Accelerating Active Machine Learning

by

Scott Sievert

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Electrical Engineering)

at the

UNIVERSITY OF WISCONSIN–MADISON

2021

Date of final oral examination: 2021-11-29

The dissertation is approved by the following members of the Final Oral Committee:

Robert Nowak, Professor, Electrical & Computer Engineering
Laurent Lessard, Associate Professor, Mechanical and Industrial Engineering (Northeastern University)
Dan Negrut, Professor, Mechanical Engineering
Timothy Rogers, Professor, Psychology

For John

ACKNOWLEDGMENTS

This dissertation describes work performed in my journey through graduate school. I am indebted to enumerable colleagues, friends and family for their invaluable help, including my co-authors. Any errors which remain are my sole responsibility.

I would like to thank some individuals in particular. I would especially like to thank Robert Nowak and Rebecca Willett for allowing me to explore the focus for this dissertation. Finding the topic for my dissertation is a direct result of the HAMLET and LUCID collaborations between social scientists and machine learning researchers, both organized in part by Robert Nowak and Timothy Rogers. Kevin Jamieson has provided motivation for large portions of this dissertation, some of which also come from the HAMLET collaboration.

In addition, I would like to thank particular individuals for particular pieces of guidance, though this list is not exhaustive. Timothy Rogers and Robert Nowak provided valuable experimental design feedback to properly test software I developed. Laurent Lessard provided valuable teaching insight in addition to providing a very pragmatic outlook on optimization and programming. Dan Negrut taught a class on high performance computation, which inspired large portions of this dissertation. He clearly identified and explained performance bottlenecks in high performance computation, and his slides and notes will remain reference materials. Matthew Rocklin has provided a (very) useful tool for distributed computation and has provided valuable development advice. Zachary Charles has provided guidance on the utility of mathematical theorems.

I would also like to thank many others for non-technical advice

and guidance, especially my friends and family. They have provided direction and guidance during my journey through graduate school that has indirectly effected this dissertation. The most relevant advice comes from my father, John Sievert who has provided very useful advice on technical presentations.

The work presented in this dissertation would not be possible without support from the Department of Defense's SMART Scholarship Program, the National Science Foundation's LUCID graduate training program, Innovative Signal Analysis, and the Electrical & Computer Engineering Department at the University of Wisconsin–Madison.

— Scott Sievert (2021)

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Machine learning (ML) models typically require many data for training. The number of data can be reduced with "active" ML, which chooses the most informative data for training. However, active ML can be computationally intense, which is costly in time and/or resources. Better active ML performance can arise from accelerating the core components required for the active ML data flow, training the ML model and determining query priority.

If the data flow of active ML is accelerated with the obvious solution of using more hardware, then several subtle architecture questions can be posed (e.g., when training a ML model with 4 graphical processing units a.k.a. GPUs, how should those GPUs communicate?). In all cases, the answers to these questions will require less wall-clock time to obtain a solution or enable previously unrealized behavior. This dissertation will clearly explain the architecture changes and benefits, and present experiments that illustrate the practical benefits that the user experiences.

1 INTRODUCTION

Many popular machine learning (ML) applications rely on large datasets provided by humans, making them expensive to collect. In one example, ML has become an "indispensable tool for drug designers" [99] and can outperform practicing radiologists [130] – but these successes rely on labels created be experts, making them costly to collect. Another example is the ImageNet database, a database of at least 3.2 million images that are hand annotated by at least 10 humans per image through a crowdsourcing tool [47]. This dataset is popular with researchers (e.g., [58, 62, 75, 91, 131, 133]), perhaps in part due to a popular annual contest that has attracted participation from over 50 institutions [137].

The data collection process is expensive whenever human intervention is required. One example requires many humans to perform a simple task, and the other requires a few experts with an advanced education to perform a nontrivial task. These collected data can be used to obtain useful models – for example, the ImageNet database has been used to create a object-detection model capable of at least 90% top-5 accuracy on unseen images [133, Sec. 3.3].

Active machine learning addresses the challenges posed by this information bottleneck by using fewer data to achieve the same result, typically a model of the same quality. This is enabled by adapting to previous responses to ask about the most useful or informative queries [31, 158]. For example, active ML can reduce the number of experiments required for drug discovery in addition to assisting scientists [112].

One illustration of active ML is with The New Yorker (TNY) Cartoon Caption Contest. Each week, TNY draws a cartoon and asks their readers for funny captions, and they collect up to about 10,000 captions. TNY has to find the funniest caption from these captions, and they now use the knowledge of the crowd to help. In their crowdsourcing interface, they

present the comic alongside one caption with buttons to rate the caption as "unfunny," "somewhat funny" or "funny" as shown in Fig. 1.1.

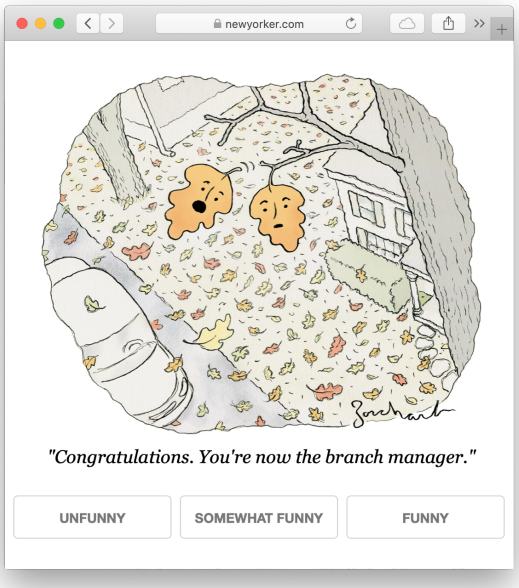


Figure 1.1: A query from The New Yorker Cartoon Caption Contest

The caption's "funniness" is evaluated by finding the average score when buttons are assigned funniness scores of 1, 2 and 3. For TNY Cartoon Caption Contest, the deployed active ML algorithm [159] clearly identifies the funniest captions given the number of responses. The intuition behind the active algorithm is simple: because the goal is find the *funniest* caption, it only makes sense to ask about the *funny* captions. Notably, this process clearly identifies the funniest captions, an improvement over a "passive" or non-active sampling algorithm. A performance indicator is with the number of captions that could possibly be the funniest, illustrated in Fig. 1.2.

TNY Cartoon Caption Contest is a remarkably successful application

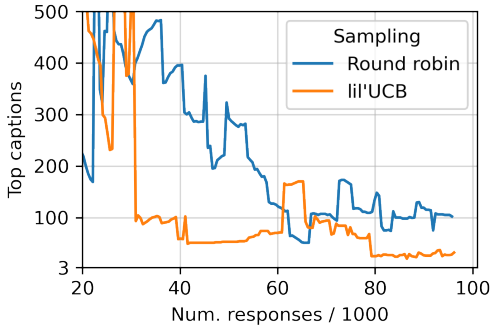


Figure 1.2: The number of captions that could possibly be the funniest aka "top captions" after a particular number of human responses have been received for contest #519. During this contest, two different sampling schemes were run, an adaptive scheme (lil'UCB [70]) and a passive scheme (round robin, which rotates through the captions). Here, a caption i with mean rating μ_i and variance σ_i^2 is a "top caption" if $\mu_3 - \sigma_3 < \mu_i + \sigma_i$ when the third highest score is μ_3 because TNY chooses 3 winners.

of active ML: TNY has been using active learning for about five years[1] and has undergone three system architecture changes.[2] In total, over 200 million human ratings have been recorded to over 1.5 million captions.[3] This dataset has been used to improve the underlying sampling algorithm [159].

Generally, deploying active ML algorithms to crowdsourcing audiences provides utility but can break down when the problem of interest is more complicated than a simple ranking. Let's examine that case more closely in Section 1.1, which will lay the groundwork for the core motivation of this dissertation in Section 1.2.

[1] At first (in Dec. 2015) they used lil'UCB [70] then switched to KL-LUCB [159] in March 2017.

[2] TNY Cartoon Caption Contest has been run with NEXT, a refactored NEXT, and a specialized solution with Amazon AWS.

[3] https://nextml.github.io/caption-contest-data/ and https://nextml.github.io/caption-contest-data2/.

1.1 Crowdsourcing active machine learning

Running active ML algorithms in a crowdsourcing application has provided significant benefit in recent years, including an 18% revenue lift on Microsoft's landing page [3]. Microsoft's system to run active ML in crowdsourcing contexts is now used *by default* on Microsoft's landing page and deployed in another half dozen contexts [3].

Other active ML systems have been developed to allow experimentalists and/or domain experts to collect data more easily [27, 35, 71]. One system in particular, NEXT is designed to accelerate the development of active ML algorithms [71]. Towards this goal, it's relatively easy to implement active ML algorithms in NEXT, and to deploy and test algorithms with experimentalists through any crowdsourcing service, including Amazon Mechanical Turk [149]. This tight feedback loop enables addressing particular issues in the algorithms through close integration with domain experts. Use cases include improving algorithms designed to find the best item for the average user [159] or for a specific user [79], to find the relative similarity between items [108, 132, 145], and to generate a clustered ranking [81].

In particular, finding a shoe that a user would like to purchase presented significant challenges [79]. In this use case, the user is presented with an image of one shoe, and they are asked to rate it as positive or negative before a new shoe appears. There is a catalog of 50,000 shoes (each of which has 1,000 features), and scoring every shoe in the catalog is required after every human response before a new shoe can appear. This requires computation of 50,000 inner products (one for each shoe) for many popular algorithms [10, 51, 94].[4] Performing this simple computation for 50,000 shoes meant the user would wait 1.2 seconds before seeing another query.[5]

[4]Each score required the computation of $\boldsymbol{x}_i^T\boldsymbol{\theta}_k + f(\boldsymbol{x}_i^T\boldsymbol{A}\boldsymbol{x}_i)$ for some function $f : \mathbb{R} \to \mathbb{R}$, a shoe's feature vector $\boldsymbol{x}_i \in \mathbb{R}^{1000}$ and a user preference vector $\boldsymbol{\theta}_k \in \mathbb{R}^{1000}$ [79].

[5]The 1.2 second delay relied on reformulation into $\boldsymbol{x}_i^T\boldsymbol{B}\boldsymbol{x}_i$ for some matrix $\boldsymbol{B}$,

This delay scaled linearly with the number of shoes: having 100,000 shoes would result in a 2.4 second delay.

A 1.2 second delay is unacceptable in a web context (e.g., when crowdsourcing with Mechanical Turk). As a result, Jun et al. developed an acceleration to the query search, one to return the optimal query with high probability [79]. This acceleration enabled deploying an active ML algorithm to crowdsourcing participants, and to scale to large shoe databases.

This dissertation will also focus on accelerating components of the active ML data flow with a different and extremely obvious method: using more hardware. Let's examine that more closely in Section 1.2.

1.2 Motivation

Accelerating active machine learning (ML) is the main theme of this dissertation. Practically, this involves accelerating the core components of any active ML algorithm: training the ML model and determining query priority, as shown in Fig. 1.3. These accelerations can enable use of active ML algorithms with crowdsourcing audiences as described in Section 1.1, and have the potential to enable higher performance.

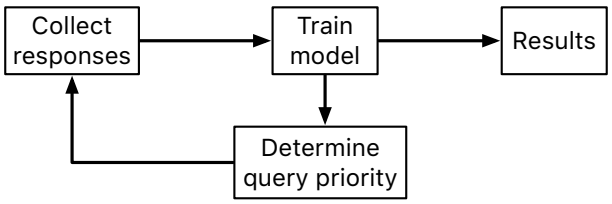


Figure 1.3: The data flow of active ML. For passive ML, each query is equally important and there is no feedback between collection of responses and model training.

One simple and brute force method of acceleration is to use more hardware by adding more computers with a distributed system. For simple

which meant the majority of computation relied on a linear algebra library [164].

computation (e.g., arithmetic) with common cloud providers, acceleration is simple and does not increase the final cost.[6] However, accelerating anything active is not nearly as simple because some serial components must be involved. In the most serial case, a trivial model update depends on a single observation, which depends on the previous model (e.g., [31]). This case gets much more interesting when multiple observations can be observed and/or the model updates are not trivial.

The goal of this dissertation is to accelerate active ML algorithms or components thereof with distributed systems. This will raise subtle architecture questions that will eventually have surprisingly practical benefits (e.g., answering "how does gradient approximation error change during training?" will eventually lead to accelerating ML model training). This dissertation will specifically address the questions below:

1. *How can an specific active ML algorithm for a widespread problem in ML be accelerated with a specific distributed system?* What accelerations are enabled by a specific pairing of algorithm and distributed system, and how large are the gains?
2. *How can a popular model update—empirical risk minimization—be accelerated using more hardware?* With this, data scientists will be able to iterate more quickly because their ML models will train more quickly, a rough proxy for active ML.[7]
3. *What software features are required to effectively deploy a challenging active ML algorithm* for the "ordinal embedding" problem to crowd-

[6] An Amazon AWS `p3.2xlarge` machine has one state-of-the-art NVIDIA V100 GPU and can be rented for $3.06/hour. A `p3.8xlarge` has 4 NVIDIA V100s and costs $12.24/hour, exactly 4 times as much (https://aws.amazon.com/ec2/pricing/on-demand/). In both cases, a job that requires 16 GPU-hours (a fixed number of floating point operations a.k.a. FLOPs) will cost $48.96.

[7] The data scientist decides which experiments to run, and collects results from those computations. In Fig. 1.3, "model training" would be the data scientist updating their internal mental model from these experiments, and choosing experiments would be a substitute for evaluating "query priority."

sourcing audiences? Active ML algorithms often have computationally intense query searches for the ordinal embedding problem. *How should query priority be determined for complete searches?*

These questions will be addressed in Chapters 2, 3, and 4 respectively. In total, every box in the core loop of Figure 1.3 will be accelerated with distributed computation.[8] As a result, the user will experience practical benefits: less than half the resources will be consumed to perform their desired task. Specifically, Chapters 2, 3 and 4 will respectively present the following conclusions:

1. Using an active ML algorithm in a distributed system means a result of a particular quality[9] will be found in $T/3$ minutes, an improvement over the T minutes a passive algorithm requires. Notably, the active ML algorithm would take at least $T/6$ minutes longer if the "query priority" box in Fig. 1.3 were not addressed.
2. ML model training can be completed in about half the time of standard SGD when a distributed system is present or when many model updates are required.
3. Using an active ML algorithm will require half as many human responses to generate a high quality "ordinal embedding."

These conclusions are listed here because there's considerable amount of context to cover before getting to the highlight(s) of each chapter.

Notation Some nomenclature will be used throughout this document:

[8] In addition, the "results" box has been somewhat addressed with the development of a "humor model" [23] has been aided by the TNY Caption Contest dataset collected with NEXT [71].

[9] A model of a particular validation loss in the "hyperparameter optimization" problem.

- "Acceleration" means "to reduce the time required to finish the relevant task."
- "active" and "adaptive" will be used interchangeably. An active ML algorithm and an adaptive ML algorithm both follow the process in Fig. 1.3.
- "Amount of computation" means the "the total number of floating point operations (FLOPs)."[10]
- "Distributed system" means "a collection of computers that work together."
- "Distributed training" means "training a ML model with a distributed system."
- "Epochs" essentially means "one pass through the dataset."[11]
- "ML" stands for "machine learning."
- "Step size" and "learning rate" will be used interchangeably to refer to the size of model update in optimization.

Additionally, bold lower-case letters (e.g, $\boldsymbol{x}$) will refer a vector and normal lower-case letters with a subscript (e.g., x_i) will refer to an element of that vector. Likewise, bold upper-case letters (e.g, $\boldsymbol{A}$) will refer to a matrix, and $A_{i,j}$ will refer to an element of that matrix in the ith row and jth column. Bold lower-case letters (e.g, $\boldsymbol{x}_i$) will refer to a matrix row (or one vector from a collection of vectors).

If a function $f : \mathbb{R}^d \to \mathbb{R}$ has a vector input $\boldsymbol{x} \in \mathbb{R}^d$ and scalar output $f(\boldsymbol{x}) \in \mathbb{R}$, then the gradient of a function will be represented by $\nabla f(\boldsymbol{x})$, which is a vector of the same dimension as $\boldsymbol{x}$.

[10]The commonly used "FLOPS" stands for "FLOPs per second" – which will not be used in this dissertation.

[11]It precisely means "if the dataset has N examples, the gradient for N examples have been computed" because some examples might have the gradient calculated more than once.

2 ACCELERATING MODEL SELECTION WITH DISTRIBUTED COMPUTING

To get started, let's examine a simple use case: accelerating a specific active ML algorithm with a particular distributed system. In relation to Fig. 1.3, this chapter involves effectively determining the query priority when ML models provide responses. In context, implementing this active ML algorithm in this distributed system will reduce the time required to obtain a high quality solution while leaving the final result unchanged.

2.1 Problem

Training any machine learning pipeline requires data, an untrained model and "hyperparameters," parameters chosen before training begins that help with cohesion between the model and data. The user needs to specify values for these hyperparameters in order to use the model. A good example is adapting the ridge regression or LASSO to the amount of noise in the data with the regularization parameter [107, 161]. Hyperparameter choice verification can not be performed until model training is completed.

Model performance strongly depends on the hyperparameters provided, even for the simple examples above. This gets much more complex when multiple hyperparameters are required. For example, a particular visualization tool, t-SNE requires (at least) three hyperparameters [104] and the first section in a study on how to use this tool effectively is titled "[t]hose hyperparameters really matter" [170].

Hyperparameters need to be specified by the user, and their values are typically found through a search over possible values through a "cross validation" search where models are scored on unseen holdout data (e.g., [9]). Even in the simple ridge regression case above, a brute force search is required [107]. This search quickly grows infeasible as the number of hyperparameters grow

because of the frequent interactions between different hyperparameters. A prime example is with deep learning, which has specialized algorithms for handling many data but has difficulty providing basic hyperparameters. For example, the commonly used stochastic gradient descent (SGD) has difficulty with the most basic hyperparameter "learning rate" [18] because of the number of data [105].

Finding the best values for the hyperparameters, or hyperparameter optimization is required if high performance is desired. In practice, it's expensive and time-consuming for machine learning researchers and practitioners. Ideally, hyperparameter optimization algorithms return high performing models quickly and are simple to use.

Quickly returning quality hyperparameters relies on making decisions about which hyperparameters to allocate training time. This might mean progressively choosing higher-performing hyperparameter values or stopping low-performing models early during training. Returning this high performing model quickly would lower the expense and/or time barrier to performing hyperparameter optimization.

