D.1: Dataset details

| name | # points | # features | | # classes | embedding we use for each feature |
|---|---|---|---|---|---|
| | | numerical | categorical | | |
| Criteo | 51,882,752 | 13 | 26 | 2 | original values for each numerical, 39-dimensional for each categorical |
| Volkert | 58,310 | 180 | 0 | 10 | original values |
| Aloi | 108,000 | 128 | 0 | 1,000 | original values |
| Connect-4 | 67,557 | 0 | 42 | 3 | 2-dimensional for each categorical |
| Higgs | 98,050 | 28 | 0 | 2 | original values |

Table D.2 shows the hyperparameters we use for stand-alone training and NAS, found by grid search. With these hyperparameters, the best architecture in each of our search spaces (introduced in Appendix D.2.2) has performance that is within ±5% of the best performance in [70] Table 2, and we achieve these scores with FFNs that only have 5% parameters of the ones there. The Adam optimizer has hyperparameters $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 0.001$. We use layer normalization [5] for all datasets. We use balanced error (weighted average of classification errors across classes) for all other datasets[2] as in [70], except for

[1]The ranking of validation losses among architectures under such splits is almost the same as that of test losses under 60%-20%-20% training-validation-test splits.

[2]The performance ranking of architectures under the balanced error metric is almost the same as under logistic loss. Also, the balanced error metric is only for reporting the final vali-

Criteo, on which we use logistic loss as in [134].

Table D.2: Weight training hyperparameter details

| name | batch size | learning rate | learning rate schedule | optimizer | # training epochs | metric |
|---|---|---|---|---|---|---|
| Criteo | 512 | 0.002 | cosine decay | Adam | 60 | log loss |
| Volkert | 32 | 0.01 | constant | SGD with momentum 0.9 | 120 | balanced error |
| Aloi | 128 | 0.0005 | constant | Adam | 50 | balanced error |
| Connect-4 | 32 | 0.0005 | cosine decay | Adam | 60 | balanced error |
| Higgs | 64 | 0.05 | constant | SGD | 60 | balanced error |

We use constant RL learning rates for NAS. The Connect-4[3] and Higgs[4] datasets are easy for both the Abs Reward and rejection-based reward, in the sense that small FFNs with fewer than 5,000 parameters can achieve near-SOTA results (±5% of the best accuracy scores listed in [70] Table 2, except that we do 80%-20% training-validation splits and use original instead of standardized features), and RL-based weight-sharing NAS with either reward can find architectures that match the Pareto-optimal reference architectures. The Aloi dataset[5] needs more parameters (more than 100k), but the other observations are similar to on Connect-4 and Higgs. Thus we omit the corresponding results.

The factorized search spaces we use for NAS are:

- Criteo: Each layer has 20 choices {8, 16, 24, 32, 48, 64, 80, 96, 112, 128, 144, 160, 176, 192, 208, 224, 240, 256, 384, 512}.
- Volkert, 4-layer networks: Each layer has 20 choices {8, 16, 24, 32, 48, 64, 80, 96, 112, 128, 144, 160, 176, 192, 208, 224, 240, 256, 384, 512}.

dation losses; both weight and RL updates use logistic loss.

[3]https://www.openml.org/d/40668

[4]https://www.openml.org/d/23512

[5]https://www.openml.org/d/42396

- Volkert, 9-layer networks: Each layer has 12 choices {8, 16, 24, 32, 48, 64, 80, 96, 112, 128, 144, 160}. This search space has fewer choices for each hidden layer than the 4-layer counterpart, but the size of the search space is over $3 \times 10^4$ times larger.

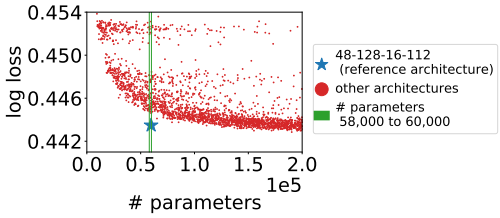
**More Details on the Tradeoff Plot (Figure 5.3)**

Each search space we use for exhaustive search and NAS has a fixed number of hidden layers. Resource-constrained NAS in a search space with varying number of hidden layers is an interesting problem for future studies. On each dataset, we randomly sample, train and evaluate architectures in the search space with the number of parameters fall within a range, in which there is a clear tradeoff between loss and number of parameters. These ranges are:

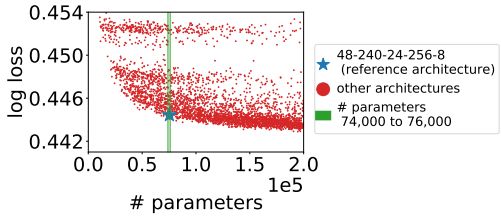
- Criteo: 0 – 200,000
- Volkert, 4-layer networks: 15,000 – 50,000
- Volkert, 9-layer networks: 40,000 – 100,000

Figure D.2 shows the tradeoffs between loss and number of parameters in these search spaces. When training each architecture 5 times, the standard deviation (std) across different runs is 0.0002 for Criteo and 0.004 for Volkert, meaning that the architectures whose performance difference is larger than 2× std are qualitatively different with high probability. We use Pareto-optimal architectures as the reference of resource-constrained NAS: we want an architecture that both matches (or even beats[6]) the performance of the reference architecture and

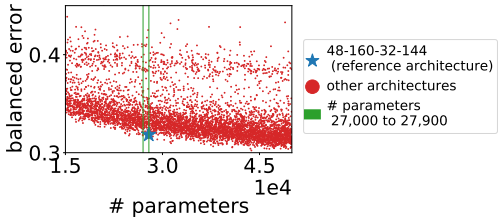
[6]Note that the Pareto optimality of the reference architecture is determined by only one round of random search. Thus because of the randomness across multiple training runs, the other architectures are likely to beat the reference architecture: a "regression toward the mean".



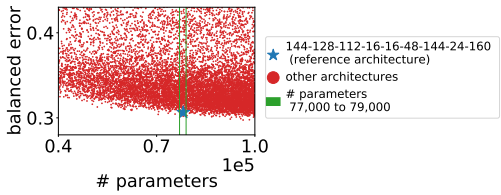

(a) Tradeoff on Criteo, in 4 layer search space




(b) Tradeoff on Criteo, in 5 layer search space




(c) Tradeoff on Volkert, in 4 layer search space




(d) Tradeoff on Volkert, in 9 layer search space

Figure D.2: Tradeoffs between validation loss and number of parameters in four search spaces.

has no more parameters than the reference. Most Pareto-optimal architectures in Figure D.2 have the bottleneck structure; Table D.3 shows some examples.

### More Details on TPU Implementation

When we run one-shot NAS on a TPU that has multiple TPU cores (for example, each Cloud TPU-v2 we use has 8 cores), each core samples an architectures independently, and we use the average loss and reward for weight and RL updates, respectively. This means our algorithm actually samples multiple architectures in each iteration and uses the `tensorflow.tpu.cross_replica_sum()` method to compute their average effect on the gradient. Since only a fraction of architectures are feasible in each search space, we set the losses and rewards given by the infeasible architectures to 0 before averaging, so that we are equivalently only averaging across the sampled architectures that are fea-

Table D.3: Some Pareto-optimal architectures in Figure D.2. All architectures shown here and almost all other Pareto-optimal architectures have the bottleneck structure.

| | search space | Pareto-optimal architecture | number of parameters | loss |
|---|---|---|---|---|
| Figure D.2(a) | Criteo 4-layer | 32-144-24-112 | 44,041 | 0.4454 |
| Figure D.2(a) | Criteo 4-layer | 48-112-8-80 | 56,537 | 0.4448 |
| Figure D.2(a) | Criteo 4-layer | 48-384-16-176 | 77,489 | 0.4441 |
| Figure D.2(a) | Criteo 4-layer | 96-144-32-240 | 125,457 | 0.4433 |
| Figure D.2(a) | Criteo 4-layer | 96-384-48-16 | 155,217 | 0.4430 |
| Figure D.2(b) | Criteo 5-layer | 32-240-16-8-96 | 45,769 | 0.4451 |
| Figure D.2(b) | Criteo 5-layer | 48-128-64-16-128 | 67,217 | 0.4446 |
| Figure D.2(b) | Criteo 5-layer | 48-256-16-8-384 | 69,977 | 0.4443 |
| Figure D.2(b) | Criteo 5-layer | 64-144-48-96-160 | 102,497 | 0.4437 |
| Figure D.2(b) | Criteo 5-layer | 96-512-24-256-48 | 179,449 | 0.4430 |
| Figure D.2(c) | Volkert 4-layer | 48-112-16-24 | 16,642 | 0.3314 |
| Figure D.2(c) | Volkert 4-layer | 32-112-24-224 | 20,050 | 0.3269 |
| Figure D.2(c) | Volkert 4-layer | 48-160-32-144 | 27,882 | 0.3149 |
| Figure D.2(c) | Volkert 4-layer | 48-256-24-112 | 31,330 | 0.3097 |
| Figure D.2(c) | Volkert 4-layer | 80-208-32-64 | 40,778 | 0.3054 |
| Figure D.2(d) | Volkert 9-layer | 64-64-160-48-16-144-16-8-48 | 40,482 | 0.3250 |
| Figure D.2(d) | Volkert 9-layer | 80-144-32-112-32-8-128-8-144 | 43,290 | 0.3238 |
| Figure D.2(d) | Volkert 9-layer | 112-144-32-32-24-24-24-128-32 | 51,890 | 0.3128 |
| Figure D.2(d) | Volkert 9-layer | 144-128-112-16-16-48-144-24-160 | 78,114 | 0.3019 |
| Figure D.2(d) | Volkert 9-layer | 160-144-144-32-112-32-48-32-144 | 94,330 | 0.3010 |

sible. We then reweight the average loss or reward with number_of_cores / number_of_feasible_architectures to obtain an unbiased estimate.