2.2 Contributions

Many active ML algorithms for hyperparameter optimization have been proposed (e.g., [14, 15, 86, 87, 95, 154] and references therein). One algorithm in particular, Hyperband requires relatively little computation (because it treats computation as a scarce resource[1]) and finds models with high "validation" scores or low validation losses [95].

Our contributions include a Hyperband implementation in a popular library for distributed ML, "Dask-ML."[2] The performance of the Dask-ML im-

[1]There is little benefit to this implementation if computation is not a scarce resource – that is, if model selection finishes quickly, in less than 15 minutes.

[2]The Dask-ML implementation is available through the `HyperbandSearchCV` class, referenced on `https://ml.dask.org/hyper-parameter-search.html`.

plementation will be illustrated in experiments, alongside some explanation and illustration of the parallel underpinnings of Dask-ML and Hyperband that allow for better performance.

First, let's step through related work in Section 2.3, mostly detailing Hyperband and Dask-ML. Then, let's mention exactly how Hyperband is implemented in Dask-ML in Section 2.4, then let's perform an evaluation of the implementation and internal details in Section 2.5.

2.3 Related work

Hyperparameter optimization finds the optimal set of hyperparameters for a given model. These hyperparameters are chosen to maximize performance on unseen data. The hyperparameter optimization process typically looks like the following:

1. Split the dataset into the train and test datasets. The test dataset is reserved for the final model evaluation.
2. Choose hyperparameters
3. Train models with those hyperparameters
4. Score those models with unseen data (a subset of the train dataset, typically referred to as the "validation set").
5. If not satisfied with the performance, go back to step (2) with refined hyperparameters.
6. Use the best performing hyperparameters to train a model with those hyperparameters on the complete train dataset
7. Score the model on the test dataset.

The score in step (6) is often the score that is reported in papers and/or production. The rest of this section will focus on steps (2), (3) and (5), which is where most of the work happens in hyperparameter optimization.

2.3.1 Hyperparameter optimization

A commonly used method for hyperparameter selection is a random selection of hyperparameters, and is typically followed by training each model to completion. This offers several advantages, including a simple implementation that is very amenable to parallelism. Other benefits include sampling "important" hyperparameters more densely than "unimportant" hyperparameters [14]. This randomized search is implemented in many places, including in Scikit-Learn [123].

These implementations are by definition *passive* because they do not adapt to previous training. *Adaptive* algorithms can return a higher quality solution with less training by adapting to previous training and choosing which hyperparameter values to evaluate. These adaptive algorithms fit into Fig. 1.3 when models with a particular hyperparameter configuration submit cross validation scores as responses.

A popular class of adaptive hyperparameter optimization algorithms are Bayesian algorithms [154]. These algorithms treat the model as a black box and the model scores as an evaluation of that black box. These algorithms have an estimate of the optimal set of hyperparameters and use some probabilistic methods to improve the estimate. The choice of which hyperparameter value to evaluate depends on previous evaluations.

Popular Bayesian searches include sequential model-based algorithm configuration (SMAC) [67], tree-structure Parzen estimator (TPE) [15], and Spearmint [154]. Many of these are available through the "robust Bayesian optimization" package RoBo [86] through AutoML. This package also includes Fabolas, a method that takes dataset size as input and allows for some computational control [87].

2.3.2 Hyperband

Hyperband is a principled early stopping scheme for randomized hyperparameter selection[3] and an adaptive hyperparameter optimization algorithm [95]. At the most basic level, it partially trains models before stopping models with low scores, then repeats. By default, it stops training the lowest performing 67% of the available models at certain times. This means that the number of models decay over time, and the surviving models have high scores (e.g., "validation accuracy" or "negative validation loss.")

Naturally, the quality of any model depends on two factors: the amount of training performed and the values of various hyperparameters. Any hyperparameter optimization algorithm (HOA) needs consider this balance. If training time only matters a little, it makes sense for HOAs to aggressively stop training models. On the flip side, if only training time influences the score, it only makes sense for HOAs to let all models train for as long as possible and not perform any stopping.

Hyperband sweeps over the relative importance of hyperparameter choice and amount of training. This sweep over training time importance enables a theorem that Hyperband will return a much higher performing model than a random search with no early stopping. This is best characterized by an informal presentation of the main theorem:

Corollary 1. *(informal presentation of [95, Theorem 5] and surrounding discussion) Assume a model's validation loss at iteration k decays like $(1/k)^{1/\alpha}$, and the final validation losses ν approximately follow the cumulative distribution function $F(\nu) = (\nu - \nu_*)^\beta$ with optimal validation loss ν_* with $\nu - \nu_* \in [0, 1]$.*

Higher values of α mean slower convergence, and higher values of β represent more difficult hyperparameter optimization problems because it's harder to obtain a validation loss close to the optimal validation loss ν_*. The

[3] In general, Hyperband is a resource-allocation scheme for model selection.

commonly used SGD has a convergence rate with $\alpha = 2$ *[19] [95, Cor. 6],*[4] *and* $\beta > 1$ *means the validation losses are not uniformly distributed.*

For any $T \in \mathbb{N}$*, let* $\widehat{i}_T$ *be the empirically best performing model when models are stopped early according to the infinite horizon Hyperband algorithm when* T *resources have been used to train models. Then with probability* $1-\delta$*, the empirically best performing model* $\widehat{i}_T$ *has loss*

$$\nu_{\widehat{i}_T} \leq \nu_* + c\left(\frac{\overline{\log}(T)^3 \cdot a}{T}\right)^{1/\max(\alpha,\ \beta)}$$

for some constant c *and* $a = \overline{\log}(\log(T)/\delta)$ *where* $\overline{\log}(x) = \log(x\log(x))$*.*

By comparison, finding the best model without the early stopping Hyperband performs (i.e., randomized searches and training until completion) after T *resources have been used to train models has loss*

$$\nu_{\widehat{i}_T} \leq \nu_* + c\left(\frac{\log(T) \cdot a}{T}\right)^{1/(\alpha+\beta)}$$

For simplicity, only the infinite horizon case is presented though much of the analysis carries over to the practical finite horizon Hyperband.[5] Because of this, it only makes sense to compare the loss when the number of resources used T is large. When this happens, the validation loss of the Hyperband produces $\nu_{\widehat{i}_T}$ decays much faster than the uniform allocation scheme.[6] This shows a definite advantage to performing early stopping on randomized searches.

Li et al. show that the model Hyperband identifies as the best is identified with a (near) minimal amount of training [95, Thm. 7], within log factors of

[4]Gradient descent has convergence rates with $\alpha = 1$ [24, Thm. 3.3].

[5]To prove results about the finite horizon algorithm Li et al. only need the result in Corollary 9 [95].

[6]This is clear by examining $\log(\nu_{\widehat{i}_T} - \nu_*)$ for Hyperband and uniform allocation.For Hyperband, the slope is approximately $-1/\max(\alpha,\ \beta)$, which decays much faster than the uniform allocation's approximate slope of $-1/(\alpha+\beta)$.

the known lower bound [82]. Adaptive searches minimize the computational effort by choosing which models to evaluate; there is not much value in that if the amount of computation is limited.

More relevant work involves combining Bayesian searches and Hyperband, which can be combined by using the Hyperband bracket framework and progressively tuning a Bayesian prior to select parameters for each bracket [50]. This work is also available through AutoML, and also works with multiple machines: if there are infinite machines, there's no gains from their Bayesian on Hyperband (BOHB) approach.

2.3.3 Dask

Dask provides advanced parallelism for analytics, especially for NumPy, Pandas and Scikit-learn [41]. It is familiar to Python users and does not require rewriting code or retraining models to scale to larger datasets or to more machines. It can scale up to clusters or to a massive dataset, and also works on laptops and presents the same interface. Dask provides two components:

- Dynamic task scheduling optimized for computation. This low level scheduler provides parallel computation and is optimized for interactive computational workloads.
- "Big Data" collections like parallel arrays, or dataframes, and lists that extend common interfaces like NumPy, Pandas, or Python iterators to larger-than-memory or distributed environments. These parallel collections run on top of dynamic task schedulers.

Dask aims to be familiar and flexible: it aims to parallelize and distribute computation and/or datasets easily while retaining a task scheduling interface for custom workloads and integration into other projects. It is fast and the scheduler has low overhead. It's implemented in pure Python and can scale from massive datasets to a cluster with thousands of cores to a

laptop running a single process. In addition, it's designed with interactive computing and provides rapid feedback and diagnostics to aid humans.

Dask's machine learning library is Dask-ML.[7] Most of the details aren't relevant and focus on preprocessing, linear models, and adapting other libraries to learning with large data. However, Dask-ML has some other hyperparameter optimization algorithms are either focused on expensive preprocessing or working well with many data (both common in NLP). These implementations are passive and can not adapt to prior training. This work will focus on filling the "compute constrained" niche, when models take a long time to train and are adaptive (they can adapt to prior training).[8]

2.4 Dask's implementation of Hyperband

We have implemented Hyperband in Dask-ML with `HyperbandSearchCV` in Dask-ML's model selection module.[9] There are some details in the implementation that enhance performance. To see that, let's begin unwrapping the architecture of both Hyperband and Dask.

Combining Dask and Hyperband is a natural fit. Hyperparameter optimization searches often require significant amounts of computation and can involve large datasets, calling for Dask's ability to scale from laptops to supercomputers.[10] There are two levels of parallelism in Hyperband which are rendered as two loops:

1. An "embarrassingly parallel" sweep over the different brackets of training time importance.

[7] https://ml.dask.org/

[8] See https://ml.dask.org/hyper-parameter-search.html for detail.

[9] The API reference is available at https://ml.dask.org/modules/generated/dask_ml.model_selection.HyperbandSearchCV.html.

[10] The existing passive hyperparameter optimization algorithms in Dask-ML have limited use because they don't adapt to previous training to reduce the amount of training required.

2. Inside each bracket, an early stopping scheme for random search is run. This means the models are trained independently in parallel, except training stops on certain models at certain times.

These two levels of parallelism are rendered as nested loops as shown in Fig. 2.1. The for-loop that runs different "brackets" represents different training vs. exploration tradeoffs, and each bracket does about the approximately the same amount of computation. Each bracket is run in parallel, and Dask's dynamic task scheduling can launch jobs from within each bracket.

Each bracket indicates a value in the trade-off between training time and hyperparameter importance, and is specified by the list of tuples in Fig. 2.1. Each bracket is specified so that the total number of `partial_fit` calls is approximately the same among different brackets. Then, having many models requires pruning models very aggressively and vice versa with few models. As an example, with `max_iter=243` the least adaptive bracket has 5 models and no pruning. The most adaptive bracket has 81 models and fairly aggressive early stopping schedule.

The exact aggressiveness of the early stopping schedule depends on one optional input to `HyperbandSearchCV`, `aggressiveness`. The default value is 3, which has some mathematical motivation [95, Sec. 2.6]. Either the most aggressive bracket or `aggressiveness=4` is likely more suitable for initial exploration when not much is known about the model, data or hyperparameters.

2.4.1 Combination of Dask and Hyperband

Dask can assign different priority levels to different jobs. When a Dask worker is free, it pops the job with the highest priority off the stack of pending jobs. This is relevant because Hyperband has many jobs to run in

```
def sha(n_models: int, iters: int, aggressiveness: int = 3):
    # Successive halving algorithm
    models = [random_model() for i in range(n_models)]
    while True:
        # Calls `m.partial_fit` a total of `iters` times
        models = [train(m, iters) for m in models]
        val_scores = [score(m) for m in models]
        surviving_models = len(models) // aggressiveness
        models = top_k(models, val_scores, k=n_surviving_models)
        iters *= aggressiveness
        if len(models) < aggressiveness:
            return top_k(models, k=1)

def hyperband(max_iter: int = 243, aggressiveness=3):
    # Each bracket does about the same amount of work.
    # Each tuple is (num_models, n_init_calls).
    # More models means more aggressive pruning
    brackets = [(81,3), (34,9), (15,27), (8,81), (5,243)]
    if max_iter != 243:
        brackets = ...  # inputs: max_iter, aggressiveness.

    best_models = [sha(n, r, aggressiveness) for n, r in brackets]
    return top_k(best_models, k=1)
```

Figure 2.1: Condensed psuedo-code for Dask-ML's `HyperbandSearchCV`. The user provides `train`, `score` (which is supplied with validation data), and `random_model`, and specifies how much data each `partial_fit` call receives. Details around `brackets` are hidden for simplicity and can be found in Alg. 1 by Li et al. [95]. Lines 6, 7 and 22 (highlighted) are run in parallel on the workers Dask has available.

parallel (e.g., `HyperbandSearchCV` creates 143 models for `max_iter=81`, all of which require training).

Dask-ML's implementation of Hyperband, `HyperbandSearchCV` can take advantage of the priority scheme by giving higher priority to high performing models. Assigning the priority of fitting the model to be the model's most recent validation score will tell Dask what models to focus on. To be clear, the final model `HyperbandSearchCV` produces won't change – but the final model will be found sooner, which can be advantageous if the user wants to stop computation early.

`HyperbandSearchCV`'s priority scheme to train high scoring models sooner works best in very serial environments: priority makes no difference in very parallel environment when every job can be scheduled instantaneously. In moderately parallel environments, the different priorities may lead to longer time to solution because of suboptimal scheduling. To get around this, the worst performing P models all have the same priority for each bracket when there are P Dask workers (the median of the P validation scores).

Now, let's go over some usability concerns for `HyperbandSearchCV`, which will influence the user-facing API and how `HyperbandSearchCV` is used.

2.4.2 Input parameter rule-of-thumb

Hyperband is also fairly easy to use. It requires two input parameters:

1. The number of `partial_fit` calls for the best model (via `max_iter`)
2. The number of examples that each `partial_fit` call sees (which is implicit and referred to as `chunks`, which can be the "chunk size" of the Dask array).

These two parameters rely on knowing how long to train the model[11]

[11] e.g., something in the form "the most trained model should see 100 times the number of examples (aka 100 epochs)"

and having a rough idea on the number of parameters to evaluate. With those parameters, a rule-of-thumb can be developed:[12]

```
n_params = 200  # sample approximately 200 parameters
training_eg = 100 * len(X_train)  # for longest trained model

## inputs to HyperbandSearchCV
max_iter = n_params
chunks = int(training_eg / n_params)
```

Trying twice as many parameters with the same amount of computation requires halving `chunks` and doubling `max_iter`. The longest-trained model will see the same amount of data. The primary advantage to Hyperband's inputs is that they do not require balancing training time importance and hyperparameter importance. Random searches also require two inputs:

1. How many parameters to try (via `num_params`).
2. The number of examples that each model sees (via `training_eg`).

Trying twice as many parameters with the same amount of computation requires doubling `num_params` and halving `training_eg`, which means every model will see half as many data. Implicitly, a balance between training time and hyperparameter importance is being decided upon. Hyperband's inputs are simpler because it sweeps over different values for this importance in different brackets.

2.4.3 Dwindling number of models

At first, Hyperband evaluates many models. As time progresses, the number of models decay because Hyperband is an early stopping scheme. This

[12] This rule-of-thumb is oriented towards large data. The dataset should be repeated if the dataset is too small (i.e., if `training_eg > n_params * len(X_train)`), possible with `dask.array.repeat` or `numpy.repeat`.

means towards the end of the computation, a few (possibly high-performing) models can be training while most of the computational hardware is idle. This is might be a problem when computational resources are rented (e.g., with cloud platforms like Amazon AWS or Google Cloud Platform).

Hyperband is a principled early stopping scheme, but it doesn't protect against at least two common cases:

- When models have converged before training completes (i.e., the score stays constant).
- When models have not converged and poor hyperparameters are chosen (i.e, the scores are not increasing).

Providing a "stop on plateau" scheme will protect against these cases because training will be stopped if a model's score stops increasing [127]. This will require two additional parameters: `patience` to determine how long to wait before stopping a model, and `tol` which determines how much the score should increase.

Hyperband's early stopping is designed to identify the highest performing model with minimal training. Setting `patience` to be high avoids interference with this scheme, but still protects against both cases above while erring towards giving models more training time. This early stopping scheme is likely most relevant during the least adaptive brackets of Hyperband. So by default, using `HyperbandSearchCV` with `patient=True` sets the high value of `patience=int(max_iter / aggressiveness)`, which is also the patience of the second least adaptive bracket in Hyperband and not incredibly far from the user-specified value of `max_iter`.

2.5 Experimental results

In this section, two hyperparameter optimizations are compared, Hyperband and random search. The incentive to compare these algorithms involves

the fact that Hyperband is a principled early stopping scheme for random search, and randomized search is "embarrassingly parallel." Both of these schemes will be evaluated with a Dask computational cluster. In this section, `HyperbandSearchCV`'s early stopping scheme will score models and stop training models with a lower score.

2.5.1 Serial environment

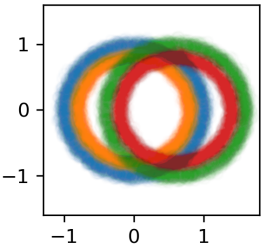
This section is focused on the initial exploration of a model and it's hyperparameters on a personal laptop. This section shows a performance comparison to illustrate the `HyperbandSearchCV`'s utility, which will use a rule-of-thumb detailed in Section 2.4.2 to determine the inputs to `HyperbandSearchCV`.

A synthetic dataset with 6 features shown in Fig. 2.2a is used for a 4-class classification problem on a personal laptop with 4 cores. Let's train a small fully-connected neural network with 24 neurons, and vary the depth/width in addition to tuning SGD. A visualization of this dataset with 4 features and more details on the model/hyperparameters is relegated to Appendix A.1.

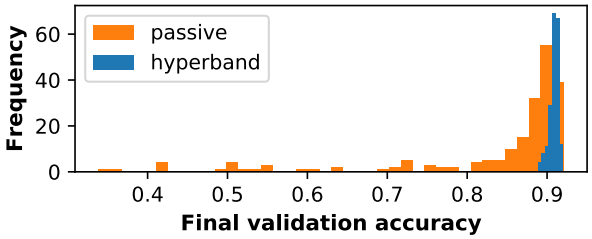
The results of these simulations are in Figs. 2.2b and 2.3 which emulates an experiment where no information on the model or dataset are known. As such, `HyperbandSearchCV`'s `aggressiveness` is set to 4 (not the default 3) and the choice on number of passes through the dataset is chosen without much knowledge.[13] The results in Fig. 2.3a validate the `HyperbandSearchCV`'s effectiveness: it finds high performing hyperparameters with minimal training. Notably, the computational environment makes the hyperparameter selection very serial and the number of `partial_fit` calls or passes through the dataset a decent proxy for time.

The results in Fig. 2.3b validate the method of assigning priorities. The two runs of `HyperbandSearchCV` shown only differ in how priorities are

[13] Accordingly, number of samples for the random search is perhaps underspecified for the random search. Regardless, Hyperband performs well with this minimal amount of computation.



(a) The synthetic dataset used as input for the serial simulations. The colors correspond to different class labels. In addition to these two informative dimensions, there are 4 uninformative dimensions with uniformly distributed random noise.



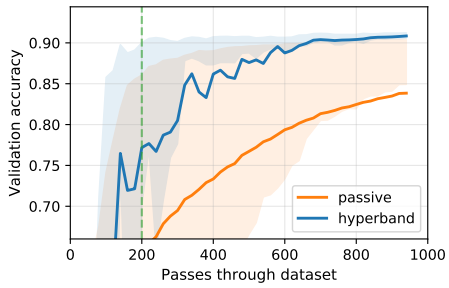
(b) The final validation accuracy over the different runs. Out of the 200 runs, the worst of the `hyperband` runs performs better than 99 of the `passive` runs, and 21 `passive` runs have final validation accuracy less than 70%.

Figure 2.2: The dataset and final validation accuracies on that dataset, which has 60,000 examples (50,000 of which are used for training).

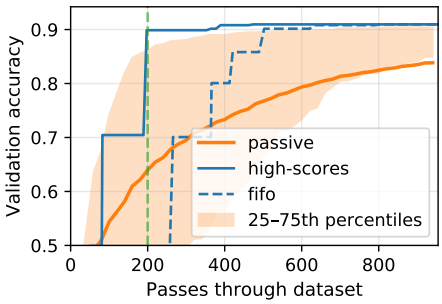
assigned: both runs see the same datasets in the same order, the models have the same internal random state, and the validation scores at the beginning/end are exactly the same. This shows the method of assigning priorities can reduce wall-clock time to obtain a particular validation score by approximately a factor of 2, relevant if computation is stopped early. In attempt to show the effect of this priority scheme in more parallel environments, we selected a run that had a relatively small difference in performance between the two priority schemes.

2.5.2 Parallel environments

This section will examine how Dask performs as the number of workers grow using a model implemented in PyTorch [121], a popular deep learning library. The inputs and desired outputs are given in Fig. 2.4. A shallow neural network will be used because the noise variance varies slightly between



(a) Validation accuracy after a given amount of passes through the dataset (epochs) for Dask's Hyperband implementation `HyperbandSearchCV` (via `hyperband`) and random search (via `passive`).



(b) The priority scheme influences Hyperband's time to solution. Two schemes are relevant: `high-scores`, which assigns higher priority to higher scoring models, and `fifo`, which assigns a "first in first out" priority and is Dask's default priority.

Figure 2.3: A visualization of Dask's performance for a hyperparameter search over 200 runs. Four Dask workers are used, and the dotted green line shows the amount of data required to train four models to completion. The shaded regions represent the 75 and 25th percentiles, and the solid/dashed lines represents the mean.

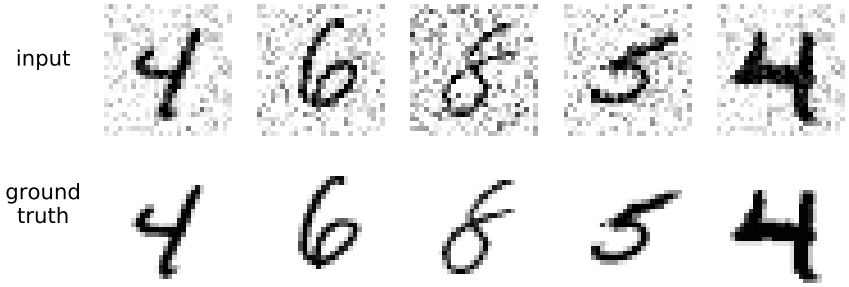


Figure 2.4: The input and ground truth for the image denoising problem. There are 70,000 images in the output, the original MNIST dataset. For the input, random noise is added to images, and amount of data grows to 350,000 input/output images. Each `partial_fit` calls sees (about) 20,780 examples and each call to `score` uses 66,500 examples for validation.

images, a fully-connected auto-encoder detailed in Appendix A.2.

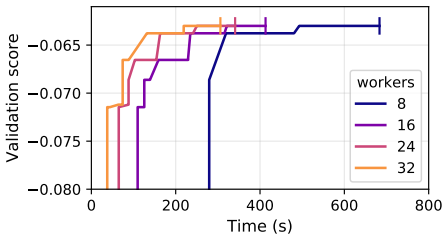
The tuned hyperparameters include one hyperparameter that affects model architecture, the activation function and varies among 4 rectified linear unit variants [37, 62, 103, 113]. The other hyperparameters all control finding the optimal model after the architecture is fixed. These hyperparameters include 3 discrete hyperparameters (with 160 unique combinations) and 3 continuous hyperparameters. Some of these hyperparameters include choices on the optimizer to use (SGD [18] or Adam [85]), initialization, regularization and optimizer hyperparameters like learning rate or momentum. Complete details are in Appendix A.2.

This implementation will set `max_iter=243` to train for about 75 epochs on the original dataset (which is inflated by a factor of 5 to add random noise), and let's set `patience=True` to stop training early plateauing models after about 25 epochs. This is a regression problem, so the negative validation loss will be used to score models (not the validation accuracy as in Section 2.5.1).

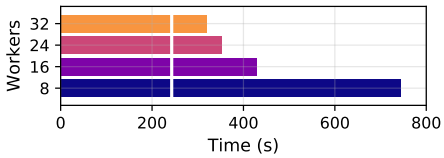
The hyperparameter search will be run once, and the history will be recorded. From this history, simulations will be run where only the number of Dask workers varies. Each model fitting/scoring will return the same values from the recorded history and will take a deterministic time. In these simulations, a `partial_fit` call consumes 1 second and `score` call consumes 1.5 seconds [59]. The results in Fig. 2.5b illustrate what happens if the number of workers changes for this rather complete search. For this search, the total search time for 8 workers is less than 15 minutes and the speedups begin to saturate between 16 and 24 workers.

2.6 Conclusion

This chapter has shown an implementation of a particular hyperparameter optimization algorithm that is amendable to parallelism. This implementation is in a popular distributed computation library, Dask-ML. The combination



(a) The wall-clock time required to reach a particular validation score (the negative loss).



(b) The total wall-clock time to solution for the runs in Fig. 2.5a. The white line marks the time required to train one model to completion.

Figure 2.5: How the Dask implementation of Hyperband scales with the number of workers.

of these two items—Hyperband and Dask—is particularly potent because of a scheme to prioritize model fitting, highlighted in Section 2.4.1. Given Section 2.4.1, the use case of using $N \leq 16$ GPUs on Amazon EC2 instead of 1 GPU is a viable method of accelerating the hyperparameter optimization by about a factor of N.

Future work involves integrating a Bayesian sampling scheme into the Hyperband implementation [50]. Falkner et al. describe updating a Bayesian model for the hyperparameters as training progresses, and initializing a new training model from the Bayesian model when the chance occurs. If there are infinite workers, the Bayesian sampling scheme has no advantage.

In the context of Fig. 1.3, this section focused on a scheme to encode query priority with more than one worker. However, each query response takes a long time to generate as one it requires training a model for some number of epochs. Now, let's focus on a direct method to accelerate ML model training with a system for distributed computation (which could be Dask, and is in Section 3.2).

3 ACCELERATING MODEL UPDATES

This section will focus on accelerating the model training portion of the active learning data flow in Fig. 1.3. In some applications, this model training may be simple. For example, algorithms used in the TNY Caption Contest in Chapter 1 only require an empirical mean estimation and a high probability precision calculation [159]. However, in many other applications the task of model training is empirical risk minimization (ERM) (e.g., [63, 91, 119, 133, 143]), which tends to be more complex. ERM takes this form:

$$\min_{\boldsymbol{w}} F(\boldsymbol{w}) = \sum_{i=1}^{n} f_i(\boldsymbol{w}) \tag{3.1}$$

where $f_i(\boldsymbol{w}) = f(\boldsymbol{w}; \boldsymbol{z}_i)$ where $\boldsymbol{z}_i$ is the ith example for some function f that is the same for all examples. In the case of least squares with a linear model with d features, $\boldsymbol{z}_i = (\boldsymbol{x}_i, y_i) \in \mathbb{R}^{d\times 1}$ and $f(\boldsymbol{w})_i = (y_i - \langle \boldsymbol{w}, \boldsymbol{x}_i \rangle)^2$. ERM is also used in active ML with the popular "follow-the-leader" framework where $n-1$ examples have been received at iteration/period n [61].

This chapter will be focused on methods to reduce the wall-clock time required to perform this optimization while not increasing the final cost. In the era of "big data," one popular method to reduce the *amount* of computation is to use stochastic gradient descent (SGD) or a variant thereof [173]. Reducing the *amount* of computation is a good way to reduce the *time* required for optimization if a single processor is used. The core computation in the iterative process of SGD takes the following form:

$$\boldsymbol{w}_{k+1} = \boldsymbol{w}_k - \gamma_k \cdot \widehat{\boldsymbol{g}}_{k,B} \tag{3.2}$$

where $\widehat{\boldsymbol{g}}_{k,B}$ is a gradient estimate of F that uses B training examples at iteration k (or modification thereof),[1] and the step size or learning rate

[1] It's common in SGD variants to use a linear combination of the model's gradient $\widehat{\boldsymbol{g}}_{k,B}$ and previous model $\boldsymbol{w}_{k-1}$ in place of $\widehat{\boldsymbol{g}}_{k,B}$ [173]. If the gradient estimate

$\gamma > 0$. For vanilla mini-batch SGD, $\widehat{\boldsymbol{g}}_{k,B} = {}^1\!/\!_B \cdot \sum_{i=1}^{B} \nabla f_{i_s}(\boldsymbol{x})$. The analysis in this chapter will use the vanilla SGD update and be largely agnostic to the gradient structure.[2]

A very simple method of acceleration is to use the SGD-style update in Eq. (3.2) and task each of P worker machines with the computation of B/P example gradients. This will require evaluation of B/P examples on each of the P machines, a common parallelization strategy with popular implementations in Tensorflow, PyTorch, Keras and Apache Spark [1, 39, 96, 110, 146, 155].[3] In this context, two parameters control the time to train any machine learning model:

1. The *number* of model updates.
2. The *time* required for each model update.

If model update i takes t_i seconds and there are N updates, then the time for model training will be $\sum_{i=1}^{N} t_i$, or tN if every model update takes t seconds. With that, two questions become relevant:

1. *How does changing the batch size B influence the optimization?* The gradient estimate of F is computed with B examples. Does changing B reduce the number of model updates?

is changed, the underlying computation doesn't change drastically (instead of evaluating $\nabla f_{i_s}(\boldsymbol{w}_k)$, $\nabla f_{i_s}(\boldsymbol{w}_k + \boldsymbol{a}_k)$ is evaluated for some vector $\boldsymbol{a}_k$ [173]).

[2]e.g., sparsity assumptions will not be required as in Recht et al. [118].

[3]It's certainly possible to distribute the minibatch SGD computation onto P different machines. However, by Amdahl's Law this does not mean the computation will be $P\times$ faster because communication and coordination are required [8]. Several studies have shown the speedups from these methods are far from optimal and saturate with a low number of workers [46, 128] (A good visualization of the work by Qi et al. [128] is at `https://talwalkarlab.github.io/paleo/` when "strong scaling" is selected). Even asynchronous methods that rely on a particular gradient structure (sparsity) exhibit very moderate accelerations with common hardware [118] (unless rare and radically different hardware is used with small modifications [185]).

2. *How can the model update time be reduced with a distributed system?* Of course, in a distributed system, the number of workers is a free parameter and gradients must be communicated between those machines for ML model training. How should the workers communicate, and how should the number of workers relate to the batch size B?

These questions will be addressed in Sections 3.1, 3.2 and 3.3. The total amount of computation is still relevant in the age of "big data" because otherwise the amount of energy/money consumed by computation quickly becomes infeasible. That means an ideal solution would minimize the total amount of computation *and* minimize the training time while retaining the same model performance. The free variables of batch size and workers in a distributed system will mean that both can be achieved simultaneously, as detailed in Sections 3.1 and 3.2. Section 3.3 details an efficient communication scheme for the not uncommon case when many model updates are required to complete distributed model training. Specifically, the conclusions of Sections 3.1, 3.2 and Section 3.3 are respectively listed below:

1. When the batch size grows in a particular manner, few model updates are required compared to SGD, and the total amount of computation does not grow.
2. The time required for model training is proportional to the number of model updates when the number of workers in a distributed system grows with the batch size.
3. The time spent while training a ML model with a distributed system is reduced when a particular type of lossy gradient compression is performed on the gradients. Many of the popular existing schemes for lossy gradient compression fit into this framework.

The conclusions of Sections 3.1 and 3.2 are particularly powerful when paired together. The conclusions are again listed here for because there is a

considerable amount of context/overhead before getting to the highlight of each section.

3.1 Improving the convergence of stochastic gradient descent (SGD) via adaptive batch sizes

The main computational bottleneck in training ML models is computing the gradient estimate $\widehat{\boldsymbol{g}}_{k,B}$ in Eq. (3.2), which uses B training data to estimate F's gradient. With a computational budget of evaluating the gradients for C examples, how should the computation be distributed? Should every model update use the same number of examples, or should the last updates compute the gradients for $C/2$ examples?

Poorly initialized models provide a useful piece of intuition to begin answering this question. In that case, evaluating the gradient estimate with 32 examples will have some error. That error will not change much when the gradient estimate is evaluated with 1000 examples because the gradient estimate doesn't change significantly – for any set of examples, the optimal model for any particular example in that set is in a similar direction. An illustration of this intuition is in Fig. 3.1, and is formalized by examining how "gradient diversity" [176] is influenced by distance to optimal solution in Appendix B.1.

Appendix B.1 suggests that the batch size should *increase* throughout an optimization if the goal is to improve the convergence of SGD, aka reduce the number of model updates. That is, the static batch size B in Eq. (3.2) should become the variable B_k and change each iteration. This section expands upon that idea by adaptively growing the batch size with model performance[4] as the optimization proceeds. Specifically, this work does the following:

- Provides methods to *adapt the batch size* to the model performance.

[4] "Model performance" defined as the objective function loss over the entire training set for convex and strongly-convex functions.

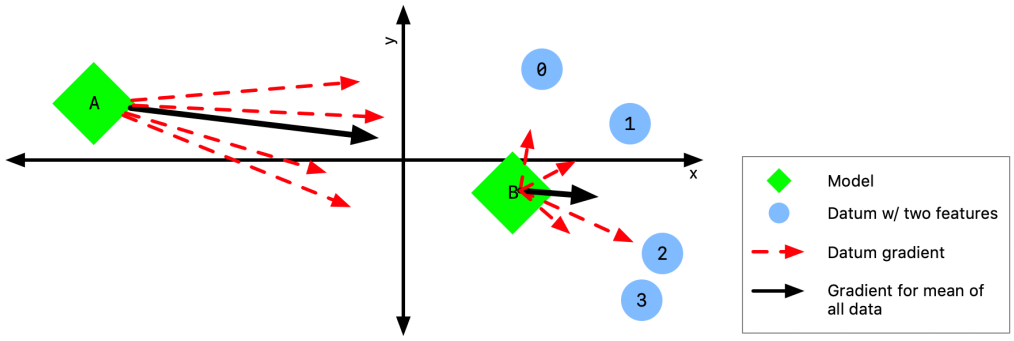


Figure 3.1: An illustration of why the batch size should increase: the gradients for each example are more "diverse" when the model is close to the optimal model. Here, the optimal model is $\boldsymbol{w} = [w_x, w_y]$ and minimizes the function $f(w_x, w_y) = \sum_{i=0}^{3}(w_x - x_i)^2 + (w_y - y_i)^2$ where x_i and y_i are the x and y coordinates of each datum. For model `B`, an estimate of the optimal model estimate, the gradients are more "diverse," so the magnitude and orientation of each datum's gradient varies more.

These methods require significant computation because they require computing model performance before every model update.

- Shows that adapting the batch size to the model performance can require *significantly fewer model updates* and approximately *the same number of gradient computations*[5] when compared with standard SGD.[6]
- Provides a simulation indicating that increasing the batch size requires far fewer model updates than SGD.
- Details a practical implementation, and illustrates performance with some experiments.

The benefit of reducing the number of model updates isn't apparent at first glance. The main benefit is that the wall-clock time required for any one model update is agnostic to the batch size with a certain distributed

[5] At least for convex and strongly-convex functions.

[6] Note that our adaptive method receives the function value in addition to the gradient, which is more information than SGD and variants thereof receive [114].

system configuration[7] [58, Sec. 5.5] (which will become particularly relevant in Section 3.2). When the batch size grows geometrically, the number of model updates is a "meaningful measure of the training time" in a similar system [152, Sec. 5.4]. Additionally, larger batch sizes improve distributed system performance [128, 176].[8]

To precisely show the points above, let's begin by showing related work and preliminary notions are introduced in Section 3.1.1 and 3.1.2 respectively. Then, let's present the adaptive batch size method and the convergence results in Section 3.1.3. This has some practical limitations, which are addressed and evaluated in Section 3.1.4. The benefit growing the batch size will be delayed until Section 3.2.

3.1.1 Related work

Mini-batch SGD with small batch sizes tends to bounce around the optimum because the gradient estimate has high variance – the optimum depends on *all* examples, not a few examples. Common methods to circumvent this issue include some step size decay schedule [21, Sec. 4] and averaging model iterates with averaged SGD (ASGD) [125]. Less common methods include stochastic average gradient (SAG) and stochastic variance reduction (SVRG) because they present memory and computational restrictions respectively [77, 142]. Our work is more similar in spirit to variance reduction techniques that use variable learning rates and batch sizes, discussed below.