**More Details on the NAS Method Comparison Plot (Figure 5.2)**

For each architecture below, we report its number of parameters and mean ± std logistic loss across 5 stand-alone training runs in brackets.

We have the reference architecture 32-144-24 (41,153 parameters, 0.4454 ± 0.0003) for NAS methods to match. In the search space with $20^3 = 8000$ candidate architectures:

- TabNAS trials with no fewer than 2,048 Monte-Carlo samples and the RL learning rate $\eta$ among {0.001, 0.005, 0.01} consistently finds either the reference architecture itself, or an architectures that is qualitatively the same as the reference, like 32-112-32 (40,241 parameters, 0.4456 ± 0.0003).
- NAS with the Abs Reward: After grid search over RL learning rate $\eta$ (among {0.0001, 0.0005, 0.001, 0.005, 0.01, 0.015, 0.02, 0.025, 0.03, 0.04, 0.05, 0.06, 0.07, 0.08, 0.09, 0.1, 0.15, 0.2, 0.25, 0.3, 0.4, 0.5, 0.75, 1.0, 1.5, 2.0}) and $\beta$ (among {-0.0005, -0.001, -0.005, -0.01, -0.05, -0.1, -0.5, -0.75, -1.0, -1.25, -1.5, -2.0, -3.0}), the RL controller finds 32-64-96 (41,345 parameters, 0.4461 ± 0.0003) or 32-80-64 (40,785 parameters, 0.4459 ± 0.0002) among over 90% trials that eventually find an architecture within ±5% of the target number of parameters 41,153.

### D.2.3 Difficulty in using the MnasNet reward

With the MnasNet reward, only fewer than 1% NAS trials in our hyperparameter grid search (the ones with a medium $\beta$) can find an architecture whose number of parameters is within ±5% of the reference, and among which none or only one (out of tens) can match the reference performance. In contrast, TuNAS with the Abs Reward finds an architecture with number of parameters within ±5% of the reference among over 50% of the grid search trials described in Appendix D.2.2, and TabNAS with the rejection-based reward consistently finds such architectures at medium RL learning rates $\eta$ and decently large numbers of MC samples $N$. This means it is significantly more difficult to use the MnasNet reward than competing approaches in the practice of resource-constrained tabular NAS.

## D.3 More failure cases of the Abs Reward

For each architecture below, we report its number of parameters and mean ± std loss across 5 stand-alone training runs (logistic loss for Criteo, balanced error for the others) in brackets.

**On Criteo, in the 4-layer search space.** We have the reference architecture 48-128-16-112 (59,697 parameters, 0.4451 ± 0.0002) for NAS to match in the search space (shown as Figure D.2(a)). Similar to Figure 5.2, we show similar results on NAS with rejection-based reward (TabNAS) and NAS with the Abs Reward (TuNAS) in Figure D.3(a). In the search space with $20^4 = 1.6 \times 10^5$ candidate architectures:

- TabNAS with 32,768 Monte-Carlo samples and RL learning rate $\eta$ among $\{0.001, 0.005, 0.01\}$ consistently finds architectures qualitatively the same as the reference. Example results include 48-128-24-32 (59,545 parameters, 0.4449 ± 0.0002), 48-144-16-48 (59,585 parameters, 0.4448 ± 0.0001), 48-112-16-144 (59,233 parameters, 0.4448 ± 0.0002) and the reference architecture itself.
- NAS with the Abs Reward successfully finds the reference architecture 48-128-16-112 in 3 out of 338 hyperparameter settings on a $\beta$-$\eta$ grid. Other found architectures include 48-80-32-112 (59,665 parameters, 0.4452 ± 0.0002), 32-128-80-144 (59,249 parameters, 0.4453 ± 0.0003) and 48-160-8-48 (58,953 parameters, 0.4448 ± 0.0003), among which the first two are inferior to the TabNAS-found counterparts.

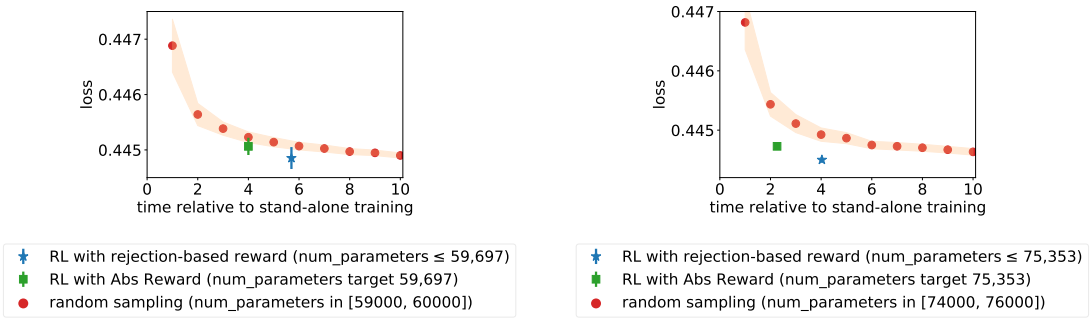
**On Criteo, in the 5-layer search space.** We have the reference architecture 48-240-24-256-8 (75,353 parameters, 0.4448 ± 0.0002) for NAS methods to match in

the search space (shown as Figure D.2(b)). Similar to Figure 5.2, we have similar results on the comparison among random sampling, NAS with rejection-based reward (TabNAS), and NAS with the Abs Reward as Figure D.3(b). In the search space with $20^5 = 3.2 \times 10^6$ candidate architectures:

- TabNAS with 32,768 Monte-Carlo samples and the RL learning rate $\eta = 0.005$ consistently finds architectures qualitatively the same as the reference. Example results include 48-176-64-16-256 (74,945 parameters, 0.4445 ± 0.0002), 48-208-48-48-64 (75,121 parameters, 0.4444 ± 0.0001), 48-256-32-80-24 (74,721 parameters, 0.4446 ± 0.0003) and 48-176-80-16-96 (75,153 parameters, 0.4445 ± 0.0002).
- NAS with the Abs Reward finds 64-80-48-8-8 (75,353 parameters, 0.4448 ± 0.0001), 64-80-24-16-112 (75,353 parameters, 0.4447 ± 0.0001), 48-144-96-16-192 (75,329 parameters, 0.4446 ± 0.0001) and 64-96-8-32-64 (75,273 parameters, 0.4445 ± 0.0001) that are mostly inferior to the TabNAS-found architectures.