Adaptive learning rates Adaptive learning rates or step sizes can help adapt the optimization to the most informative features with AdaGrad [49, 169] or to estimate the first and second moments of the gradients with Adam [85]. AdaGrad has inspired Adadelta [181] which makes some modifi-

[7] Specifically when the number of workers is proportional to the batch size

[8] See `https://talwalkarlab.github.io/paleo/` with "strong scaling" (which keeps the batch size constant regardless of the number of workers).

cations to average over a certain window and approximate the Hessian. Such methods are useful for convergence and a reduction in hyperparameter tuning.[9] AdaGrad and variants thereof give principled, robust ways to vary the learning rate that avoid having to tune learning rate decay schedules [169].

Constant batch sizes Using moderately large batch sizes yields high quality results more quickly and, in practice, requires no more computation than small batch sizes, both empirically [58] and theoretically [176]. Large constant batch sizes present generalization challenges [36, 58], hypothesized to come from convergence to a "sharp" minima, strongly influenced by the learning rate and noise in the gradient estimate [83]. To match performance on the training dataset, careful thought about choice of hyperparameters is required [58, Sec. 3 and 5.2]. In fact, this has motivated algorithms specifically designed for large constant batch sizes and distributed systems [75, 78, 178], which generally require fewer model updates to obtain similar solutions to SGD.

There are many methods to choose the best constant batch size (e.g., [53, 84]). Some methods are data dependent [176], and others depend on the model complexity. In particular, one method uses hardware topology (e.g., network bandwidth) in a distributed system [124].

Increasing batch sizes Increasing the batch size as an optimization proceeds is another method of variance reduction. Strongly convex functions provably benefit from geometrically increasing batch sizes in terms of the number of model updates while requiring no more gradient computations than SGD [20, Ch. 5]. The number of model updates required for strongly convex, convex and non-convex functions is improved with batch sizes that

[9] The original work on SGD stated that the learning rate should decay to meet some conditions, but did not specify the decay schedule [136].

increase like $\mathcal{O}\left(r^k\right)$, $\mathcal{O}\left(k^2\right)$ and $\mathcal{O}\left(k\right)$ respectively [186].[10]

Smith et al. perform variance reduction by geometrically increasing the batch size or decreasing the learning rate by the same factor, both in discrete steps (e.g., every 60 epochs) [152]. Specifically, Smith et al. motivate their method by connecting variance reduction to simulated annealing, in which reducing the SGD model update variance or "noise scale" in a series of discrete steps enhances the likelihood of reaching a "robust" minima [152, Sec. 3]. Smith et al. show that increasing the batch size yields similar results to decaying the learning rate by the same amount, which suggests that "it is the noise scale which is relevant, not the learning rate" [152, Sec. 5.1]. By that analogy, adaptive batch sizes are to geometrically increasing batch sizes as adaptive learning rate methods are to SGD learning rate decay schedules.

Adaptive batch sizes Several schemes to adapt the batch size to the model have been developed, ranging from model specific schemes [120] to more general schemes [11, 26, 42]. These methods tend to look at the sample variance of every individual gradient, which involves the computation of a single gradient norm $\|\nabla f_i(\boldsymbol{w})\|$ for every example i in the current batch [11, 26, 42]. Naively, this requires feeding every example through the model *individually*. This can be circumvented; Balles et al. present an approximation method to avoid the variance estimation that requires about $1.25\times$ more computation than the standard mini-batch SGD update, with some techniques to avoid memory constraints [11, Sec. 4.2].

Friedlander et al. use adaptive batch sizes to prove linear convergence for strongly convex functions and a $\mathcal{O}\left(1/k\right)$ convergence rate for convex functions [52]. Their adaptive approach relies on providing a batch size that satisfies certain error bounds on the gradient residual (in Eq. 2.6), which provides motivation for geometrically increasing batch sizes [52, Sec. 3].

[10] In HSGD, convex functions require $\mathcal{O}\left(\varepsilon^{-3}\right)$ gradient computations [186, Cor. 2]. As illustrated in Table 3.1, this work and SGD require $\mathcal{O}\left(\varepsilon^{-2}\right)$ gradient computations.

Work developed concurrently with this work includes an SVRG modification [74], which involves modifying the outer-loop of SVRG. Instead of calculating the gradient for all n examples during every loop, they propose a scheme to calculate the gradient for N_s examples where N_s is inversely proportional to the average gradient variance.[11]

3.1.2 Preliminaries

First, some basic definitions:

Definition 1. A function $F : \mathbb{R}^d \to \mathbb{R}^k$ is L-Lipschitz if $\|F(\boldsymbol{w}_1) - F(\boldsymbol{w}_2)\|_2 \leq L \|\boldsymbol{w}_1 - \boldsymbol{w}_2\|_2$ for all $\boldsymbol{w}_1, \boldsymbol{w}_2 \in \mathbb{R}^d$.

The norm $\|\boldsymbol{x}\|_2$ refers to the Euclidean norm of $\boldsymbol{x}$. From here on out, $\|\boldsymbol{x}\| := \|\boldsymbol{x}\|_2$.

Definition 2. A function $F : \mathbb{R}^d \to \mathbb{R}$ is β-smooth if the gradients ∇F are β-Lipschitz, or if $\|\nabla F(\boldsymbol{w}_1) - \nabla F(\boldsymbol{w}_2)\| \leq \beta \|\boldsymbol{w}_1 - \boldsymbol{w}_2\|$ for all $\boldsymbol{w}_1, \boldsymbol{w}_2 \in \mathbb{R}^d$.

The class of β-smooth functions is a result of the gradient norm being bounded, or that all the eigenvalues of the Hessian are smaller than β. If a function F is β-smooth, the function also obeys $\forall \boldsymbol{w}_1, \boldsymbol{w}_2,\ F(\boldsymbol{w}_1) \leq F(\boldsymbol{w}_2) + \langle \nabla F(\boldsymbol{w}_2), \boldsymbol{w}_1 - \boldsymbol{w}_2 \rangle + \frac{\beta}{2} \|\boldsymbol{w}_1 - \boldsymbol{w}_2\|_2^2$ [24, Lemma 3.4].

Definition 3. A function $F : \mathbb{R}^d \to \mathbb{R}$ is α-strongly convex if $\forall \boldsymbol{w}_1, \boldsymbol{w}_2,\ F(\boldsymbol{w}_1) \geq F(\boldsymbol{w}_2) + \langle \nabla F(\boldsymbol{w}_2), \boldsymbol{w}_1 - \boldsymbol{w}_2 \rangle + \frac{\alpha}{2} \|\boldsymbol{w}_1 - \boldsymbol{w}_2\|_2^2$.

α-strongly convex functions grow quadratically away from the optimum $\boldsymbol{w}^\star = \arg\min_{\boldsymbol{w}} F(\boldsymbol{w})$ since $F(\boldsymbol{w}) - F(\boldsymbol{w}^\star) \geq \frac{\alpha}{2} \|\boldsymbol{w} - \boldsymbol{w}^\star\|_2^2$. While amenable to analysis, this criterion is often too restrictive. The Polyak- Łojasiewicz

[11] In later revisions of their work, they provide a comparison with this work, which includes a similar proof to Theorem 5 [74, Appendix D].

condition is a generalization of strong convexity that's less restrictive [80, 126]:

Definition 4. A function $F : \mathbb{R}^d \to \mathbb{R}$ obeys Polyak-Łojasiewicz (PL) condition with parameter $\alpha > 0$ if $\frac{1}{2} \|\nabla F(\boldsymbol{w})\|_2^2 \geq \alpha(F(\boldsymbol{w}) - F^\star)$ when $F^\star = \min_{\boldsymbol{w}} F(\boldsymbol{w})$.

For simplicity, we refer to these functions F satisfying this condition as being "α-PL". The class of α-PL functions includes α-strongly convex functions and a certain class non-convex functions [80]. One important constraint of α-PL functions is that every stationary point must be a global minimizer, though stationary points are not necessarily unique. Recent work has shown similar convergence rates for α-PL and α-strongly convex functions for a variety of different algorithms [80].

A bound on the expected gradient norm will also be useful because it will appear in theorem statements. For ease of notation, the definition $f_i(\boldsymbol{w}) := f(\boldsymbol{w}; \boldsymbol{z}_i)$ will continue to be used.

Definition 5. For model $\boldsymbol{w}$, let $M^2(\boldsymbol{w}) := {}^1\!/_n \sum_{i=1}^n \|\nabla f_i(\boldsymbol{w})\|_2^2$ and let $\mathcal{M} := \{M^2(\boldsymbol{w}_k) : k \in \{0, 1, \ldots, T-1\}\}$ when T model updates are performed. Define $M_L^2 := \min \mathcal{M}$ and $M_U^2 := \max \mathcal{M}$.

3.1.3 Convergence

In this section we will prove convergence rates for mini-batch SGD with adaptive batch sizes and give bounds on the number of gradient computations needed. Our main results are summarized in Table 3.1. In general, we show that mini-batch SGD with appropriately chosen adaptive batch sizes converges as quickly as gradient descent in terms of the number of model updates required, but does not require more total gradient computations than serial SGD (up to constants).[12]

[12]In this theoretical discussion, the batch size will unrealistically not require any computation and be provided by an oracle. Some methods to workaround

Table 3.1: The number of model updates or gradient computations required to reach a model of error at most ϵ. All function classes are β-smooth, and for α-strongly convex functions the condition number κ is given by $\kappa = \beta/\alpha$. The function class column in Table 3.1a is shared with Table 3.1b. Error is defined with loss $F(\boldsymbol{w}_T) - F^\star \leq \epsilon$ for smooth & convex functions and α-strongly convex (α-SC) functions, and with gradient norm for smooth functions, $\min_{k=0,\ldots,T-1} \|\nabla F(\boldsymbol{w}_k)\| \leq \epsilon$. See Section 3.1.3 for details and references. Notably, the batch size is provided by an oracle for smooth functions. Cells with minimum model updates/gradient computations (up to constants) with $n = 60 \cdot 10^3$, $\varepsilon = 0.01$, $\alpha = 0.1$ and $\beta = 1$ are highlighted.

Function class	**SGD**	**Adaptive batch sizes**	**Gradient descent**
α-SC	$\mathcal{O}\left(\kappa/\beta\epsilon\right)$	$\mathcal{O}\left(\kappa \log(1/\epsilon)\right)$	$\mathcal{O}\left(\kappa \log(1/\epsilon)\right)$
Convex	$\mathcal{O}\left(1/\epsilon^2\right)$	$\mathcal{O}\left(1/\epsilon\right)$	$\mathcal{O}\left(1/\epsilon\right)$
Smooth	$\mathcal{O}\left(1/\epsilon^4\right)$	$\mathcal{O}\left(1/\epsilon^2\right)$	$\mathcal{O}\left(1/\epsilon^2\right)$

(a) Total number of **model updates** required during optimization.

Function class	**SGD**	**Adaptive batch sizes**	**Gradient descent**
α-SC	$\mathcal{O}\left(\kappa/\beta\epsilon\right)$	$\mathcal{O}\left(\kappa/\epsilon \log(1/\epsilon)\right)$	$\mathcal{O}\left(n\kappa \log(1/\epsilon)\right)$
Convex	$\mathcal{O}\left(1/\epsilon^2\right)$	$\mathcal{O}\left(1/\epsilon^2\right)$	$\mathcal{O}\left(n/\epsilon\right)$
Smooth	$\mathcal{O}\left(1/\epsilon^4\right)$	$\mathcal{O}\left(1/\epsilon^3\right)$	$\mathcal{O}\left(n/\epsilon^2\right)$

(b) Total number of **gradient computations** required during optimization.

In general, the adaptive batch sizes are inversely proportional to the current model's loss. This method is motivated by an approximate measure of gradient dissimilarity as detailed in Appendix B.1. These adaptive batch sizes B_k are computed with the current model $\boldsymbol{w}_k$ and use the model update in Eq. 3.2 to produce a new model $\boldsymbol{w}_{k+1}$.

Section 3.1.3 analyzes the required number of model updates, and analyzes the required number of gradient computations. The theory in this section might require significant computation; methods in Section 3.1.4 circumvent some of these issues.

this unrealistic assumptions are presented in Section 3.1.4.

3.1.3.1 Model updates

Let's start in the context of α-PL functions. In this setting, SGD requires $\mathcal{O}(1/\varepsilon)$ model updates [80, Thm. 4]. Gradient descent with a constant learning rate requires $\log(1/\varepsilon)$ model updates [80, Thm. 1], as does SGD with geometrically increasing batch sizes for strongly convex functions [20, Cor. 5.2]. We show that $\log(1/\varepsilon)$ model updates are also required when the adaptive batch size is chosen appropriately:

Theorem 2. *Let $\boldsymbol{w}_k$ denote the k-th iterate of mini-batch SGD with step-size γ on a β-smooth and α-PL function F. If the batch size B_k at each iteration k is given by*

$$B_k = \left\lceil \frac{c}{F(\boldsymbol{w}_k) - F^\star} \right\rceil \tag{3.3}$$

and the learning rate $\gamma = \alpha/[\beta(\alpha + M_U^2/2c)]$ for some constant $c > 0$, then

$$\mathbb{E}[F(\boldsymbol{w}_T)] - F^\star \leq (1-r)^T (F(\boldsymbol{w}_0) - F^\star)$$

where $r := \alpha^2/(\beta(\alpha + M_U^2/2c))$. This implies $T \geq \mathcal{O}(\log(1/\varepsilon))$ model updates are required to obtain $\boldsymbol{w}_T$ such that $\mathbb{E}[F(\boldsymbol{w}_T)] - F^\star \leq \varepsilon$.

The proof is detailed in Appendix B.2.1 and follows from the definition of B_k, β-smooth and α-PL. This theorem can also be applied to Euclidean distance from the optimal model for α-strongly convex functions because $\alpha/2 \|\boldsymbol{w}_k - \boldsymbol{w}^\star\|_2^2 \leq F(\boldsymbol{w}_k) - F(\boldsymbol{w}^\star)$. The learning rate γ is typically a user-specified hyperparameter determined through trial-and-error (e.g, [141, 151]). This theorem makes a fairly standard assumption that the optimal training loss $F^\star$ is known, which influences γ by Ward et al. [169, Sec. 1.1] and Orr [120, Eq. 15].[13]

[13] For most overparameterized neural nets, the optimal training loss is (approximately) 0 [12, 140, 182].

When F is convex, the same adaptive batch size method obtains comparable convergence rates to gradient descent. Gradient descent with constant learning rate requires $\mathcal{O}\left(1/\varepsilon\right)$ model updates [24, Thm. 3.3], and has linear convergence if an exact line search is used [22, Eq. 9.18]. SGD requires $\mathcal{O}\left(1/\varepsilon^2\right)$ model updates [24, Thm. 6.3]. Using adaptive batch sizes with SGD also requires $\mathcal{O}\left(1/\varepsilon\right)$ model updates:

Theorem 3. *Let $\boldsymbol{x}_k$ denote the k-th iterate of mini-batch SGD with step size γ on some β-smooth and convex function F. If the batch size B_k at each iteration is given by Equation 3.3 and $\gamma = (\beta + 1/c)^{-1}$, then for any $T \geq 1$,*

$$\mathbb{E}\left[F\left(\overline{\boldsymbol{w}}_T\right)\right] - F^\star \leq \frac{r}{T}$$

where $r := \|\boldsymbol{w}_0 - \boldsymbol{w}^*\|^2 \left(\beta + \frac{M_U^2}{c}\right) + F(\boldsymbol{w}_0) - F^*$ and $\overline{\boldsymbol{w}}_T := \frac{1}{T}\sum_{i=0}^{T-1} \boldsymbol{w}_{i+1}$. This implies $T \geq r/\varepsilon$ model updates are required to obtain $\boldsymbol{w}_T$ such that $\mathbb{E}\left[F(\boldsymbol{w}_T)\right] - F^\star \leq \varepsilon$.

This is proved in Appendix B.2.2, which is an adaptation of the classic SGD convergence analysis [24].

Key Lemma Theorems 2 and 3 rely on a key lemma, one that controls the gradient approximation error $\mathbb{E}\left[\|\nabla F(\boldsymbol{w}_k) - \widehat{\boldsymbol{g}}_k\|\right]$ as a function of the number of model updates and the loss for a given gradient approximation $\widehat{\boldsymbol{g}}_k$ and the adaptive batch sizes B_k:

Lemma 4. *Let the batch size B_k be chosen as in Eq. 3.3. Then when the gradient estimate $\widehat{\boldsymbol{g}}_k = 1/B_k \sum_{i=1}^{B_k} \nabla f_{i_s}(\boldsymbol{w}_k)$ is created with i_s chosen uniformly at random, then the expected gradient error*

$$\mathbb{E}\left[\|\nabla F(\boldsymbol{w}_k) - \widehat{\boldsymbol{g}}_k\|_2^2 \;\middle|\; \boldsymbol{w}_k\right] \leq (F(\boldsymbol{w}_k) - F^\star) M_U^2 c^{-1}$$

The proof is Appendix B.2 and relies on substituting the definition of the batch size B_k into the gradient approximation error, $\mathbb{E}\left[\|\nabla F(\boldsymbol{w}_k) - \widehat{\boldsymbol{g}}_k\|_2^2\right]$.

Lemma 4 is used to factor the approximation error $F(\boldsymbol{w}_k) - F^\star$ out of both terms in the upper bound in the proof of Theorem 2, which allows for linear convergence. It's used in a similar manner in Theorem 3, specifically to obtain multiples of the same term on both sides of the inequality.[14]

When F is smooth and non-convex, we'll provide an upper bound on the number of model updates required to find an ϵ-approximate critical point so that $\|\nabla F(\boldsymbol{w})\| \leq \epsilon$, which requires computing the adaptive batch size differently. In this setting, SGD requires $\mathcal{O}\left(1/\epsilon^4\right)$ model updates [176, Thm. 2], and gradient descent requires $\mathcal{O}\left(1/\epsilon^2\right)$ model updates [76, Thm. 2]. Adaptive batch sizes require $\mathcal{O}\left(1/\varepsilon^2\right)$ model updates:

Theorem 5. *Let $\boldsymbol{w}_k$ denote the k-th iterate of mini-batch SGD on a β-smooth function F. If the batch size B_k at each iteration satisfies*

$$B_k = \left\lceil \frac{c}{\|\nabla F(\boldsymbol{w}_k)\|_2^2} \right\rceil \tag{3.4}$$

for some $c > 0$ and the step size $\gamma = \beta^{-1} \cdot c/(c + M_L^2)$, then for any $T \geq 1$,

$$\min_{k=0,\ldots,T-1} \|\nabla F(\boldsymbol{w}_k)\| \leq \sqrt{\frac{r}{T}}$$

where $r \coloneqq 2(F(\boldsymbol{w}_0) - F^\star) \cdot \beta\left(M_L^2 c^{-1} + 1\right)$. This implies $T \geq r^2/\varepsilon^2$ model updates are required to obtain $\boldsymbol{w}_T$ such that $\min_{k=0,\ldots,T-1} \|\nabla F(\boldsymbol{w}_k)\| \leq \varepsilon$.

This theorem is proved in Appendix B.2.3. The proof adapts the proof of Theorem 2 by Yin et al. [176] to the batch size in Eq. (3.4) and .

3.1.3.2 Number of gradient computations

The convergence rates above show that adaptively chosen batch sizes can lead to fast convergence in terms of the number of model updates – which is great if the time spent training is proportional to the number of model

[14] Theorem 5 relies on a similar procedure.

updates.[15] However, in this case, the number of model updates doesn't characterize the total amount of work performed: when the model is close to the optimum, the batch size will be large but only one model update will be computed. A better metric for the amount of work performed is on the number of gradient computations required to reach a model of a particular error, or the sum of the batch sizes $\sum_{i=0}^{T-1} B_i$.

In short, the number of gradient computations $\sum_{i=0}^{T-1} B_i$ that are required by the adaptive batch size method is similar to the number of gradient computations for SGD. A comparison is concisely summarized in Table 3.1.[16] In this table, line searches are not performed for gradient descent on convex functions.

Let's start with α-PL and convex functions. When increasing the batch size geometrically for α-strong convex functions, only $\mathcal{O}(1/\epsilon)$ gradient computations are required [20, Thm. 5.3].

Corollary 6. *For an α-PL function F, no more than $4cr \log(1/\epsilon)/\epsilon$ gradient computations are required in Theorem 2 where c and r are defined in Theorem 2.*

Corollary 7. *For a convex and β-smooth function F, no more than $4cr/\epsilon^2$ gradient computations are required in Theorem 3 where c and r are defined in Theorem 3.*

Now, let's look at the gradient computations required for smooth functions. For illustration, let's assume the batch size in Eq. 3.4 is given by an oracle and does not require any gradient computation.

[15] Which is possible in Section 3.2, even with batch size growth.

[16] The number of gradient computations for SGD and gradient descent are reflected in the model update count; SGD and gradient descent require computing 1 and n gradients per model update respectively.

Corollary 8. *For β-smooth functions F, no more than $4cr/\epsilon^3$ gradient computations are required to estimate the loss function's gradient in Theorem 5 where c and r are defined in Theorem 5.*

Proof is delegated to Appendix B.3. Corollaries 6, 7 and 8 rely on Lemma 18, which is not tight. Tightening this bound requires finding *lower* bounds on model loss, a statement of the form $F(\boldsymbol{w}_k) - F^\star \geq g(\epsilon, k)$ for some function g. There are classical bounds of this sort for gradient descent [115, Thms. 2.1.7 and 2.1.13], and more recent lower bounds for SGD [117]. However, deriving a comprehensive understanding of lower bounds for mini-batch SGD remains an open problem.

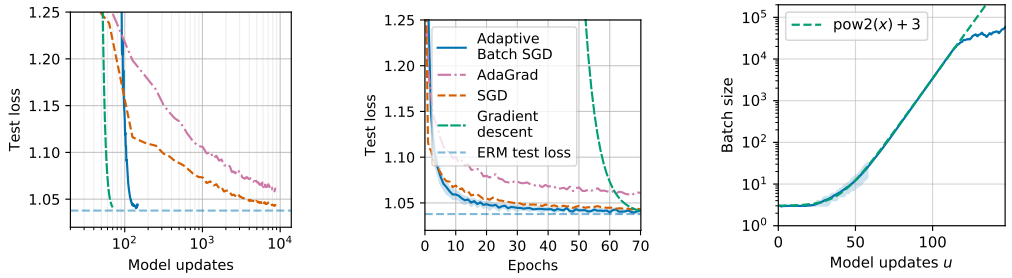
3.1.4 Experiments

In this section, we first show that the theory above works as expected: far fewer model are required to obtain a model of a particular loss, and the total number of gradient computations is the same as standard mini-batch SGD. However, the implementation above is impractical: the batch size requires significant computation. We suggest some workarounds to address these practical issues, and provide experiments that compare the proposed method with relevant work.[17]

3.1.4.1 Synthetic simulations

First, let's train a neural network with linear activations to illustrate our theoretical contributions. Practically speaking, this is an extremely inefficient and roundabout way to compute a linear function. However, the associated loss function is non-convex and more difficult to optimize. Despite the non-convexity it satisfies the PL inequality almost everywhere in a measure–theoretic sense [32, Thm. 13]. This section will focus on this optimization:

[17] These experiments are available at `https://github.com/stsievert/adadamp-experiments`.



(a) The number of model updates required to reach a particular test loss.

(b) The number of epochs required to be processed to reach a particular test loss.

(c) The batch sizes and an exponential line. In the legend, $x = 0.17(u - 31)$ for u model updates.

Figure 3.2: Different performance metrics for different optimizers for the minimization in Section 3.1.4.1. The legend in Figure 3.2b is shared with Figures 3.2a and 3.2c, and the "ERM test loss" is the test loss of the linear ERM solution. The solid lines represent the mean over 50 runs, and the shaded region represent the interquartile range.

$$\widehat{\boldsymbol{w}}_1, \widehat{\boldsymbol{W}}_2, \widehat{\boldsymbol{W}}_3 = \arg \min_{\boldsymbol{w}_1, \boldsymbol{W}_2, \boldsymbol{W}_3} \sum_{i=1}^{n} \left(y_i - \boldsymbol{w}_1^T \boldsymbol{W}_2 \boldsymbol{W}_3 \boldsymbol{x}_i \right)^2 \tag{3.5}$$

where there are $n = 10^4$ observations and each feature vector has $d = 100$ dimensions, and $\boldsymbol{w}_1 \in \mathbb{R}^d$, $\boldsymbol{W}_2, \boldsymbol{W}_3 \in \mathbb{R}^{d,d}$. We generate synthetic data $\boldsymbol{x}_i$ with coordinates drawn independently from $\mathcal{N}(0, 1)$. Each label y_i is given by $y_i = \boldsymbol{x}_i^T \boldsymbol{w}^\star + n_i$ where $n_i \sim \mathcal{N}(0, d/100)$ and $\boldsymbol{w}^\star \sim \mathcal{N}(0, 1)$. Of the $n = 10^4$ observations, $2,000$ observations are used as test data.

In order to understand our adaptive batch size method, we compare the model updates in Theorem 2 (aka "Adaptive Batch SGD") with mini-batch SGD to standard mini-batch SGD with decaying step size (SGD), gradient descent and AdaGrad. The hyperparameters for these optimizers are not tuned and details are in Appendix B.4.1. AdaGrad and SGD are run with batch size $B = 64$.

Figure 3.2 shows that Adaptive Batch SGD requires far fewer model

updates, not far from the number that gradient descent requires. Adaptive Batch SGD and SGD require nearly the same number of data, with AdaGrad requiring more data than SGD but far less than gradient descent. Figure 3.2c shows that the batch size grows nearly exponentially, very similar to the passive batch size growth of Bottou et al. [20, Eq. 5.7].

3.1.4.2 Functional implementation

A practical issue immediately presents itself: the computation of the batch size B_k. This is clearly infeasible because it requires evaluating the entire training dataset every model update.[18] To work around this issue, let's approximate the training loss with a rolling-average of batch losses, similar to other stochastic optimization algorithms [85, 181]. Additionally, generalization[19] and GPU memory concerns may be present. To address these concerns, prior work sets a maximum batch size and decays the learning rate by the same amount the batch size would have increased [48, 152]. Both actions reduces the "noise scale" or variance of the model update, and the results in Smith et al. "suggest that it is the noise scale which is relevant, not the learning rate" [152]. This additional noise decay might help with generalization escape "sharp minima" that generalize poorly [33, 83, 153]

The implementation of this algorithm is shown shown in Algorithm 1, which uses a *r*olling average to *ada*ptively *damp* the noise in the gradient estimate. This algorithm is designed with these experiments in mind, the

[18] Another method to remove this computational concern is to *passively* approximate the batch size. If the bounds in Theorem 3 characterize how the loss decreases, then the loss will decay like $1/(k+1)$ at model update k. If this is the case, then that suggests the batch size $B_k = B_0 + \lceil mk \rceil$ for some constant $m > 0$. This mirrors batch size increase in HSGD [186] for smooth functions but differs for convex functions; HSGD increases the batch size like $\mathcal{O}\left(k^2\right)$. However, HSGD requires $\mathcal{O}\left(1/\varepsilon^3\right)$ [186, Cor. 2] while the adaptive batch size scheme requires $\mathcal{O}\left(1/\varepsilon^2\right)$ gradient computations (Thm. 3).

[19] There are concerns with large static batch sizes [73, 153]; it's unclear what happens for *variable* batch sizes.

reason the batch size is inversely proportional to a linear combination of the training loss and gradient norm.

To evaluate our method, let's use a convolutional neural network on the Fashion-MNIST dataset [174] with optimization algorithms that either passively or adaptively change the learning rate or batch size. Specifically, let's compare RadaDamp with SGD, "GeoDamp" [152], and AdaGrad [49].[20] During this, let's tune the batch size increase schedule for RadaDamp/GeoDamp, and use the same schedule for the corresponding algorithms that only decay the learning rate ("RadaDamp-LR" and SGD respectively). Details are in Appendix B.4.

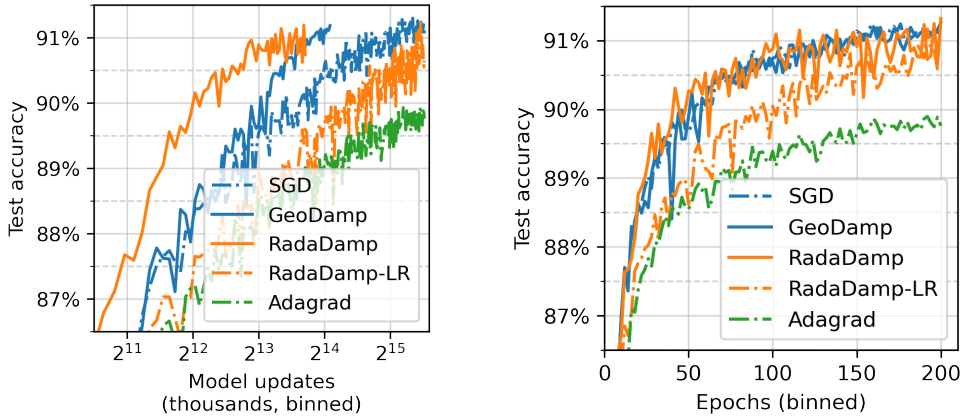
Algorithm 1 RadaDamp(step size γ, memory $\rho = 0.999$, initial batch size B_0, max. batch size $B_{\max}$, model $\boldsymbol{w}_0$, regularization $\lambda = 10^{-3}$)

for $k \in [0, 1, 2, \ldots]$ **do**
  $\gamma' \leftarrow \gamma$
  if $B_k \geq B_{\max}$ **then**
    $\gamma' \leftarrow \gamma B_{\max}/B_k$
    $B_k \leftarrow B_{\max}$
  $\widehat{L}_B \leftarrow {}^1/_{B_k} \sum_{i=1}^{B_k} f_{i_s}(\boldsymbol{w}_k)$
  $\boldsymbol{w}_{k+1} \leftarrow \boldsymbol{w}_k - \gamma' \nabla \widehat{L}_B$
  $t_k \leftarrow \widehat{L}_B + \lambda \left\| \nabla \widehat{L}_B \right\|_2^2$
  if $k > 0$ **then**
    $\widehat{d}_k \leftarrow \rho \cdot \widehat{d}_{k-1} + (1-\rho) t_k$
  else
    $\widehat{d}_0 \leftarrow t_k$
  $B_{k+1} \leftarrow \left\lceil B_0 \widehat{d}_0 / \widehat{d}_k \right\rceil$
return $\boldsymbol{w}_{k+1}$

Our experimental results are shown in Figure 3.3. As expected, they show that RadaDamp and GeoDamp require far fewer model updates than RadaDamp-LR and SGD, and similar performance is obtained for all methods in terms of epochs.[21] If the "noise scale" of the model updates is relevant as Smith et al. hypothesize [152], then perhaps the relevant comparison is between passive and adaptive methods of changing the "noise scale" (i.e., RadaDamp is to AdaGrad as GeoDamp is to SGD).

[20] All optimizers use the same learning rate, momentum and initial/max batch size, and basic tuning on the batch size increase/learning rate decay schedule is performed

[21] With the exception of AdaGrad, possibly due to the fact that has been run with only one random seed, not two.



(a) The number of model updates vs. test accuracy.

(b) The number of epochs vs. test accuracy.

Figure 3.3: Performance on the Fashion-MNIST dataset. "Binned epochs/-model updates" means "rounded to multiple of 2/200" respectively, and the mean of relevant values is shown. Before optimization begins, all models start optimization with the same weights.

RadaDamp requires far fewer model updates than GeoDamp to reach any test accuracy RadaDamp obtains (though GeoDamp obtains a final test accuracy that is approximately 0.4% higher). Of course, Both AdaGrad and RadaDamp require far less tuning than GeoDamp and SGD because of the adaptivity to the (estimated) training loss. Figure B.1 shows that both RadaDamp and GeoDamp (approximately) increase the batch size exponentially as functions of model updates, at least initially. However GeoDamp's learning rate decays much more and far quicker than RadaDamp's (perhaps a reason for GeoDamp's increased performance).

3.1.5 Conclusion

This work presents theoretical motivation to increase the batch size, at least if few model updates are desired. At first, this is in internal benefit because it only reduces an internal variable (the number of model updates) from

SGD, not the total amount of computation (FLOPs or number of gradient computations). Simulations and experiments confirm few model updates are required.

Currently, large *static* batch sizes suffer from decreased performance on the test set due to poor generalization [58, 75, 83]. Future work involves determining how *variable* batch sizes influence this generalization gap.

Notably, GeoDamp [152] outperforms RadaDamp in terms of test accuracy. Let's use GeoDamp to show the primary benefit of increasing the batch size: in some distributed systems, the time for training is strongly correlated with the number of model updates.

3.2 Training PyTorch models faster with Dask

Training deep machine learning (ML) models takes a long time. For example, training a popular image classification model [133] to reasonable accuracy takes "around 17 hours" on Google servers.[22] Another example includes training an NLP model for 10 days on 8 high-end GPUs [129].[23] Notably, the number of floating point operations (FLOPs) required for "the largest AI training runs" doubles every 3.4 months.[24]

Increasing the batch size B_k will reduce the number of model updates while not requiring more FLOPs or gradient computations – both empirically [152] and theoretically [20, 186][150, in Sec. 3.1]. The number of FLOPs controls the cost, as expressed in Section 1.2. Typically, the number of FLOPs controls the training time because training is performed with a single processor – so fewer model updates seems like an internal benefit that doesn't affect training time.

The benefit comes when training with multiple machines, aka a distributed system because the time required to complete a single model update is (nearly) agnostic to the batch size provided the number of workers in a distributed system grows with the batch size [58, Sec. 5.5], [152, Sec. 5.4]. So, it seems that ML model training can be accelerated over standard SGD at no additional cost to the experimentalist as long as the proper distributed system is used. Let's investigate that more closely.

[22] Specifically, while training a ResNet-50 model on the ImageNet database using a Google Tensor Processing Unit (TPU) (github.com/tensorflow/tpu/.../resnet/README.md).

[23] See OpenAI's blog post "Improving Language Understanding with Unsupervised Learning."

[24] See OpenAI's blog post "AI and Compute."

3.2.1 Contributions

We provide software to accelerate ML model training, at least with certain distributed systems. For acceleration, the distributed system must be capable of assigning a different number of workers according to a fixed schedule. Specifically, this work provides the following:

- Proposes a scheme to reduce the time required for training ML models: growing the batch size and the number of workers in a distributed system.
- A Python software package to train ML models with a distributed system. Our software works on any cluster that is configured to work with Dask, many of which can change the number of workers on demand.[25] The implementation[26] provides a Scikit-learn API [25] to PyTorch models [122].
- Experiments and simulations that illustrate that proposed training scheme can effectively reduce the wall-clock time required for model training.

A key component of our software is that the number of workers grows with the batch size because then, the model update time is nearly agnostic to the batch size [58, Sec. 5.5]. We envision our software being used by a single data scientist to accelerate ML model training on Amazon AWS or on their computational cluster (e.g., a SLURM or Kubernetes cluster owned/operated by their company/institution). In that use case, the number of servers is almost transparent to the data scientist because Dask can easily request more computational resources (e.g., through Dask Cloud Provider[27]).

[25] Including the default usage (through LocalCluster), supercomputers (through Dask Job-Queue), on cloud providers like Amazon AWS (through Dask Cloud Provider), YARN/Hadoop clusters (through Dask Yarn) and Kubernetes clusters (through Dask Kubernetes).

[26] `https://github.com/stsievert/adadamp`

[27] `https://cloudprovider.dask.org/`

Our software will also likely be useful for a team of data scientists sharing a computational cluster (e.g., a bank of GPUs) for ML model training.

Now, let's cover related work to gain understanding of why variable batch sizes provide a benefit in a distributed system. Then, let's cover the details of our software before presenting simulations. These simulations confirm that model training can be accelerated if the number of workers grows with the batch size.[28]

3.2.2 Related work

The data flow for distributed model training involves distributing the computation of the gradient estimate, $\widehat{\boldsymbol{g}}_{k,B} = \frac{1}{B}\sum_{i=1}^{B} \nabla f(\boldsymbol{w}_k; \boldsymbol{z}_{i_s})$. Typically, each worker computes the gradients for B/P examples when there is a batch size of B and P machines. Then, the average of these gradients is taken and the model is updated.[29]

Clearly, Amdahl's law is relevant because there are diminishing returns as the number of workers P is increased [57]. This as referred to as "strong scaling" because the batch size is fixed and the number of workers is treated as an internal detail. By contrast, growing the amount of data with the number of workers is known as "weak scaling." Of course, relevant experiments show that weak scaling exhibits better scaling than strong scaling [128].

3.2.2.1 Constant batch sizes

To circumvent Amdahl's law, a common technique is to increase the batch size alongside the learning rate [78, 183]. Using moderately large batch sizes yields high quality results more quickly and, in practice, requires

[28] Methods to workaround limitations on the number of workers will be presented in Appendix C.2.