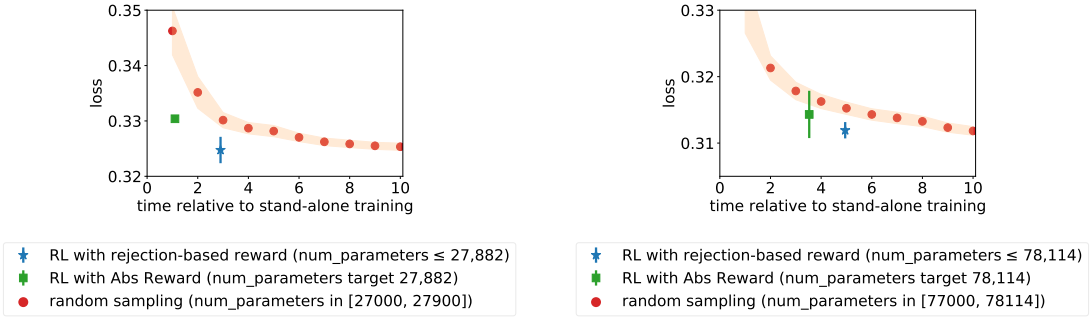
**On Volkert, in the 4-layer search space.** We have the reference architecture 48-160-32-144 (27,882 parameters, 0.3244 ± 0.0040) for NAS to match in the search space (shown as Figure D.2(c)). Similar to Figure 5.2, we draw the comparison plot among random sampling, NAS with rejection-based reward (TabNAS), and NAS with the Abs Reward as Figure D.3(c). In the search space with $1.6 \times 10^5$ candidate architectures:

- TabNAS with $1 \times 10^4$ Monte-Carlo samples and the RL learning rate $\eta \in \{0.001, 0.005, 0.01, 0.05\}$ consistently finds either the reference architecture itself or other architectures qualitatively the same. Examples include 64-128-48-16 (27,050 parameters, 0.3237 ± 0.0040), 80-48-112-32 (27,802 param-




(a) Criteo, 4-layer search space

(b) Criteo, 5-layer search space




(c) Volkert, 4-layer search space

(d) Volkert, 9-layer search space

Figure D.3: Rejection-based reward distributionally outperforms random search and resource-aware Abs Reward in a number of search spaces. The points and error bars have the same meaning as in Figure 5.2. The time taken for each stand-alone training run (the unit length for x axes) is 2.5 hours on Criteo (Figure 5.2, D.3(a) and D.3(b)), 10 minutes on Volkert with 4-layer FFNs (Figure D.3(c)), and 22-25 minutes on Volkert with 9-layer FFNs (Figure D.3(d)).

eters, 0.3274 ± 0.0037), 64-96-80-24 (27,778 parameters, 0.3279 ± 0.0005), and 64-144-32-48 (27,658 parameters, 0.3204 ± 0.0038).

- NAS with the Abs Reward finds 96-64-32-48 (27,738 parameters, 0.3302 ± 0.0042), 96-48-32-96 (27,738 parameters, 0.3305 ± 0.0047), 96-80-16-48 (27,738 parameters, 0.3302 ± 0.0050), 112-48-24-24 (27,722 parameters, 0.3301 ± 0.0034) and 80-80-48-48 (27,690 parameters, 0.3309 ± 0.0022) that are inferior.

**On Volkert, in the 9-layer search space.** We further do NAS on Volkert in the 9-layer search space to test the ability of TabNAS in searching among significantly

deeper FFNs. The tradeoff between loss and number of parameters in the search space is shown in Figure D.2(d). We have the reference architecture 144-128-112-16-16-48-144-24-160 (78,114 parameters, 0.3126 ± 0.0050) for NAS to match. We compare random sampling, NAS with rejection-based reward (TabNAS), and NAS with the Abs Reward in Figure D.3(d). In the search space with $5.2 \times 10^9$ candidate architectures (which is nearly impossible for exhaustive search):

- TabNAS with $5 \times 10^6$ Monte-Carlo samples and the RL learning rate $\eta \in \{0.002, 0.005\}$ consistently finds architectures that are qualitatively the same as the reference. These architectures are found when the RL controller is far from converged and when $\mathbb{P}(V)$ slightly decreases after RL starts. Example results include 144-144-112-64-24-16-128-8-128 (78,026 parameters, 0.3120 ± 0.0049), 128-160-96-32-24-64-64-32-160 (77,890 parameters, 0.3127 ± 0.0040), 128-144-112-32-64-64-80-16-128 (77,834 parameters, 0.3094 ± 0.0012), 160-128-96-32-48-64-48-24-112 (78,002 parameters, 0.3137 ± 0.0021), and 144-112-160-24-112-16-16-128-48 (77,986 parameters, 0.3119 ± 0.0029).
- NAS with the Abs Reward finds 144-96-80-80-48-64-96-80-32 (78,170 parameters, 0.3094 ± 0.0039), 160-80-160-24-80-16-80-64-128 (78,114 parameters, 0.3158 ± 0.0020), 128-96-80-80-64-64-80-80-80 (78,106 parameters, 0.3128 ± 0.0020), and 144-128-80-16-16-16-96-160-24 (78,050 parameters, 0.3192 ± 0.0014). Interestingly, all architectures except 144-96-80-80-48-64-96-80-32 are inferior to the TabNAS-found architectures despite having slightly more parameters, and 144-96-80-80-48-64-96-80-32 does not have an evident bottleneck structure like the other architectures found here.

As a side note, previous works like MnasNet and TuNAS (often or only on vision tasks) do often have inverted bottleneck blocks [107] in their search spaces. However, the search spaces used there have a hard-coded requirement

that certain layers must have bottlenecks. In contrast, our search spaces permit the controller to automatically determine whether to use bottleneck structures based on the task under consideration. This is important because networks with bottlenecks do not always outperform others on all tasks. For example, the reference architecture 32-144-24 outperforms the TuNAS-found 32-64-96 on Criteo, but the reference 64-192-48-32 (64,568 parameters, 0.0662 ± 0.0011) is on par with the TuNAS-and-TabNAS-found 96-80-96-32 (64,024 parameters, 0.0669 ± 0.0013) on Aloi.

## D.4 Comparison with weight-sharing Bayesian optimization and evolutionary search

Bayesian optimization (BO) and evolutionary search (ES) are popular stratigies for NAS (see e.g., [67, 71, 136, 153]; [85, 4]). We are not aware of any work that successfully applies BO or ES for weight-sharing NAS. Thus we design the following (novel) methods of BO or ES for weight-sharing NAS: train the SuperNet for the same number of epochs as RL, and then do BO (by Gaussian processes [105] with expected improvement [94, 69]) or ES in the set of feasible architectures with the SuperNet one-shot losses (evaluated from SuperNet weights). These methods omit the extra forward passes for RL controller training, but need extra forward passes to evaluate child networks. We control the number of passes for a fair comparison. On Criteo, the cost of forward passes for RL is comparable to evaluating 405 child networks, and the search space of 5-layer FFNs has 340,590 feasible architectures below the 75,353 parameters limit in Figure 11(b). The corresponding reference architecture is 48-240-24-

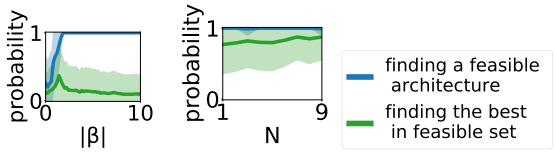
Table D.4: Comparison of TabNAS, Bayesian optimization (BO) and evolutionary search (ES) with weight sharing. TabNAS finds the architecture with the smallest loss.

| method | found architecture (number of parameters, mean ± std loss) |
|---|---|
| TabNAS ($N$=32,768) | **48-176-64-16-256 (74,945 parameters, 0.4445 ± 0.0002)** |
| BO (RBF kernel L=10) | 64-80-8-16-16 (72,073 parameters, 0.4447 ± 0.0002) |
| BO (RBF kernel L=1) | 48-80-48-96-96 (71,265 parameters, 0.4451 ± 0.0003) |
| ES (10 steps, population 100) | 48-96-16-144-64 (67,393 parameters, 0.4450 ± 0.0002) |
| ES (60 steps, population 50) | 48-48-80-80-64 (67,345 parameters, 0.4452 ± 0.0003) |

256-8. The architectures found by BO and ES (under multiple hyperparameter settings) are sensitive to initialization and are worse (see Table D.4). Thus RL explores the search space more efficiently than BO and ES: it finds the global optimum with fewer forward passes.

## D.5 Difficulty of hyperparameter tuning

Hyperparameter tuning has always been a headache for machine learning. In the design of NAS approaches, the hope is that the NAS hyperparameters are much easier to tune than the architectures NAS search over. We denote the RL learning rate and the number of MC samples by $\eta$ and $N$, respectively. The three resource-aware rewards (in MnasNet and TuNAS) have both $\eta$ and $\beta$ as hyperparameters; our TabNAS with the rejection-based reward has $\eta$ and $N$ to tune.




(a) Tuning $\beta$ in Abs Reward (b) Tuning $N$ in rejection

Figure D.4: **Tuning $\beta$ and $N$ on the toy example (Figure 5.1)**: the number of MC samples $N$ in rejection-based reward is easier to tune than $\beta$ in Abs Reward, and is easier to succeed. The lines and shaded regions are mean and standard deviation across 200 independent runs, respectively.

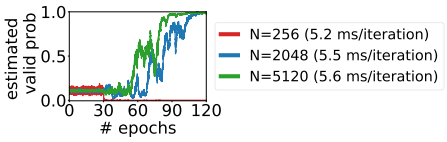



Figure D.5: **Tuning $N$ on Criteo**: the change of $\widehat{\mathbb{P}}(V)$ when the number of Monte-Carlo samples $N$ is 256, 2,048 or 5,120, and the time taken for each iteration. We show results with RL learning rate $\eta = 0.005$; those other $\eta$ values have similar failure patterns.