[29] Related but tangential methods include methods to efficiently communicate the gradient estimates [7, 60], [166, in Sec. 3.3].

no more computation than small batch sizes, both empirically [58] and theoretically [176].

There are many methods to choose the best constant batch size (e.g., [53, 84]). Some methods are data dependent [176], and others depend on the model complexity. In particular, one method uses hardware topology (e.g., network bandwidth) in a distributed system [124].

Large constant batch sizes present generalization challenges [58]. The generalization error is hypothesized to come from "sharp" minima, strongly influenced by the learning rate and noise in the gradient estimate [83]. To match performance on the training dataset, careful thought must be given to hyperparameter selection [58, Sec. 3 and 5.2]. In fact, this has motivated algorithms specifically designed for large constant batch sizes and distributed systems [75, 78, 178]. These algorithms mitigates the generalization issues with large batches [75, 78, 177].

3.2.2.2 Increasing the batch size

Much of the work in Section 3.1.1 is relevant here too, and omitted for brevity. Both growing the batch size [20, 150] and using large constant batch sizes [78, 176, 179] should require fewer model updates and the same number of floating point operations as standard SGD with small batch sizes to reach a particular training loss. Some proof techniques suggest that variable batch size methods mirror gradient descent [80, 150], so correspondingly, the implementations do not require much additional hyperparameter tuning.

3.2.3 Distributed training with Dask

We have written AdaDamp, a software package to to train a PyTorch model with a Scikit-learn API on any Dask cluster, a distributed system detailed in Section 2.3.3.[30] It supports the use of constant or variable batch sizes,

[30]While our software works with a constant batch size, the native implementations work with constant batch sizes and very likely have less overhead (e.g.,

which fits nicely with Dask's ability to change the number of workers.[31] In this section, we will walk through the basic architecture of our software and an example usage. We will defer showing the primary benefit of our software to the experimental results.

3.2.3.1 Architecture

Our software uses a centralized synchronous parameter server and controls the data flow of the optimization as illustrated in Fig. 3.4. We use Dask to implement this data flow, which adds some overhead.[32] AdaDamp supports static batch sizes; however, there is little incentive to use AdaDamp with a static batch sizes: the native solution has PyTorch less overhead [97], and already has a Dask wrapper.[33] Example usage of our software is in Appendix C.1.

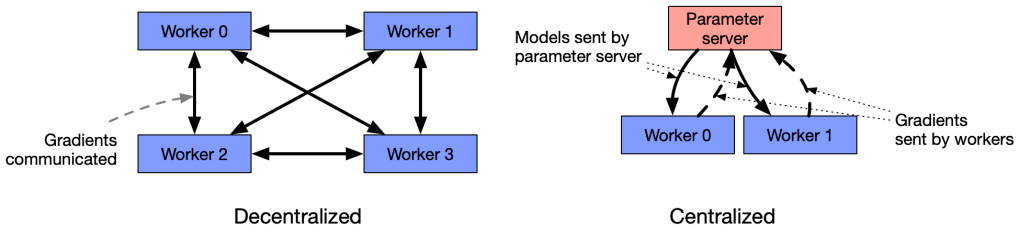


Figure 3.4: Let's say the optimization scheme dictates a batch size of $B = 1024$. For the **decentralized case** with 4 workers, each worker will compute the gradient for 256 examples. The sum of all 1024 gradients will be communicated to every worker, and each worker will perform the same model update. For the **centralized case** with 2 workers, each worker will compute the gradients for 512 examples and send the sum back to the parameter server (PS). The PS will perform the model update, then send the new model to the workers to compute another gradient.

PyTorch Distributed [97]).

[31] https://github.com/stsievert/adadamp

[32] An opportunity for future work.

[33] https://github.com/saturncloud/dask-pytorch-ddp

The key component of AdaDamp is that the number of workers grows with the batch size. There's some evidence the model update time is nearly agnostic to the batch size in this case [58, Sec. 5.5], [152, Sec. 5.4].

3.2.4 Performance

In this section, we present two sets of experiments.[34] Both experiments will use the same setup, a Wide-ResNet model in a "16-4" architecture [180] to perform image classification on the CIFAR10 dataset [90]. This is a deep learning model with about 2.75 million weights that requires a GPU to train.[35] The experiments will provide evidence for the following points:

1. Increasing the batch size reduces the number of model updates.
2. The time required for model training is roughly proportional to the number of model updates (presuming the number of workers grows with the batch size as mentioned in Section 3.2.3.1).

To provide evidence for these points, let's run one set of experiments that varies the batch size increase schedule. These experiments will mirror the experiments by Smith et al. [152]. Let's train each batch size increase schedule once, and then write the historical performance to disk. This reduces the need for many GPUs, and allows us to simulate different networks and highlight the performance of Dask. That means that in our simulations, we simulate model training by having the computer sleep for an appropriate and realistic amount of time.

[34] Full detail on these experiments can be found at `https://github.com/stsievert/adadamp-experiments`

[35] Specifically, we used a NVIDIA T4 GPU with an Amazon `g4dn.xlarge` instance. Training consumes 2.2GB of GPU memory with a batch size of 32, and 5.5GB with a batch size of 256.

3.2.4.1 Base training

First, let's show that batch size increase schedules can require fewer model updates. These experiments mirror the experiments by Smith et al. [152, Sec. 5.1], which helps reduce the parameter tuning. These experiments only differ in the choice of batch size and learning rate, as shown in Fig. 3.5.[36] In these experiments, either the learning rate is decreased or the batch size increases by a specified factor (5) at particular intervals (epochs 60, 120 and 180). This means that the variance of the model update is reduced by a constant factor at each update.

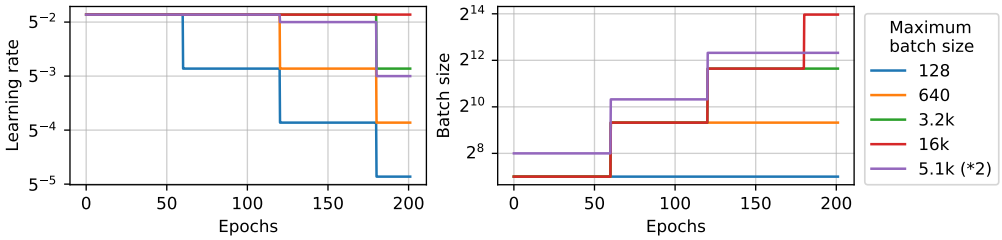


Figure 3.5: The learning rate and batch size decrease/increase schedules for various optimizers. After the maximum batch size is reached, the learning rate decays. A postfix of `(*2)` means the initial batch size twice as large (256 instead of 128). The legend applies to both plots.

These different decay schedules exhibit the same performance in terms of number of epochs, which is proportional to the number of FLOPs, as shown in Fig. 3.6. The number of FLOPs is (approximately) proportional to the cost on Amazon EC2 where the cost to rent a server tends to be proportional to the number of GPUs.[37]

[36] As in the Smith et al. experiments, every optimizer uses Nesterov momentum [115] and the same momentum (0.9) and weight decay ($0.5 \cdot 10^{-3}$). They start with the same initial learning rate (0.05), These hyperparameters are the same as Smith et al. [152] with the exception of learning rate (which had to be reduced by a factor of 2).

[37] An Amazon AWS `p3.2xlarge` machine has one state-of-the-art NVIDIA V100 GPU and can be rented for $3.06/hour. A `p3.8xlarge` has 4 NVIDIA

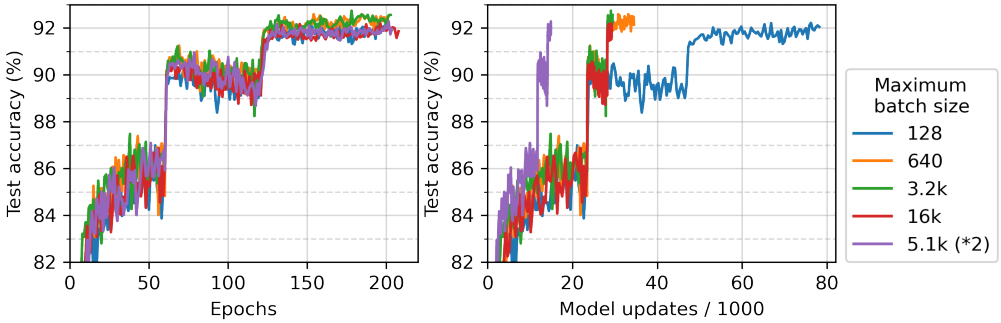


Figure 3.6: The test set performance after a certain amount of epochs/model updates. The legend applies to both plots.

3.2.4.2 Distributed training

Importantly, this work focuses on increasing the number of workers with the batch size – the effect of which is hidden in Fig. 3.6. However, the fact that the performance does not change with different schedules means that choosing a different batch size increase schedule will not require more wall-clock time if only a single worker is available. Combined with the hyperparameter similarity between the different schedules, this reduces deployment and debugging concerns.

If the number of workers grows with the batch size, then the number of model updates is relevant to the wall-clock time. Figure 3.7 shows the number of model updates and wall-clock time required to reach a model of a particular test accuracy.

3.2.5 Conclusion

In this work, we have provided a package to train ML models implemented with PyTorch ML with Dask cluster. Notably, this package reduces the amount of time required to train a model when the number of workers grows

V100s and costs $12.24/hour, exactly 4 times as much (https://aws.amazon.com/ec2/pricing/on-demand/).

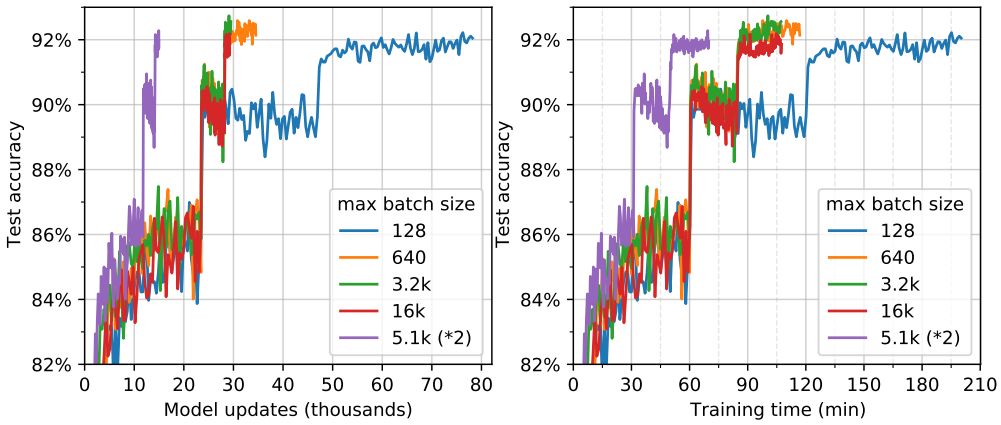


Figure 3.7: The same simulations as in Fig. 3.6, but plotted with the number of model updates and wall-clock time plotted on the x-axis (the loss obeys a similar behavior as illustrated in Appendix C.3).

with the batch size (a.k.a weak scaling). In this case, the number of model updates is strongly correlated with the time required to complete training.

However, this package presents significant overhead, and some potential solutions are discussed in Appendix C.2.[38] When performance under these improvements is simulated, only 45 minutes are required for training for a particular model – an improvement over the 120 minutes required with standard SGD. These simulations show the same relative performance as Fig. 3.7, and a summary is presented in Fig. 3.8.

[38]We hypothesize the overhead comes from the centralized architecture, which could be removed with with the integration of PyTorch's decentralized communication [97]. The (concurrently developed) wrapper for Dask is available at `https://github.com/saturncloud/dask-pytorch-ddp`.

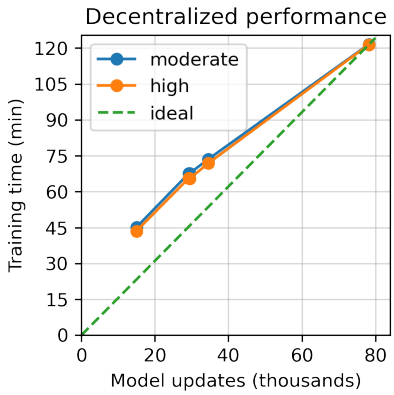


Figure 3.8: A single point represents one run in Figure C.1. The point with about 80k model updates represents a single worker, so there's no overhead in this decentralized simulation. Different network qualities are shown with different colors, and the "ideal" line is as if every model update is agnostic to batch size.

Now, let's turn our focus to the case when distributed ML model training is performed and many model updates are required. Of course, for training workers will have to communicate gradients, as illustrated in Fig. 3.4. How can gradient estimates be efficiently communicated?

3.3 Improving communication in distributed model updates

With many model updates, communication overheads are one of the main bottlenecks in distributed machine learning, and typically responsible for the speedup saturation phenomenon [46, 60, 128, 144, 156]. Communication bottlenecks are largely attributed to frequent gradient updates transmitted between compute nodes. As the number of parameters in state-of-the-art models scales to hundreds of millions [63, 66], the size of gradients scales proportionally.

To reduce the cost of of communication during distributed model training, a series of recent studies propose communicating low-precision or sparsified versions of the computed gradients during model updates. Partially initiated by a 1-bit implementation of SGD by Microsoft [144], a large number of recent studies revisited the idea of low-precision training as a means to reduce communication [7, 17, 43, 44, 45, 131, 171, 184, 187]. Other approaches for low-communication training focus on sparsification of gradients, either by thresholding small entries or by random sampling [5, 34, 93, 98, 106, 135, 156, 162]. Several approaches, including "QSGD" and "TernGrad," implicitly combine quantization and sparsification to maximize performance gains [7, 88, 89, 157, 171], while providing provable guarantees for convergence and performance. We note that quantization methods in the context of gradient based updates have a rich history, dating back to at least as early as the 1970s [6, 16, 55].

Notably, requiring fewer model updates also reduces the amount of communication because communication only happens before a model update. Reducing the number of model updates is also an attractive means to reduce the amount of communication in a ML training session, which makes Section 3.1 and references therein relevant.

3.3.1 Contributions

An atomic decomposition represents a vector as a linear combination of simple building blocks in an inner product space. In this work, we show that stochastic gradient sparsification and quantization are facets of a general approach that sparsifies a gradient in any possible atomic decomposition, including its entry-wise or singular value decomposition, its Fourier decomposition, and more. With this in mind, we develop ATOMO, a general framework for atomic sparsification of stochastic gradients. ATOMO sets up and optimally solves a meta-optimization that minimizes the variance of the sparsified gradient, subject to the constraints that it is sparse on the atomic basis, and also is an unbiased estimator of the input.

We show that 1-bit QSGD and TernGrad are in fact special cases of ATOMO, and each is optimal (in terms of variance and sparsity), in different parameter regimes. Then, we argue that for some neural network applications, viewing the gradient as a concatenation of matrices (each corresponding to a layer), and applying atomic sparsification to their SVD is meaningful and well-motivated by the fact that these matrices are approximately low-rank (see Fig. 3.9). We show that ATOMO on the SVD of each layer's gradient, can lead to less variance, and faster training, for the same communication budget as that of QSGD or TernGrad. We present extensive experiments showing that using ATOMO with SVD sparsification can lead to up to $2\times/3\times$ faster training time (including the time to compute the SVD) compared to QSGD/TernGrad.

3.3.2 Prior work

ATOMO is closely related to work on communication-efficient distributed mean estimation [88, 157]. These works both note, as we do, that variance (or equivalently the mean squared error) controls important quantities such as convergence, and they seek to find a low-communication vector averaging

scheme that minimizes it. Our work differs in two key aspects. First, we derive a closed-form solution to the variance minimization problem for all input gradients. Second, ATOMO applies to any atomic decomposition, which allows us to compare entry-wise against singular value sparsification for matrices. Using this, we derive explicit conditions for which SVD sparsification leads to lower variance for the same sparsity budget.

The idea of viewing gradient sparsification through a meta-optimization lens was also used by Wangni et al. [168]. Our work differs in two key ways. First, Wangni et al. [168] consider the problem of minimizing the sparsity of a gradient for a fixed variance, while we consider the reverse problem, that is, minimizing the variance subject to a sparsity budget. The second more important difference is that while Wangni et al. [168] focus on entry-wise sparsification, we consider a general problem where we sparsify according to any atomic decomposition. One use case of our approach allows directly sparsifying the singular values of the gradient matrix, which gives rise to faster training algorithms in experimentally.

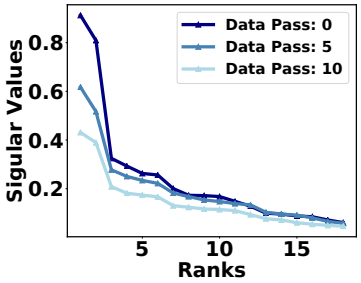


Figure 3.9: The singular values of a convolutional layer's gradient, for ResNet-18 while training on CIFAR-10. The gradient of a layer can be seen as a matrix, once we vectorize and appropriately stack the conv-filters. For all presented data passes, there is a sharp decay in singular values, with the top 3 standing out.

Finally, low-rank factorizations and sketches of the gradients when viewed as matrices have been proposed in many works (e.g, [68, 88, 139, 172, 175]). Arguably most of these methods (with the exception of Konečnỳ et al. [88]) aimed to address the high FLOPs required when training low-rank models. Though they did not directly aim to reduce communication, this arises as a useful side effect.

3.3.3 Preliminaries

Mini-batch SGD is easily parallelized, as mentioned in Fig. 3.4. For this work, let's focus on the centralized architecture,[39] in which the parameter server (PS) stores the global model, and P compute nodes split the effort of computing the B gradients. Once the PS receives these gradients, it applies them to the model, and sends it back to the compute nodes.

To prove convergence bounds for stochastic-gradient based methods, we usually require $\widehat{\boldsymbol{g}}$ to be an unbiased estimator of the full-batch gradient, and to have small variance $\mathbb{E}\left[\|\widehat{\boldsymbol{g}}\|^2\right]$, as this controls the speed of convergence. To see this, suppose $\boldsymbol{w}_k$ is the model after model update k and that $\boldsymbol{w}^*$ is a critical point of F, then we have

$$\mathbb{E}\left[\|\boldsymbol{w}_{k+1}-\boldsymbol{w}^*\|_2^2\right] = \mathbb{E}\left[\|\boldsymbol{w}_k-\boldsymbol{w}^*\|_2^2\right] - \underbrace{\left(2\gamma\langle\nabla F(\boldsymbol{w}_k),\boldsymbol{w}_k-\boldsymbol{w}_*\rangle - \gamma^2\mathbb{E}[\|\widehat{\boldsymbol{g}}\|_2^2]\right)}_{\text{progress at step } t}.$$

In particular, the *progress* made by the algorithm at a single step is, in expectation, controlled by the term $\mathbb{E}\left[\|\widehat{\boldsymbol{g}}\|\right]_2^2$; the smaller it is, the bigger the progress. This is a well-known fact in optimization, and most convergence bounds for stochastic-gradient based methods, including minibatch, involve upper bounds on $\mathbb{E}\left[\|\widehat{\boldsymbol{g}}\|_2^2\right]$, in a multiplicative form, for both convex and nonconvex setups [24, 38, 42, 54, 80, 118, 134, 138, 176]. Hence, recent results on low-communication variants of SGD design unbiased quantized or sparse gradients, and try to minimize their variance [7, 89, 168].

Since variance is a proxy for speed of convergence, in the context of communication-efficient stochastic gradient methods, one can ask: *what is the smallest possible variance of a stochastic gradient that is represented*

[39] This work can also use the decentralized case each worker receives *every* worker's gradient, not the *sum of* every worker's gradient as other in distributed SGD trainings [128, 146].

with k bits? This can be cast as the following meta-optimization:

$$\begin{aligned} &\min \mathbb{E}\left[\|\widehat{\boldsymbol{g}}_k\|_2^2\right] \\ &\text{s.t. } \mathbb{E}\left[\widehat{\boldsymbol{g}}_k\right] = \boldsymbol{g}_k, \quad \widehat{\boldsymbol{g}}_k \text{ can be expressed in } k \text{ bits} \end{aligned} \tag{3.6}$$

Here, the expectation is taken over the randomness of $\widehat{\boldsymbol{g}}$. We are interested in designing a stochastic approximation $\widehat{\boldsymbol{g}}$ that "solves" this optimization. However, it seems difficult to design a formal, tractable version of the last constraint. In the next section, we replace this with a simpler constraint that instead requires that $\widehat{\boldsymbol{g}}$ is sparse with respect to a given atomic decomposition.

3.3.4 Main results

Let's create the gradient estimate $\widehat{\boldsymbol{g}}$ to be well-suited common bases (e.g. the standard, Fourier, or wavelet bases). Let's represent a basis vector with $\boldsymbol{a}_i$ where $\|\boldsymbol{a}_i\|_2^2 = 1$ for all i, and let the gradient estimate be given by

$$\widehat{\boldsymbol{g}} = \sum_{i=1}^{n} \frac{t_i}{p_i} \lambda_i \boldsymbol{a}_i \tag{3.7}$$

where λ_i is a constant to ensure $\mathbb{E}\left[\widehat{\boldsymbol{g}}\right] = \boldsymbol{g}$ when there are n basis vectors, and the t_i's are independent random variables with $t_i \in \text{Bernoulli}(p_i)$ for $p_i \in (0, 1]$. We refer to this scheme as *atomic sparsification*. With an expected communication budget of s basis vectors and the formulation above, Eq. (3.6) becomes

$$\min_{\boldsymbol{p}} f(\boldsymbol{p}) := \sum_{i=1}^{n} \frac{\lambda_i^2}{p_i} \quad \text{s.t. } \forall i \; p_i \in (0, 1], \sum_{i=1}^{n} p_i = s \tag{3.8}$$

with an expected communication budget of s. Theorem 23 solves this optimization problem, which is relegated to Appendix D.1 because of edge cases. Instead, here's an informal solution:

Corollary 9 (informal presentation of Theorem 23)**.** *Given a communication*

budget s, Eq. (3.8) is solved when the probabilities in Eq. (3.7) are

$$p_i = \frac{s\,|\lambda_i|}{\|\boldsymbol{\lambda}\|_1}$$

if all $p_i \leq 1$.

This avoids edge cases where the gradient vector isn't "balanced," specifically when $p_i > 1$, and first discovered by Konečnỳ et al. [89]. Briefly, if there exists an index where $p_i > 1$, the optimal strategy of assigning probabilities is to set $p_j = 1$ and recurse the solution with the other $n - 1$ entries where $j = \arg\max_{i\in[n]} |\lambda_i|$. A formal solution is in Appendix D.1.

3.3.4.1 Relation to QSGD and TernGrad

In this section, we will discuss how ATOMO is related to two recent quantization schemes (1-bit QSGD [7] and TernGrad [171]) that have been extensively tested/used [60, 88, 89, 157] We will show that in certain cases, these schemes are versions of the ATOMO for a specific sparsity budget s. Both schemes use the entry-wise atomic decomposition.

QSGD takes as input $\boldsymbol{g} \in \mathbb{R}^n$ and $b \geq 1$. This b governs the number of quantization buckets. When $b = 1$, QSGD produces a random vector $Q(\boldsymbol{g})$ defined by

$$Q(\boldsymbol{g})_i = \|\boldsymbol{g}\|_2 \text{sign } (g_i)\zeta_i.$$

Here, the $\zeta_i \sim \text{Bernoulli}(|g_i|/\|\boldsymbol{g}\|_2)$ are independent random variables. One can show this is equivalent to Eq. (3.7) with $p_i = |g_i|/\|\boldsymbol{g}\|_2$ and sparsity budget $s = \|\boldsymbol{g}\|_1/\|\boldsymbol{g}\|_2$. Note that by definition, any $\boldsymbol{g}$ is s-balanced for this s. Therefore, Theorem 23 implies that the optimal way to assign p_i with this given s is $p_i = |g_i|/\|\boldsymbol{g}\|_2$, which agrees with 1-bit QSGD.

TernGrad takes $\boldsymbol{g} \in \mathbb{R}^n$ and produces a sparsified version $T(\boldsymbol{g})$ given by

$$T(\boldsymbol{g})_i = \|\boldsymbol{g}\|_\infty \text{sign } (g_i)\zeta_i$$

where $\zeta_i \sim \text{Bernoulli}(|g_i|/\|\boldsymbol{g}\|_\infty)$. This is equivalent to Eq. (3.7) with $p_i = |g_i|/\|\boldsymbol{g}\|_\infty$ and sparsity budget $s = \|\boldsymbol{g}\|_1/\|\boldsymbol{g}\|_\infty$. Once again, any g is s-balanced for this s by definition. Therefore, Theorem 23 implies that the optimal assignment of the p_i for this s is $p_i = |g_i|/\|\boldsymbol{g}\|_\infty$, which agrees with TernGrad.

We can generalize both of these with the following quantization method. Fix $q \in (0, \infty]$. Given $\boldsymbol{g} \in \mathbb{R}^n$, we define the ℓ_q-quantization of $\boldsymbol{g}$, denoted $L_q(\boldsymbol{g})$, by

$$L_q(\boldsymbol{g})_i = \|\boldsymbol{g}\|_q \text{sign } (g_i)\zeta_i$$

where $\zeta_i \sim \text{Bernoulli}(|g_i|/\|\boldsymbol{g}\|_q)$. By the reasoning above, we derive the following theorem.

Theorem 10. *ℓ_q-quantization performs atomic sparsification in the standard basis with $p_i = |g_i|/\|\boldsymbol{g}\|_q$. This solves Eq. (3.8) for $s = \|\boldsymbol{g}\|_1/\|\boldsymbol{g}\|_q$ and satisfies $\mathbb{E}[\|L_q(\boldsymbol{g})\|_2^2] = \|\boldsymbol{g}\|_1\|\boldsymbol{g}\|_q$.*

In particular, 1-bit QSGD is obtained with $q = 2$ and Terngrad is obtained with $q = \infty$.

3.3.5 Experimental results

We have developed an encoding technique that uses the singular vectors as the orthogonal basis vectors, a.k.a. spectral-ATOMO. We have run an extensive empirical study to evaluate performance under real distributed environments. The following are our main findings:

- We observe that spectral-ATOMO provides a useful alternative to entry-wise sparsification methods. It reduces communication compared to vanilla mini-batch SGD and can reduce training time compared to QSGD and TernGrad by up to a factor of 2× and 3× respectively.

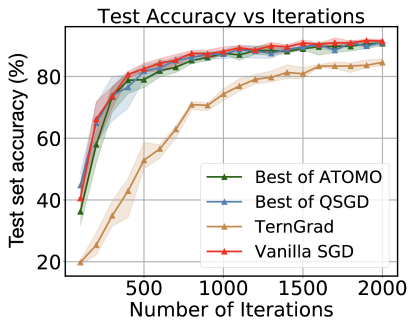
- We observe that spectral-ATOMO in distributed settings leads to models with negligible accuracy loss when combined with parameter tuning.

In this section, a comparison between spectral-ATOMO, QSGD [7] and TernGrad [171] will be presented. Both ATOMO and QSGD have parameters to control the amount of communication (the number of singular values s and the number of bits b).

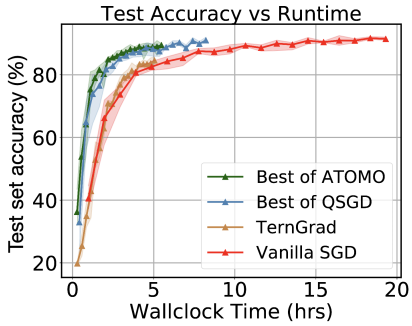
We evaluate the end-to-end convergence performance on different datasets and neural networks, training with spectral-ATOMO(with sparsity budget $s = 1, 2, 3, 4$), QSGD (with $n = 1, 2, 4, 8$ bits), and ordinary mini-batch SGD. The model used is ResNet-18 [63] for the SVHN dataset [116], and the training uses a distributed cluster with 16 compute nodes. SGD with a batch size of 512 will be used, and SGD's step size will be tuned (as detailed in Appendix D.4). For simplicity and to reduce the formidable tuning, momentum and weight decay will be disregarded.

Our distributed cluster with a parameter server (PS), implemented in mpi4py [40] and PyTorch [121] and deployed on Amazon AWS EC2 `g2.2xlarge` machines, which have NVIDIA GRID GPUs. The PS implementation has 16 workers and is standard with point-to-point communications (as in Abadi et al. [2, Fig. 5b]). At the most basic level, the PS implementation receives gradients from each compute nodes and broadcasts the updated model once a batch has been received.

The gradients of convolutional layers are 4 dimensional tensors with shape of $[x, y, k, k]$ where x, y are two spatial dimensions and k is the size of the convolutional kernel. However, matrices are required to compute the SVD for spectral-ATOMO, and we choose to reshape each layer into a matrix of size $[xy/2,\ 2k^2]$. This provides more flexibility on the sparsity budget for the SVD sparsification. For QSGD, we use the bucketing and Elias recursive coding methods proposed by Alistarh et al. [7], with bucket size equal to the number of parameters in each layer of the neural network.



(a) Convergence against model updates/iterations.



(b) Convergence against wall-clock time.

Figure 3.10: Convergence rates for ResNet-18 on the SVHN dataset with four different optimizers: vanilla SGD, ATOMO, QSGD, and TernGrad.

We observe that QSGD and ATOMO speed up model training significantly and achieve similar accuracy to vanilla mini-batch SGD in Fig. 3.10. Tuning the communication budget is required in QSGD and ATOMO, so the best performance over four different communication budgets is shown. During this, we've observed that the best performance is not achieve with the most sparsified/quantized method, but the optimal method lies somewhere in the middle where enough information is preserved during the sparsification. For instance, 8-bit QSGD converges faster than 4-bit QSGD, and spectral-ATOMO with sparsity budget 3, or 4 seems to be the fastest. Higher sparsity can lead to a faster running time, but extreme sparsification can adversely affect convergence. For example, for a fixed number of iterations, 1-bit QSGD has the smallest time cost, but may converge much more slowly to an accurate model.

3.3.6 Conclusion

In this chapter, we have presented and analyzed ATOMO, a general sparsification method for distributed stochastic gradient based methods. ATOMO applies to any atomic decomposition, including the entry-wise and the SVD

of a matrix. ATOMO generalizes 1-bit QSGD and TernGrad, and provably minimizes the variance of the sparsified gradient subject to a sparsity constraint on the atomic decomposition. We focus on the use ATOMO for sparsifying matrices, especially the gradients in neural network training.

Future work involves determining which atoms are appropriate for different neural networks/layers.[40] That's especially relevant with the Fourier decomposition given it's use in image/audio compression.[41]

[40] Grubic et al. report that convolutional layers require more bits than fully-connected layers [60].

[41] Since publication, there has been some relevant work on communicating the Fourier transforms of the gradients between worker nodes [167]. Though they have a high performance cluster and implementation, they don't use the directly applicable atomic sparsification described above, and instead rely on an extra assumption to guarantee convergence [167, Assumption 3.2].

3.4 Conclusion

In summary, the following has been shown:

1. Few model updates are required when the batch size grows in a particular manner. The particular method of batch size growth requires the same amount of gradient computation as standard SGD.[42]
2. Training time is (approximately) proportional to the number of model updates provided that the number of workers in a distributed system is proportional to the batch size. This is particularly useful when combined with item (1).
3. A particular type of lossy gradient compression can reduce training time in the case when many model updates are required. The proposed lossy gradient compression framework generalizes several other gradient compression schemes [7, 167, 171].

With this, the "model training" block in Fig. 1.3 will require less time. Of course, the methods above are not the only methods to reduce the time required for this block; methods specifically designed for large batch training perform rather well [58, 75, 78, 178], potentially at the cost of debugging with many GPUs.

[42] At least for convex and strongly convex functions.

4 EFFICIENT DEPLOYMENT OF ACTIVE MACHINE LEARNING ALGORITHMS FOR CROWDSOURCING

So far, this dissertation has focused on accelerating each individual block in Fig. 1.3, specifically on adapting query priority to a distributed system in Chapter 2 and accelerating model training in Chapter 3. In this chapter, let's focus on efficiently searching for queries and collecting responses while deploying active ML algorithms to crowdsourcing audiences.

The problem of interest is the "ordinal embedding" problem, which generates embeddings that cluster similar objects together. It's common for social scientists to generate these embeddings to estimate human perceived similarity between objects. Use cases include generating embeddings for facial emotions [108], molecules [109], shoes [64] and materials [92].

These embeddings are generated from relatively similarity judgments via "triplet queries," questions of the form "is item a or b more similar to item h?" In practice a "prohibitively large number of responses" [64] are required, meaning query collection is expensive and can limit the number of objects in the embedding. As such, many active or adaptive machine learning algorithms have been developed to reduce the number of queries by selecting the most informative queries [4, 28, 65, 158, 163].

However, active ML algorithms for the ordinal embedding problem face several challenges:

- **There are many possible queries.** The number of triplet queries grows like $\mathcal{O}(n^3)$ for an embedding of n objects.
- **Scoring each query with information gain is computationally intensive** [64, 158]. Maximizing information gain is a popular method to select the next query to show a user [64, 71, 158], but scoring every triplet is "prohibitively expensive" [64].
- **The model update requires a minimization over all data** re-

ceived thus far, aka the "follow the leader" strategy [148]. Online model updates require careful thought because the loss functions for the ordinal embedding problem are (typically) non-convex [101][1] (definitely with neural networks [165]).

We have developed a software system that addresses these constraints. When deployed to crowdsourcing audiences, our system and the implemented active ML algorithms can require fewer responses than passive ML algorithms. To show that, let's mention some related work in Section 4.1 before summarizing the features of our system in Section 4.2. Then, we'll show experimental results in Section 4.3 and mention future work in Section 4.4. During this, we represent an embedding of n objects in d dimensions with the matrix $\boldsymbol{X} \in \mathbb{R}^{n \times d}$ (typically, $d = 2$). The coordinates of embedding object i will be represented by $\boldsymbol{x}_i \in \mathbb{R}^d$. For the query "is item a or b more similar to item h?", we will refer to item h as the "head" and items a and b as the "feet."

4.1 Related Work

The most relevant work is with NEXT, a system to deploy general active ML algorithms to crowdsourcing audiences [71]. For the ordinal embedding problem, Jamieson et al. used a variety of active ML algorithms but found "no evidence [of] gains from adaptive sampling" [71, Sec. 3.2] despite finding gains for other problems [71, Sec. 3.1]. NEXT mirrors prior work by performing information gain maximization over a subset of possible triplets [64, 158].

[1] Even the strongly convex dual formulation requires projection onto the positive semi-definite cone, which poses challenges for online methods [65].

4.1.1 Embedding schemes

A good overview of methods for generating embeddings is presented by Vankadara et al. [165]. Briefly, it's possible to find an embedding that agrees with human responses by minimizing the negative log-likelihood when the probability of response i being correct is characterized by p_i [158, 163]. It's also possible to find embeddings with the generalized non-metric multidimensional scaling (GNMDS), essentially hinge loss on the squared distances [4].

ERKLE is a method of generating ordinal embeddings that is designed for online computation [65]. ERKLE uses the Gram matrix $\boldsymbol{G} \in \mathbb{R}^{n \times n}$ instead of the embedding matrix $\boldsymbol{X} \in \mathbb{R}^{n \times d}$, which provides a convex embedding formulation at the cost of projecting onto the positive semi-definite cone (an $\mathcal{O}(n^3)$ operation) to ensure the Gram matrix has positive eigenvalues. Heim et al. present a method to minimize the number of times this operation is required and the number of eigenvalues/vectors that need to be computed [65].

Regardless of the embedding scheme, $\mathcal{O}(nd \log n)$ responses will be required with high probability for random sampling to generate an approximate embedding for n objects into d dimensions [69, Sec. 4]. Even with an adaptive scheme, at least $\Omega(nd \log n)$ responses are required [72].

4.1.2 Choosing queries

In the classic serial pool based active learning setup, every query is scored and the top scoring query receives a single response [147, Sec. 2.3]. Then, a new model is produced that minimizes the loss function over all examples received thus far in the "follow-the-leader" approach [148].

Two popular methods to score queries involve maximizing information gain [28, 29, 64, 71, 158] or uncertainty [147, Sec. 3.1][100].[2] Evaluating

[2] "Most uncertain" can mean the query that's closest to the decision boundary

the information gain or uncertainty of a single query is an $\mathcal{O}(nd)$ or $\mathcal{O}(d)$ operation respectively.[3] Performing information gain maximization can require significant computational effort [64, 158] but avoids low information queries.[4] For example, evaluating the score of every query with $n = 50$ objects will take about a minute if evaluating the information gain of a single query takes 1 millisecond. This quickly becomes infeasible because the number of queries grows like $\mathcal{O}(n^3)$ for an embedding with n objects. Correspondingly, there are (easier to compute) methods to approximate information gain [29].