### D.5.1 Resource hyperparameter $\beta$

$\beta$ is difficult to tune in experiments: the best value varies by dataset and lies in the middle of its search space. Since $\beta < 0$, we discuss its absolute value. In a NAS search space, the architecture that is feasible and can match the reference performance often has the number of parameters that is more than 98% of the reference. A too small $|\beta|$ is not powerful enough to enforce the resource constraint, in which case NAS finds an architecture that is far from the target number of parameters and makes the search nearly unconstrained (e.g., the Abs Reward with $|\beta| = 1$ in the toy example, shown in Figure 5.1 and towards the left end in Figure D.4(a)). A too large $|\beta|$ severely penalizes the violation of the resource constraint, in which case the RL controller would always give an architecture close to the reference, with much bias (e.g., the Abs Reward with $|\beta| = 2$ in Figure 5.1, and towards the right end in Figure D.4(a)). Thus practitioners seek a medium $|\beta|$ in hyperparameter tuning to both obey the resource constraint and achieve a better result. In our experiments, such "appropriate" medium values vary largely across datasets: 1 on Criteo with the 32-144-24 ref-

erence architecture (41,153 parameters), 2 on Volkert with the 48-160-32-144 reference architecture (27,882 parameters), and 25 on Aloi with the 64-192-48-32 reference architecture (64,568 parameters).

### D.5.2 RL learning rate $\eta$

The RL learning rate $\eta$ is easier to tune and more generalizable across datasets than $\beta$. With a large $\eta$, the RL controller quickly converges right after the first 25% epochs of layer warmup; with a small $\eta$, the RL controller converges slowly or may not converge, although there may still be enough signal from the layer-wise probabilities to get the final result. It is thus straightforward to tune $\eta$ by observing the convergence behavior of sampling probabilities. In our experiments, the appropriate value of $\eta$ does not significantly vary across tasks: a constant $\eta \in [0.001, 0.01]$ is appropriate for all datasets and all number of parameter limits.

### D.5.3 Number of MC samples $N$

The number of MC samples $N$ is also easier to tune than $\beta$. Resource permitting, $N$ is the larger, the better (Figure D.4(b)), so that $\mathbb{P}(V)$ can be better estimated. When $N$ is too small, the MC sampling has a high chance of missing the valid architectures in the search space, and thus incurs large bias and variance for the estimate of $\nabla \log[\mathbb{P}(y|y \in V)]$. In such cases, $\widehat{\mathbb{P}}(V)$ may miss all valid architectures at the beginning of RL and quickly converge to 0. $\widehat{\mathbb{P}}(V)$ being equal or close to 0 is a bad case for our rejection-based algorithm: the single-step RL objective

$J(y)$ that has a $-\log(\widehat{\mathbb{P}}(V))$ term grows extremely large and gives an explosive gradient to stuck the RL controller in the current choice. Consequently, the criterion for choosing $N$ is to choose the largest that can afford, and hopefully, at least choose the smallest that can make $\widehat{\mathbb{P}}(V)$ steadily increase during RL. Figure D.5 shows the changes of $\widehat{\mathbb{P}}(V)$ on Criteo with the 32-144-24 reference in the search space of 8,000 architectures at three $N$ values. The NAS succeeds when $N \geq 2048$, same as the threshold that makes $\widehat{\mathbb{P}}(V)$ increase.

Overall, the RL controller with our rejection-based reward has hyperparameters that are easier to tune than with resource-aware rewards in MnasNet and TuNAS.

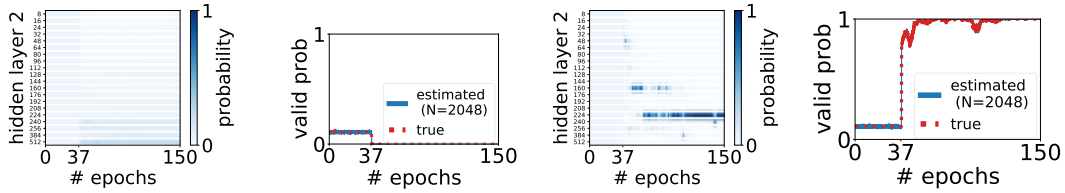
## D.6 More on ablation with a non-differentiable $\mathbb{P}(V)$ (or $\widehat{\mathbb{P}}(V)$)

As discussed in Section 5.4.3, the found architectures are significantly worse when $\mathbb{P}(V)$ is omitted from $J(y)$. Below are our experimental findings:

- In the case that we do not skip infeasible architectures in weight updates, the largest hidden layer sizes may gain and maintain the largest sampling probabilities soon after RL starts. This is because most architectures in the 3-layer Criteo search space are above the number of parameters limit 41,153. When RL starts, the sampled feasible architectures underperform the moving average, thus their logits are severely penalized, making the logits of the infeasible architectures (which often have wide hidden layers) quickly dominate (Figure D.6(a)). Accordingly, the (estimated) valid probability $\mathbb{P}(V)$ (or $\widehat{\mathbb{P}}(V)$) quickly decrease to 0 (Figure D.6(b)), and the RL controller gets stuck (as described in Appendix D.5.3) in these large choices

for hidden layer sizes.

- In the case that we skip infeasible architectures in both weight and RL updates, the RL controller eventually picks feasible architectures with bottleneck structures, but the found architectures are almost always suboptimal: when RL starts, the controller severely boosts the logits of the sampled feasible architectures without much exploration in the search space, and quickly gets stuck there. For example, the search in Figure D.6(c)) finds 24-384-16 (40,449 parameters) that is feasible but suboptimal; $\mathbb{P}(V)$ and $\widehat{\mathbb{P}}(V)$ quickly increase to 1 after RL starts (Figure D.6(d)).




(a) Layer 2, not skipping infeasible in training

(b) Valid probabilities, not skipping infeasible in training

(c) Layer 2, skipping infeasible in training

(d) Valid probabilities, skipping infeasible in training

Figure D.6: **Failure cases in ablation when $\widehat{\mathbb{P}}(V)$ is non-differentiable**. We show results with RL learning rate $\eta = 0.005$; those under other $\eta$ values are similar.

## D.7 Proofs

### D.7.1 $\widehat{\mathbb{P}}(V)$ is an unbiased and consistent estimate of $\mathbb{P}(V)$

Within the search space $S$, recall the definitions of $\mathbb{P}(V)$ and $\widehat{\mathbb{P}}(V)$:

- $\mathbb{P}(V) = \sum_{z^{(i)} \in S} p^{(i)} \mathbb{1}(z^{(i)} \in V)$

- $\widehat{\mathbb{P}}(V) = \frac{1}{N} \sum_{k\in[N], z^{(k)}\in V} \frac{p^{(k)}}{q^{(k)}} = \frac{1}{N} \sum_{k\in[N]} \frac{p^{(k)}}{q^{(k)}} \mathbb{1}(z^{(k)} \in V)$

**Unbiasedness.** With $N$ architectures sampled from the proposal distribution $q$, we take the expectation with respect to $N$ sampled architectures:

$$\begin{aligned}
\mathbb{E}[\widehat{\mathbb{P}}(V)] &= \frac{1}{N} \mathbb{E}\left[ \sum_{k\in[N], z^{(k)}\in V} \frac{p^{(k)}}{q^{(k)}} \right] \\
&= \frac{1}{N} \mathbb{E}\left[ \sum_{k\in[N]} \frac{p^{(k)}}{q^{(k)}} \mathbb{1}(z^{(k)} \in V) \right] \\
&= \frac{1}{N} \sum_{k\in[N]} \mathbb{E}\left[ \frac{p^{(k)}}{q^{(k)}} \mathbb{1}(z^{(k)} \in V) \right],
\end{aligned}$$

in which each summand

$$\begin{aligned}
\mathbb{E}\left[ \frac{p^{(k)}}{q^{(k)}} \mathbb{1}(z^{(k)} \in V) \right] &= \sum_{z^{(k)}\in S} q^{(k)} \frac{p^{(k)}}{q^{(k)}} \mathbb{1}(z^{(k)} \in V) \\
&= \mathbb{P}(V),
\end{aligned}$$

Thus $\mathbb{E}[\widehat{\mathbb{P}}(V)] = \mathbb{P}(V)$.

**Consistency.** We first show the variance of $\mathbb{P}(V)$ converges to 0 as the number of MC samples $N \to \infty$. Because of independence among samples,

$$\mathrm{Var}[\widehat{\mathbb{P}}(V)] = \frac{1}{N} \sum_{k\in[N]} \mathrm{Var}\left[ \frac{p^{(k)}}{q^{(k)}} \mathbb{1}(z^{(k)} \in V) \right],$$

in which each summand

$$\begin{aligned}
\mathrm{Var}\left[ \frac{p^{(k)}}{q^{(k)}} \mathbb{1}(z^{(k)} \in V) \right] &= \mathbb{E}\left[ \frac{p^{(k)}}{q^{(k)}} \mathbb{1}(z^{(k)} \in V) - \mathbb{P}(V) \right] \\
&= \sum_{z^{(k)}\notin V} q^{(k)} \mathbb{P}(V)^2 + \sum_{z^{(k)}\in V} q^{(k)} \left[ \frac{p^{(k)}}{q^{(k)}} - \mathbb{P}(V) \right]^2 \qquad \text{(D.1)} \\
&= -\mathbb{P}(V)^2 + \sum_{z^{(k)}\in V} \frac{(p^{(k)})^2}{q^{(k)}},
\end{aligned}$$

thus the variance

$$\begin{aligned}
\mathrm{Var}[\widehat{\mathbb{P}}(V)] &= \frac{1}{N} \sum_{k\in[N]} \mathrm{Var}\left[ \frac{p^{(k)}}{q^{(k)}} \mathbb{1}(z^{(k)} \in V) \right] \\
&\frac{1}{N} \left[ -\mathbb{P}(V)^2 + \sum_{z^{(k)}\in V} \frac{(p^{(k)})^2}{q^{(k)}} \right],
\end{aligned}$$

which goes to 0 as $N \to \infty$. It worths noting that when we set $q = \text{stop\_grad}(p)$, the single-summand variance (Equation D.1) becomes $\mathbb{P}(V) - \mathbb{P}(V)^2$, which is the variance of a Bernoulli distribution with mean $\mathbb{P}(V)$.

The Chebyshev's Inequality states that for a random variable $X$ with expectation $\mu$, for any $a > 0$, $\mathbb{P}(|X - \mu| > a) \leq \frac{\text{Var}(X)}{a^2}$. Thus $\lim_{N\to\infty} \text{Var}(X) = 0$ implies that $\lim_{N\to\infty} \mathbb{P}(|X - \mu| > a) = 0$ for any $a > 0$, indicating consistency.

### D.7.2 $\nabla \log[\mathbb{P}(y)/\widehat{\mathbb{P}}(V)]$ is a consistent estimate of $\nabla \log[\mathbb{P}(y \mid y \in V)]$

Since $\mathbb{P}(y \mid y \in V) = \frac{\mathbb{P}(y)}{\mathbb{P}(V)}$, we show $\operatorname{plim}_{N\to\infty} \nabla \log \widehat{\mathbb{P}}(V) = \nabla \log \mathbb{P}(V)$ below to prove consistency, in which $\operatorname{plim}_{N\to\infty}$ denotes convergence in probability.

Recall $p^{(i)}$ is the probability of sampling the $i$-th architecture $z^{(i)}$ within the search space $S$, and the definitions of $\mathbb{P}(V)$ and $\widehat{\mathbb{P}}(V)$ are:

- $\mathbb{P}(V) = \sum_{z^{(i)} \in S} p^{(i)} \mathbb{1}(z^{(i)} \in V)$,
- $\widehat{\mathbb{P}}(V) = \frac{1}{N} \sum_{k\in[N], z^{(k)}\in V} \frac{p^{(k)}}{q^{(k)}} = \frac{1}{N} \sum_{k\in[N]} \frac{p^{(k)}}{q^{(k)}} \mathbb{1}(z^{(k)} \in V)$, in which each $p^{(k)}$ is differentiable with respect to all logits $\{\ell_{ij}\}_{i\in[L], j\in[C_i]}$.

Thus we have

$$\begin{aligned}
\operatorname*{plim}_{N\to\infty} \widehat{\mathbb{P}}(V) &= \operatorname*{plim}_{N\to\infty} \frac{1}{N} \sum_{k\in[N]} \frac{p^{(k)}}{q^{(k)}} \mathbb{1}(z^{(k)} \in V) \\
&= \frac{1}{N} \sum_{z^{(k)}\in S} \frac{p^{(k)}}{q^{(k)}} \mathbb{1}(z^{(k)} \in V) N q^{(k)} \\
&= \sum_{z^{(k)}\in S} p^{(k)} \mathbb{1}(z^{(k)} \in V) = \mathbb{P}(V),
\end{aligned}$$

and

$$\begin{aligned}\operatorname*{plim}_{N\to\infty} \nabla\widehat{\mathbb{P}}(V) &= \operatorname*{plim}_{N\to\infty} \frac{1}{N}\sum_{k\in[N]} \frac{\nabla p^{(k)}}{q^{(k)}} \mathbb{1}(z^{(k)} \in V) \\ &= \frac{1}{N}\sum_{z^{(k)}\in S} \frac{\nabla p^{(k)}}{q^{(k)}} \mathbb{1}(z^{(k)} \in V) N q^{(k)} \\ &= \sum_{z^{(k)}\in S} \nabla p^{(k)} \mathbb{1}(z^{(k)} \in V) = \nabla\mathbb{P}(V).\end{aligned}$$

Together with the condition that $\mathbb{P}(V) > 0$ (the search space contains at least one feasible architecture), we have the desired result for consistency as $\operatorname*{plim}_{N\to\infty} \nabla \log \widehat{\mathbb{P}}(V) = \operatorname*{plim}_{N\to\infty} \frac{\nabla\widehat{\mathbb{P}}(V)}{\widehat{\mathbb{P}}(V)} = \frac{\operatorname*{plim}_{N\to\infty} \nabla\widehat{\mathbb{P}}(V)}{\operatorname*{plim}_{N\to\infty} \widehat{\mathbb{P}}(V)} = \frac{\nabla\mathbb{P}(V)}{\mathbb{P}(V)} = \nabla \log \mathbb{P}(V)$, in which the equalities hold due to the properties of convergence in probability.

## BIBLIOGRAPHY


[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Marcin Andrychowicz, Misha Denil, Sergio Gomez, Matthew W Hoffman, David Pfau, Tom Schaul, Brendan Shillingford, and Nando De Freitas. Learning to learn by gradient descent by gradient descent. *Advances in neural information processing systems*, 29, 2016.

[3] Charles Audet, Jean Bigeon, Dominique Cartier, Sébastien Le Digabel, and Ludovic Salomon. Performance indicators in multiobjective optimization. *European journal of operational research*, 2020.

[4] Noor Awad, Neeratyoy Mallik, and Frank Hutter. Differential evolution for neural architecture search. *arXiv preprint arXiv:2012.06400*, 2020.

[5] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

[6] Maria-Florina Balcan, Tuomas Sandholm, and Ellen Vitercik. Generalization in portfolio-based algorithm selection. *arXiv preprint arXiv:2012.13315*, 2020.

[7] Rémi Bardenet, Mátyás Brendel, Balázs Kégl, and Michele Sebag. Collaborative hyperparameter tuning. In *International conference on machine learning*, pages 199–207. PMLR, 2013.

[8] Thomas Bartz-Beielstein and Sandor Markon. Tuning search algorithms for real-world applications: A regression tree based approach. In *Congress on Evolutionary Computation*, volume 1, pages 1111–1118. IEEE, 2004.

[9] Stephen R Becker, Emmanuel J Candès, and Michael C Grant. Templates for convex cone problems with applications to sparse signal recovery. *Mathematical programming computation*, 3(3):165, 2011.

[10] Gabriel Bender, Pieter-Jan Kindermans, Barret Zoph, Vijay Vasudevan, and Quoc Le. Understanding and simplifying one-shot architecture search. In *International Conference on Machine Learning*, pages 550–559. PMLR, 2018.

[11] Gabriel Bender, Hanxiao Liu, Bo Chen, Grace Chu, Shuyang Cheng, Pieter-Jan Kindermans, and Quoc V Le. Can weight sharing outperform random architecture search? An investigation with TuNAS. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 14323–14332, 2020.

[12] Yoshua Bengio. Gradient-based optimization of hyperparameters. *Neural computation*, 12(8):1889–1900, 2000.

[13] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. *Advances in neural information processing systems*, 24, 2011.

[14] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of machine learning research*, 13(2), 2012.

[15] Bernd Bischl, Jakob Richter, Jakob Bossek, Daniel Horn, Janek Thomas, and Michel Lang. mlrMBO: A modular framework for model-based optimization of expensive black-box functions. *arXiv preprint arXiv:1703.03373*, 2017.

[16] C.M. Bishop. *Pattern Recognition and Machine Learning*. Information Science and Statistics. Springer, 2006.

[17] George EP Box. Science and statistics. *Journal of the American Statistical Association*, 71(356):791–799, 1976.

[18] Stephen Boyd and Lieven Vandenberghe. *Convex optimization*. Cambridge University Press, 2004.

[19] Leon Bungert, Tim Roith, Daniel Tenbrinck, and Martin Burger. A bregman learning framework for sparse neural networks. *arXiv preprint arXiv:2105.04319*, 2021.

[20] Han Cai, Ligeng Zhu, and Song Han. Proxylessnas: Direct neural architecture search on target task and hardware. *arXiv preprint arXiv:1812.00332*, 2018.

[21] Zhaowei Cai and Nuno Vasconcelos. Rethinking differentiable search for mixed-precision neural networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2349–2358, 2020.

[22] J Douglas Carroll and Jih-Jie Chang. Analysis of individual differences in multidimensional scaling via an n-way generalization of "eckart-young" decomposition. *Psychometrika*, 35(3):283–319, 1970.

[23] Rich Caruana, Art Munson, and Alexandru Niculescu-Mizil. Getting the most out of ensemble selection. In *ICDM*, pages 828–833. IEEE, 2006.

[24] Rich Caruana, Alexandru Niculescu-Mizil, Geoff Crew, and Alex Ksikes. Ensemble selection from libraries of models. In *ICML*, page 18. ACM, 2004.

[25] Boyuan Chen, Harvey Wu, Warren Mo, Ishanu Chattopadhyay, and Hod Lipson. Autostacker: A compositional evolutionary learning system. In *Proceedings of the genetic and evolutionary computation conference*, pages 402–409, 2018.

[26] Pei-Hung Chen, Hsiang-Fu Yu, Inderjit Dhillon, and Cho-Jui Hsieh. Drone: Data-aware low-rank compression for large nlp models. *Advances in Neural Information Processing Systems*, 34, 2021.

[27] Carlos A Coello Coello and Margarita Reyes Sierra. Multiobjective evolutionary algorithms: classifications, analyses, and new innovations. In *Evolutionary Computation*. Citeseer, 1999.

[28] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Training deep neural networks with low precision multiplications. *arXiv preprint arXiv:1412.7024*, 2014.

[29] Tiago Cunha, Carlos Soares, and André C. P. L. F. de Carvalho. Cf4cf: Recommending collaborative filtering algorithms using collaborative filtering. In *Proceedings of the 12th ACM Conference on Recommender Systems*, RecSys '18, pages 357–361, New York, NY, USA, 2018. ACM.

[30] Christopher De Sa, Megan Leszczynski, Jian Zhang, Alana Marzoev,

Christopher R Aberger, Kunle Olukotun, and Christopher Ré. High-accuracy low-precision training. *arXiv preprint arXiv:1803.03383*, 2018.

[31] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, 2002.

[32] Arthur P Dempster, Nan M Laird, and Donald B Rubin. Maximum likelihood from incomplete data via the em algorithm. *Journal of the Royal Statistical Society: Series B (Methodological)*, 39(1):1–22, 1977.

[33] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. ImageNet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255. IEEE, 2009.

[34] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[35] Xuanyi Dong and Yi Yang. Nas-bench-201: Extending the scope of reproducible neural architecture search. *arXiv preprint arXiv:2001.00326*, 2020.

[36] Iddo Drori, Yamuna Krishnamurthy, Remi Rampin, Raoni Lourenço, J One, Kyunghyun Cho, Claudio Silva, and Juliana Freire. Alphad3m: Machine learning pipeline synthesis. In *AutoML Workshop at ICML*, 2018.

[37] Iddo Drori, Yamuna Krishnamurthy, Remi Rampin, Raoni de Paula Lourenco, Jorge Piazentin Ono, Kyunghyun Cho, Claudio Silva, and Juliana Freire. Alphad3m: Machine learning pipeline synthesis. *arXiv preprint arXiv:2111.02508*, 2021.

[38] Iddo Drori, Lu Liu, Yi Nian, Sharath C Koorathota, Jie S Li, Antonio Khalil Moretti, Juliana Freire, and Madeleine Udell. AutoML using metadata language embeddings. *arXiv preprint arXiv:1910.03698*, 2019.

[39] Dheeru Dua and Casey Graff. UCI machine learning repository, 2017.

[40] Romain Egele, Prasanna Balaprakash, Isabelle Guyon, Venkatram Vishwanath, Fangfang Xia, Rick Stevens, and Zhengying Liu. Agebo-tabular: joint neural architecture and hyperparameter search with autotuned data-parallel training for tabular data. In *Proceedings of the International Con-*

*ference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2021.

[41] Nick Erickson, Jonas Mueller, Alexander Shirkov, Hang Zhang, Pedro Larroy, Mu Li, and Alexander Smola. Autogluon-tabular: Robust and accurate AutoML for structured data. *arXiv preprint arXiv:2003.06505*, 2020.

[42] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Springenberg, Manuel Blum, and Frank Hutter. Efficient and robust automated machine learning. *Advances in neural information processing systems*, 28, 2015.

[43] Matthias Feurer, Jost Tobias Springenberg, and Frank Hutter. Using meta-learning to initialize bayesian optimization of hyperparameters. In *MetaSel@ ECAI*, pages 3–10. Citeseer, 2014.

[44] Matthias Feurer, Jan N van Rijn, Arlind Kadra, Pieter Gijsbers, Neeratyoy Mallik, Sahithya Ravi, Andreas Müller, Joaquin Vanschoren, and Frank Hutter. Openml-python: an extensible python api for openml. *arXiv preprint arXiv:1911.02490*, 2019.

[45] Peter I Frazier. A tutorial on bayesian optimization. *arXiv preprint arXiv:1807.02811*, 2018.

[46] Nicolo Fusi, Rishit Sheth, and Melih Elibol. Probabilistic matrix factorization for automated machine learning. *Advances in neural information processing systems*, 31, 2018.

[47] Timur Garipov, Pavel Izmailov, Dmitrii Podoprikhin, Dmitry P Vetrov, and Andrew G Wilson. Loss surfaces, mode connectivity, and fast ensembling of dnns. In *Advances in Neural Information Processing Systems*, pages 8789–8798, 2018.

[48] P. Gijsbers, E. LeDell, S. Poirier, J. Thomas, B. Bischl, and J. Vanschoren. An Open Source AutoML Benchmark. *arXiv preprint arXiv:1907.00909 [cs.LG]*, 2019. Accepted at AutoML Workshop at ICML 2019.

[49] Gene H Golub and Charles F Van Loan. *Matrix computations*. JHU Press, 2012.

[50] Yury Gorishniy, Ivan Rubachev, Valentin Khrulkov, and Artem Babenko. Revisiting deep learning models for tabular data. *arXiv preprint arXiv:2106.11959*, 2021.

[51] Ming Gu and Stanley C Eisenstat. Efficient algorithms for computing a strong rank-revealing qr factorization. *SIAM Journal on Scientific Computing*, 17(4):848–869, 1996.

[52] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep learning with limited numerical precision. In *International conference on machine learning*, pages 1737–1746. PMLR, 2015.

[53] William W Hager. Updating the inverse of a matrix. *SIAM review*, 31(2):221–239, 1989.

[54] Richard A Harshman et al. Foundations of the parafac procedure: Models and conditions for an" explanatory" multimodal factor analysis. 1970.

[55] David A Harville. Matrix algebra from a statistician's perspective, 1998.

[56] Soheil Hashemi, Nicholas Anthony, Hokchhay Tann, R Iris Bahar, and Sherief Reda. Understanding the impact of precision quantization on the accuracy and energy of neural networks. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, pages 1474–1479. IEEE, 2017.

[57] Trevor Hastie, Rahul Mazumder, Jason D Lee, and Reza Zadeh. Matrix completion and low-rank svd via fast alternating least squares. *The Journal of Machine Learning Research*, 16(1):3367–3402, 2015.

[58] Elad Hazan, Adam Klivans, and Yang Yuan. Hyperparameter optimization: a spectral approach. In *ICLR*, 2018.

[59] Ralf Herbrich, Neil D Lawrence, and Matthias Seeger. Fast sparse Gaussian process methods: The informative vector machine. In *Advances in Neural Information Processing Systems*, pages 625–632, 2003.

[60] Sara Hooker. The hardware lottery. *arXiv preprint arXiv:2009.06489*, 2020.

[61] Timothy Hospedales, Antreas Antoniou, Paul Micaelli, and Amos Storkey. Meta-learning in neural networks: A survey. *arXiv preprint arXiv:2004.05439*, 2020.

[62] Andrew Howard, Mark Sandler, Grace Chu, Liang-Chieh Chen, Bo Chen, Mingxing Tan, Weijun Wang, Yukun Zhu, Ruoming Pang, Vijay Vasudevan, et al. Searching for mobilenetv3. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 1314–1324, 2019.

[63] Ling Huang, Jinzhu Jia, Bin Yu, Byung-Gon Chun, Petros Maniatis, and Mayur Naik. Predicting execution time of computer programs using sparse polynomial regression. In *Advances in Neural Information Processing Systems*, pages 883–891, 2010.

[64] Frank Hutter, Youssef Hamadi, Holger H Hoos, and Kevin Leyton-Brown. Performance prediction and automated tuning of randomized and parametric algorithms. In *International Conference on Principles and Practice of Constraint Programming*, pages 213–228. Springer, 2006.

[65] Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Sequential Model-Based Optimization for General Algorithm Configuration. *LION*, 5:507–523, 2011.

[66] Frank Hutter, Lin Xu, Holger H Hoos, and Kevin Leyton-Brown. Algorithm runtime prediction: Methods & evaluation. *Artificial Intelligence*, 206:79–111, 2014.

[67] Haifeng Jin, Qingquan Song, and Xia Hu. Auto-keras: An efficient neural architecture search system. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 1946–1956, 2019.

[68] RC St John and Norman R Draper. D-optimality for regression designs: a review. *Technometrics*, 17(1):15–23, 1975.

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

[70] Arlind Kadra, Marius Lindauer, Frank Hutter, and Josif Grabocka. Well-tuned simple nets excel on tabular datasets. In *Thirty-Fifth Conference on Neural Information Processing Systems*, 2021.

[71] Kirthevasan Kandasamy, Willie Neiswanger, Jeff Schneider, Barnabas Poczos, and Eric P Xing. Neural architecture search with bayesian optimisation and optimal transport. *Advances in neural information processing systems*, 31, 2018.

[72] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. On large-batch training for deep learning: Generalization gap and sharp minima. 2017.

[73] Farhan Khawar, Xu Hang, Ruiming Tang, Bin Liu, Zhenguo Li, and Xiuqiang He. Autofeature: Searching for feature interactions and their architectures for click-through rate prediction. In *Proceedings of the 29th ACM International Conference on Information & Knowledge Management*, pages 625–634, 2020.

[74] Mikhail Khodak, Neil A. Tenenholtz, Lester Mackey, and Nicolò Fusi. Initalization and regularization of factorized neural layers. In *Proceedings of the 10th International Conference on Learning Representations*, 2021.

[75] Tamara G Kolda and Brett W Bader. Tensor decompositions and applications. *SIAM review*, 51(3):455–500, 2009.

[76] Andreas Krause, Ajit Singh, and Carlos Guestrin. Near-optimal sensor placements in Gaussian processes: Theory, efficient algorithms and empirical studies. *Journal of Machine Learning Research*, 9(Feb):235–284, 2008.

[77] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009.

[78] Brenden M Lake, Tomer D Ullman, Joshua B Tenenbaum, and Samuel J Gershman. Building machines that learn and think like people. *Behavioral and brain sciences*, 40, 2017.

[79] Hamed F Langroudi, Zachariah Carmichael, David Pastuch, and Dhireesha Kudithipudi. Cheetah: Mixed low-precision hardware & software codesign framework for dnns on the edge. *arXiv preprint arXiv:1908.02386*, 2019.

[80] Rui Leite, Pavel Brazdil, and Joaquin Vanschoren. Selecting classification algorithms with active testing. In *International Workshop on Machine Learning and Data Mining in Pattern Recognition*, pages 117–131. Springer, 2012.

[81] Christiane Lemke, Marcin Budka, and Bogdan Gabrys. Metalearning: a survey of trends and technologies. *Artificial Intelligence Review*, 44(1):117–130, 2015.

[82] Hao Li, Zheng Xu, Gavin Taylor, Christoph Studer, and Tom Goldstein. Visualizing the loss landscape of neural nets. In *Advances in Neural Information Processing Systems*, pages 6389–6399, 2018.

[83] Miqing Li, Shengxiang Yang, and Xiaohui Liu. Diversity comparison of

Pareto front approximations in many-objective optimization. *IEEE Transactions on Cybernetics*, 44(12):2568–2584, 2014.

[84] Bin Liu, Chenxu Zhu, Guilin Li, Weinan Zhang, Jincai Lai, Ruiming Tang, Xiuqiang He, Zhenguo Li, and Yong Yu. Autofis: Automatic feature interaction selection in factorization models for click-through rate prediction. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2636–2645, 2020.

[85] Chenxi Liu, Barret Zoph, Maxim Neumann, Jonathon Shlens, Wei Hua, Li-Jia Li, Li Fei-Fei, Alan Yuille, Jonathan Huang, and Kevin Murphy. Progressive neural architecture search. In *Proceedings of the European conference on computer vision (ECCV)*, pages 19–34, 2018.

[86] Hanxiao Liu, Karen Simonyan, and Yiming Yang. Darts: Differentiable architecture search. *arXiv preprint arXiv:1806.09055*, 2018.

[87] Sijia Liu, Parikshit Ram, Deepak Vijaykeerthy, Djallel Bouneffouf, Gregory Bramble, Horst Samulowitz, Dakuo Wang, Andrew Conn, and Alexander Gray. An ADMM Based Framework for AutoML Pipeline Configuration. *arXiv preprint arXiv:1905.00424*, 2019.

[88] Wei Ma and George H Chen. Missing not at random in matrix completion: The effectiveness of estimating missingness probabilities under a low nuclear norm assumption. In *Advances in Neural Information Processing Systems*, volume 32, pages 14900–14909, 2019.

[89] David JC MacKay. Information-based objective functions for active data selection. *Neural Computation*, 4(4):590–604, 1992.

[90] Vivek Madan, Mohit Singh, Uthaipon Tantipongpipat, and Weijun Xie. Combinatorial algorithms for optimal design. In *Conference on Learning Theory*, pages 2210–2258, 2019.

[91] Rahul Mazumder, Trevor Hastie, and Robert Tibshirani. Spectral regularization algorithms for learning large incomplete matrices. *The Journal of Machine Learning Research*, 11:2287–2322, 2010.

[92] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, et al. Mixed precision training. *arXiv preprint arXiv:1710.03740*, 2017.

[93] Mustafa Mısır and Michèle Sebag. Alors: An algorithm recommender system. *Artificial Intelligence*, 244:291–314, 2017.

[94] Jonas Močkus. On Bayesian methods for seeking the extremum. In *Optimization techniques IFIP technical conference*, pages 400–404. Springer, 1975.

[95] Alexander M Mood et al. On Hotelling's weighing problem. *The Annals of Mathematical Statistics*, 17(4):432–446, 1946.

[96] George L Nemhauser, Laurence A Wolsey, and Marshall L Fisher. An analysis of approximations for maximizing submodular set functions—i. *Mathematical programming*, 14(1):265–294, 1978.

[97] Randal S Olson and Jason H Moore. Tpot: A tree-based pipeline optimization tool for automating machine learning. In *Automated Machine Learning*, pages 151–160. Springer, 2019.

[98] Randal S Olson, Ryan J Urbanowicz, Peter C Andrews, Nicole A Lavender, La Creis Kidd, and Jason H Moore. Automating biomedical data science through tree-based pipeline optimization. In *European conference on the applications of evolutionary computation*, pages 123–137. Springer, 2016.

[99] Ivan V Oseledets. Tensor-train decomposition. *SIAM Journal on Scientific Computing*, 33(5):2295–2317, 2011.

[100] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32:8026–8037, 2019.

[101] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, 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.

[102] Bernhard Pfahringer, Hilan Bensusan, and Christophe G Giraud-Carrier. Meta-learning by landmarking various learning algorithms. In *ICML*, pages 743–750, 2000.

[103] Hieu Pham, Melody Guan, Barret Zoph, Quoc Le, and Jeff Dean. Effi-

cient neural architecture search via parameters sharing. In *International Conference on Machine Learning*, pages 4095–4104. PMLR, 2018.

[104] Friedrich Pukelsheim. *Optimal design of experiments*, volume 50. SIAM, 1993.

[105] Carl Edward Rasmussen. Gaussian processes in machine learning. In *Summer school on machine learning*, pages 63–71. Springer, 2003.

[106] Carl Edward Rasmussen and Christopher KI Williams. *Gaussian processes for machine learning*. the MIT Press, 2006.

[107] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4510–4520, 2018.

[108] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE transactions on neural networks*, 20(1):61–80, 2008.

[109] Roy Schwartz, Jesse Dodge, Noah A Smith, and Oren Etzioni. Green ai. *arXiv preprint arXiv:1907.10597*, 2019.

[110] Paola Sebastiani and Henry P Wynn. Maximum entropy sampling and optimal Bayesian experimental design. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 62(1):145–157, 2000.

[111] Zeyuan Shang, Emanuel Zgraggen, Benedetto Buratti, Ferdinand Kossmann, Philipp Eichmann, Yeounoh Chung, Carsten Binnig, Eli Upfal, and Tim Kraska. Democratizing data science through interactive curation of ml pipelines. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1171–1188, 2019.

[112] Or Sharir, Barak Peleg, and Yoav Shoham. The cost of training nlp models: A concise overview. *arXiv preprint arXiv:2004.08900*, 2020.

[113] Jack Sherman and Winifred J Morrison. Adjustment of an inverse matrix corresponding to a change in one element of a given matrix. *The Annals of Mathematical Statistics*, 21(1):124–127, 1950.

[114] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[115] Kate Smith-Miles and Jano van Hemert. Discovering the suitability of optimisation algorithms by learning from evolved instances. *Annals of Mathematics and Artificial Intelligence*, 61(2):87–104, 2011.

[116] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. *Advances in neural information processing systems*, 25, 2012.

[117] Nimit Sharad Sohoni, Christopher Richard Aberger, Megan Leszczynski, Jian Zhang, and Christopher Ré. Low-memory neural network training: A technical report. *arXiv preprint arXiv:1904.10631*, 2019.

[118] Nimit Sharad Sohoni, Christopher Richard Aberger, Megan Leszczynski, Jian Zhang, and Christopher Ré. Low-memory neural network training: A technical report, 2019.

[119] Qingquan Song, Hancheng Ge, James Caverlee, and Xia Hu. Tensor completion algorithms in big data analytics. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 13(1):1–48, 2019.

[120] Niranjan Srinivas, Andreas Krause, Sham Kakade, and Matthias Seeger. Gaussian Process Optimization in the Bandit Setting: No Regret and Experimental Design. In *ICML*, pages 1015–1022, 2010.

[121] David H Stern, Horst Samulowitz, Ralf Herbrich, Thore Graepel, Luca Pulina, and Armando Tacchella. Collaborative Expert Portfolio Management. In *AAAI*, pages 179–184, 2010.

[122] Xiao Sun, Jungwook Choi, Chia-Yu Chen, Naigang Wang, Swagath Venkataramani, Vijayalakshmi (Viji) Srinivasan, Xiaodong Cui, Wei Zhang, and Kailash Gopalakrishnan. Hybrid 8-bit floating point (hfp8) training and inference for deep neural networks. *Advances in Neural Information Processing Systems*, 32, 2019.

[123] Xiao Sun, Naigang Wang, Chia-Yu Chen, Jiamin Ni, Ankur Agrawal, Xiaodong Cui, Swagath Venkataramani, Kaoutar El Maghraoui, Vijayalakshmi Viji Srinivasan, and Kailash Gopalakrishnan. Ultra-low precision 4-bit training of deep neural networks. *Advances in Neural Information Processing Systems*, 33, 2020.

[124] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V Le. Mnasnet: Platform-aware neural architecture search for mobile. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2820–2828, 2019.

[125] Chris Thornton, Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Auto-weka: Combined selection and hyperparameter optimization of classification algorithms. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 847–855, 2013.

[126] Sebastian Thrun and Lorien Pratt. *Learning to learn*. Springer Science & Business Media, 2012.

[127] Ledyard R Tucker. Some mathematical notes on three-mode factor analysis. *Psychometrika*, 31(3):279–311, 1966.

[128] Madeleine Udell and Alex Townsend. Why are big data matrices approximately low rank? *SIAM Mathematics of Data Science (SIMODS), to appear*, 2018.

[129] Joaquin Vanschoren. Meta-learning: A survey. *arXiv preprint arXiv:1810.03548*, 2018.

[130] Joaquin Vanschoren, Jan N. van Rijn, Bernd Bischl, and Luis Torgo. Openml: Networked science in machine learning. *SIGKDD Explorations*, 15(2):49–60, 2013.

[131] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[132] Abraham Wald. On the efficient design of statistical investigations. *The Annals of Mathematical Statistics*, 14(2):134–140, 1943.

[133] Bram Wallace and Bharath Hariharan. Extending and analyzing self-supervised learning across domains. In *European Conference on Computer Vision*, pages 717–734. Springer, 2020.

[134] Ruoxi Wang, Rakesh Shivanna, Derek Cheng, Sagar Jain, Dong Lin, Lichan Hong, and Ed Chi. Dcn v2: Improved deep & cross network and

practical lessons for web-scale learning to rank systems. In *Proceedings of the Web Conference 2021*, pages 1785–1797, 2021.

[135] Wei Wen, Hanxiao Liu, Yiran Chen, Hai Li, Gabriel Bender, and Pieter-Jan Kindermans. Neural predictor for neural architecture search. In *European Conference on Computer Vision*, pages 660–676. Springer, 2020.

[136] Colin White, Willie Neiswanger, and Yash Savani. Bananas: Bayesian optimization with neural architectures for neural architecture search. *arXiv preprint arXiv:1910.11858*, 1(2):4, 2019.

[137] Martin Wistuba, Nicolas Schilling, and Lars Schmidt-Thieme. Learning hyperparameter optimization initializations. In *2015 IEEE international conference on data science and advanced analytics (DSAA)*, pages 1–10. IEEE, 2015.

[138] David H Wolpert and William G Macready. No free lunch theorems for optimization. *IEEE transactions on evolutionary computation*, 1(1):67–82, 1997.

[139] Bichen Wu, Yanghan Wang, Peizhao Zhang, Yuandong Tian, Peter Vajda, and Kurt Keutzer. Mixed precision quantization of convnets via differentiable neural architecture search. *arXiv preprint arXiv:1812.00090*, 2018.

[140] Jin Wu and Shapour Azarm. Metrics for quality assessment of a multiobjective design optimization solution set. *J. Mech. Des.*, 123(1):18–25, 2001.

[141] Chengrun Yang, Yuji Akimoto, Dae Won Kim, and Madeleine Udell. OBOE: Collaborative filtering for AutoML model selection. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1173–1183, 2019.

[142] Chengrun Yang, Gabriel Bender, Hanxiao Liu, Pieter-Jan Kindermans, Madeleine Udell, Yifeng Lu, Quoc Le, and Da Huang. Resource-constrained neural architecture search on tabular datasets. *arXiv preprint arXiv:2204.07615*, 2022.

[143] Chengrun Yang, Lijun Ding, Ziyang Wu, and Madeleine Udell. TenIPS: Inverse Propensity Sampling for Tensor Completion. In *International Conference on Artificial Intelligence and Statistics*, pages 3160–3168. PMLR, 2021.

[144] Chengrun Yang, Jicong Fan, Ziyang Wu, and Madeleine Udell. AutoML

pipeline selection: Efficiently navigating the combinatorial space. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1446–1456, 2020.

[145] Chengrun Yang, Ziyang Wu, Jerry Chee, Christopher De Sa, and Madeleine Udell. How low can we go: Trading memory for error in low-precision training. *arXiv preprint arXiv:2106.09686*, 2021.

[146] Yuan Yao, Lorenzo Rosasco, and Andrea Caponnetto. On early stopping in gradient descent learning. *Constructive Approximation*, 26(2):289–315, 2007.

[147] Chris Ying, Aaron Klein, Eric Christiansen, Esteban Real, Kevin Murphy, and Frank Hutter. Nas-bench-101: Towards reproducible neural architecture search. In *International Conference on Machine Learning*, pages 7105–7114. PMLR, 2019.

[148] Dani Yogatama and Gideon Mann. Efficient transfer learning method for automatic hyperparameter tuning. In *Artificial Intelligence and Statistics*, pages 1077–1085, 2014.

[149] Chris Zhang, Mengye Ren, and Raquel Urtasun. Graph hypernetworks for neural architecture search. *arXiv preprint arXiv:1810.05749*, 2018.

[150] Hantian Zhang, Jerry Li, Kaan Kara, Dan Alistarh, Ji Liu, and Ce Zhang. The zipml framework for training models with end-to-end low precision: The cans, the cannots, and a little bit of deep learning. *arXiv preprint arXiv:1611.05402*, 2016.

[151] Tianyi Zhang, Zhiqiu Lin, Guandao Yang, and Christopher De Sa. QPyTorch: A low-precision arithmetic simulation framework, 2019.

[152] Yuyu Zhang, Mohammad Taha Bahadori, Hang Su, and Jimeng Sun. FLASH: fast Bayesian optimization for data analytic pipelines. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 2065–2074. ACM, 2016.

[153] Hongpeng Zhou, Minghao Yang, Jun Wang, and Wei Pan. Bayesnas: A bayesian approach for neural architecture search. In *International conference on machine learning*, pages 7603–7613. PMLR, 2019.

[154] Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng

Zou. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *arXiv preprint arXiv:1606.06160*, 2016.

[155] Eckart Zitzler and Lothar Thiele. Multiobjective optimization using evolutionary algorithms—a comparative case study. In *International conference on parallel problem solving from nature*, pages 292–301. Springer, 1998.

[156] Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016.