4.2 Crowdsourcing active machine learning algorithms

We have developed Salmon, a software system to serve triplet queries to crowdsourcing audiences. Salmon has a clear separation between the primary components of deploying active ML algorithms to crowdsourcing audiences – *searching* for queries and *serving* those queries (and likewise, *receiving* responses and *processing* those responses). Salmon separates these two components with a frontend and a backend, which communicate with a database as shown in Figure 4.1.

Salmon's backend searches for queries (then posts them to the database) and generates embeddings from responses. All the tasks on the backend are run concurrently and synchronously, as illustrated in Fig. 4.2. The task scheduler is Dask Distributed [41], the same manager in Chapter 2

for hinge-loss, or closest to 50% probability with probabilistic models.

[3] For any scoring scheme, the computational complexity is reduced by a factor of d if the Gram matrix $\boldsymbol{G} \in \mathbb{R}^{n \times n}$ is used instead of the embedding matrix $\boldsymbol{X} \in \mathbb{R}^{n \times d}$.

[4] A low information query is one that's truly uncertain, common if the embedding has similar objects. A similar phenomena have observed in natural language processing [160, 188].

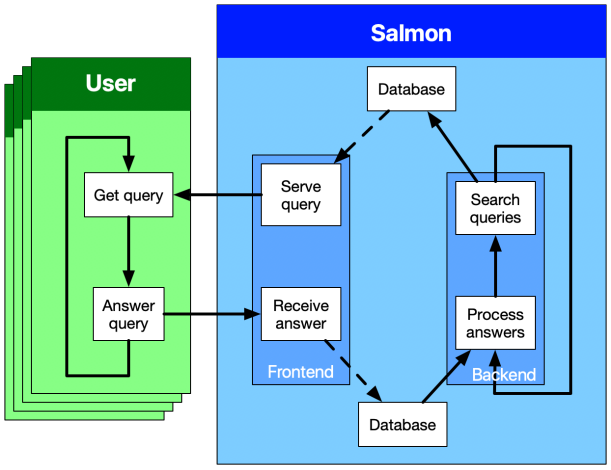


Figure 4.1: Four users receiving queries from SALMON, and SALMON's architecture for serving queries and processing responses. Both the frontend and backend can be run independently; a random query is served if no queries are present in the database.

and Section 3.2. The backend is only relevant for adaptive algorithms because it's used to generate embeddings, which passive algorithms do not require to generate queries (almost by definition).

4.2.1 Generating embeddings

SALMON can generate embeddings from several loss functions, including crowd kernel (CK) [158], [t-Distributed] stochastic triplet embedding ([t]STE) [163], soft ordinal embeeding (SOE) [165], and GNMDS [4]. By default, the t-STE noise model is used because it has a heavy-tailed probability distribution (which is more robust to embedding errors [163, Sec. 4]).

The classic "follow-the-leader" minimization [148] is performed when the "process answers" block in Figure 4.1 is executed for the kth time:

$$\boldsymbol{X}_{k+1} = \arg\min_{\boldsymbol{X}} \sum_{i=0}^{T_k} \ell(\boldsymbol{X}; \boldsymbol{q}_i, y_i)$$

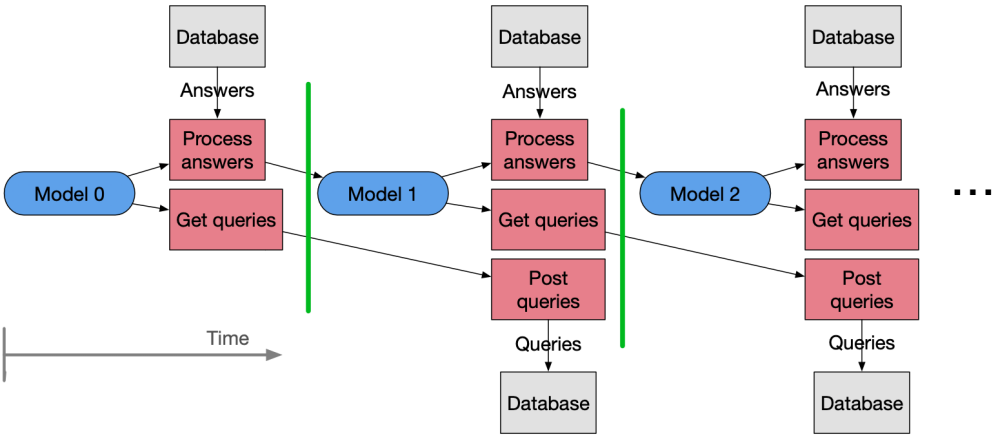


Figure 4.2: An illustration of the backend timing for SALMON. All red blocks must be finished before the green line is reached. "Process answer" controls the timing. When it's finished, it sends a signal to stop searching/posting queries, the reason the "process answer" box is the shortest. Model 1 is not produced until sufficient answers have been received (and likewise for the independent process of searching/posting queries).

when T_k responses have been received when iteration k begins, ℓ is the loss function for one example, $\boldsymbol{q}_i \in \{1, \ldots, n\}^3$ is the ith query shown and $y_i \in \{\pm 1\}$ is the human's response indicating which of the query's "feet" is closer to the query "head." To perform this minimization, we use a modern machine learning library (PyTorch[5] [121]) with adaptive learning rates via the Adadelta optimizer [181] by default. This minimization is performed for between 50 and 200 epochs over the training data received thus far,[6] provided a sufficient number of responses have been received. This computation is repeated until no answers have been received for 20 consecutive iterations of Fig. 4.2 (approximately 2 minutes), and restarts shortly after one response is received.

[5]In the SALMON context, PyTorch offers significant benefits around saving models to the database because PyTorch models can be serialized.

[6]Enough for convergence for random sampling in brief testing.

4.2.2 Searching for queries

By default, SALMON performs information gain maximization, a common method for ordinal embeddings [64, 71, 158]. By default, SALMON relies on the heavy-tailed t-STE probability proposed by Maaten et al. [163]. We have significantly accelerated the query search computation in two ways:

- **By using the distance matrix.** The information gain of a query only depends on the Euclidean distances between embedding points. Correspondingly, most of the time is spent in the computation of (squared) Euclidean distance $\|\boldsymbol{x}_i - \boldsymbol{x}_j\|_2^2$ for two vectors $\boldsymbol{x}_i, \boldsymbol{x}_j \in \mathbb{R}^d$, which can be circumvented with quick computation of the distance matrix.[7]
- **By vectorization.** An array of queries is scored instead of scoring a single query. This allows utilization of efficient C and Fortran code instead of (the much slower) Python code [164].

These accelerations mean that SALMON can search far more queries than prior work [71], even without the architecture differences. This is illustrated in Fig. 4.3, which shows a slightly improved version[8] of NEXT's publicly available search.[9]

4.2.3 Usage

Launching SALMON only requires a web browser and Amazon AWS account, as detailed in the directions at https://docs.stsievert.com/salmon/installation.html#experimentalist. Briefly, these directions

[7] The distance matrix can be computed from the Gram matrix $\boldsymbol{G} = \boldsymbol{X}\boldsymbol{X}^T$ [65, Sec. 3].

[8] NEXT's search is accelerated by slightly less than factor of 3 by precomputing a probability vector (instead of needlessly re-computing it twice).

[9] https://github.com/nextml/NEXT/blob/v1.2.5/apps/PoolBasedTripletMDS/algs/STE/myAlg.py#L64-L90

Metric	NEXT	SALMON	Ratio
Queries searched in 50ms	≤ 60	$\approx 10,000$	≈ 167
Effective search time per response	50ms	$\geq$ 200ms	≥ 4
Queries searched per response	≤ 60	$\geq 40,000$	≥ 667

Figure 4.3: A comparison between (a slightly accelerated version of) NEXT and SALMON's search for $n = 85$ and $d = 2$. SALMON is capable of searching far more queries per user, even with a moderately heavy load of 5 users with a minimum response time of 1s.

detail launching an Amazon EC2 instance with a particular Amazon Machine Image (AMI).[10] This documentation recommends launching with a `t3.xlarge` machine,[11] which is used in all of our experiments in Section 4.3.

After SALMON is launched, the experimentalist needs to upload a file(s) to initialize their experiment[12] and send a particular URL to the crowdsourcing audience. Then, they can monitor the results and SALMON performance, and download the responses and embedding.

4.3 Experimental results

Let's run two sets of experiments[13] that will embed $n = 90$ "alien eggs" shown in Fig. 4.4 into $d = 2$ dimensions as in prior work [71]. The first set of experiments will deploy SALMON to a crowdsourcing audience, and the second set of experiments will use a synthetic noise model for a more detailed comparison.

[10] `ami-0e3134e3437ec5b85`, available in the `us-west-2` region.

[11] Which costs about $4/day.

[12] Documentation is available at https://docs.stsievert.com/salmon/getting-started.html.

[13] Complete details are at https://github.com/stsievert/salmon-experiments.

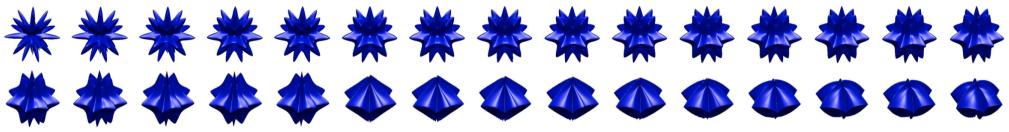


Figure 4.4: A subset of the $n = 90$ "alien eggs" used in this section. Humans provide responses to the question "is alien egg a or b most similar to alien egg h?"

4.3.1 Crowdsourcing

In this experiment, let's recruit crowdsourcing participants from Amazon's Mechanical Turk. Let's focus on three different search strategies:

- `Random`, which presents the user with a randomly chosen query without regard to the current embedding.
- `Greedy`, which presents the query that maximizes information gain given the current embedding [71].
- `ARR` aka "adaptive round robin," which is nearly the same as `Greedy` but randomly chooses the query's "head" then finds the two "feet" that maximize information gain (a slight modification of prior work [64, 158]).

In addition, we will collect some responses to random queries for testing. Both `Greedy` and `ARR` will perform a near exhaustive search.[14] Round-robin schemes are not uncommon in the literature [28, 64, 158], likely due to the fact that the exhaustive searches are "computationally intensive" [158] or "prohibitively expensive" [64].

Both of these strategies will be implemented in Salmon to provide a fair comparison between query selection strategies. Each crowdsourcing participant will answer 50 queries from the same strategy, and each of

[14]Better performance may be obtained by tuning the search length, as shown in Appendix E.2.

these schemes will be run independently 10 times. In total, 194,286 human responses were collected.[15]

We have generated embeddings from these responses offline after downloading the responses from SALMON. The performance of these embeddings is shown in Fig. 4.5, which is measured with two metrics: accuracy on a test dataset and a measure of embedding "tightness," the average distance to the true nearest neighbor (NN) for each target item.[16]

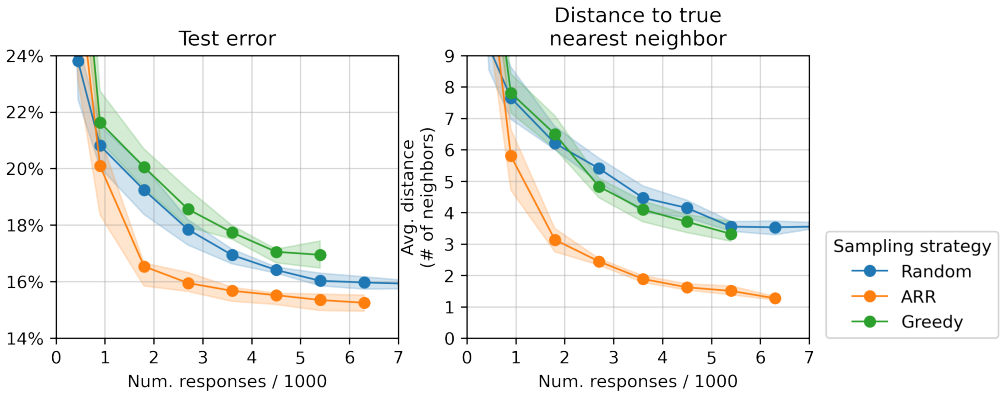


Figure 4.5: The performance over 10 different independent runs of each scheme. The solid line is the mean, and the shaded region is the inter-quartile range.

In these experiments, `Greedy` and `Random` perform about equivalently, and `ARR` requires fewer responses than either `Random` or `Greedy` to obtain an embedding of a particular quality. For example, to reach a mean NN distance of 4, the median `ARR` run requires at least 1,800 responses, and the both the median `Greedy` and `Random` run require at least 4,500 responses.

[15]The responses collected for `Random`, `Greedy`, `ARR`, and testing are 52,806, 54,284, 67,583, and 19,613 respectively. All 10 `ARR` runs have at least 5,400 responses, and 8 runs have 6,300 responses. All 10 `Greedy` runs have at least 4,700 responses, and 4 runs have 5,400 responses.

[16]The "smoothness" parameter for each alien egg is known. We define the "true nearest neighbor of an alien egg" to be "the alien egg that has the smallest absolute difference in smoothness parameter."

`Greedy` does not perform well, on par with `Random`. One possible reason for the reduced performance is that the query head tends to remain constant for many trials as illustrated in Fig. 4.6. We have found some evidence that this "constant head" issue persists throughout a model update, as illustrated in Appendix E.1.

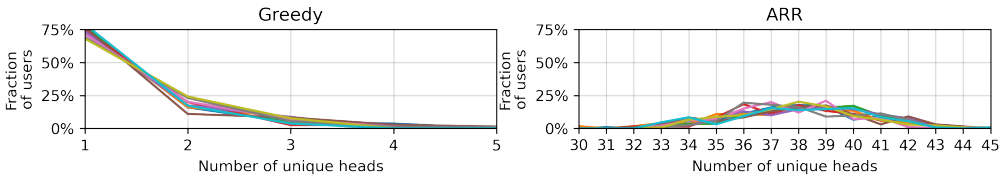


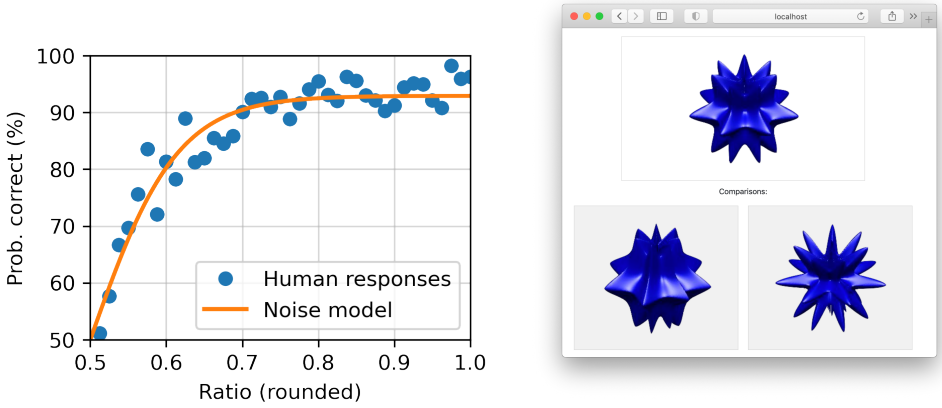
Figure 4.6: The number unique heads for the crowdsourcing participants that answer at least 50 queries and have at least 95% of their queries chosen from the adaptive search. The left and right plots show `Greedy` and `ARR` respectively. In each plot, each of the 10 lines represents one crowdsourcing server.

4.3.2 Synthetic comparisons

Let's look at the searches in more detail. To aid that, let's isolate software performance by using a synthetic noise model shown in Fig. 4.7 and submitting responses at a constant response rate (an average of 1 response/sec).[17] This section will also compare against prior work. Let's run simulations with five different sampling strategies:

1. `Random`, which asks about a random query regardless of the current embedding.

[17] In our crowdsourcing experiments, we found that more difficult questions had a slightly longer response time (an average of 2.5s and 4.5s for the easiest and hardest queries respectively).



(a) The synthetic noise model and human responses.

(b) An example query with ratio 0.59 (and some text removed).

Figure 4.7: The probability that the synthetic noise model answers correctly is characterized by the orange line in (a), a sigmoid derived from the human responses supplied by Jamieson et al. [71, Sec. 3.2]. The ratio is $\max(|h-a|, |h-b|)/(|h-a|+|h-b|)$ when h, a and b are the smoothness parameters for the alien eggs in the head, left and right positions respectively.

2. `NEXT`, which evaluates the information gain of k random queries for any one user and shows the query with maximum information gain. Let's use the public version of NEXT,[18] which has $k \approx 20$.
3. `Greedy-100`, which implements a search strategy in Salmon that mirrors the search that `NEXT` performs. In addition to the difference in backend embedding strategy, k is also approximately between 30 and 100,[19] slightly more than `NEXT`.
4. `Salmon`, which is same `ARR` scheme in the crowdsourcing experiments above. Again, it randomly chooses the query "head" then finds the two "feet" that maximize information gain (a slight modification of

[18] NEXT v1.2.5 via their AMI on a `c4.4xlarge` instance which costs $19.10/day. NEXT's documentation recommends launching on a `c3.4xlarge` instance (which is in the previous generation).

[19] The implementation of this speedup in NEXT is entirely feasible. The range for k comes from the fact that Salmon has a frontend and a backend.

prior work [64, 158]).

5. `ARR-100`, the same search as `Salmon`, but one that is expected to search between 37 and 111 queries per head,[20] making it very approximately similar to `Greedy-100` with random head selection.

These search strategies are summarized in Fig. 4.8, which details two confounding variables: search length and head selection. Note that `Greedy-100`, `ARR-100` and `NEXT` all score roughly the same number of queries, and that `Greedy-100` and `ARR-100` really only differ in head selection. Every strategy except `NEXT` is implemented in Salmon to provide a fair comparison of improvements. `Greedy-100` and `NEXT` are designed to be similar implementations with different backend systems to verify that they generate embeddings of similar quality.

Sampler	Search length k	Head selection	Server
`Random`	$k = 1$	n/a	Salmon
`NEXT`	short ($k \approx 20$)	Greedy	NEXT
`Greedy-100`	short ($k \approx 100$)	Greedy	Salmon
`ARR-100`	short ($k \approx 111$)	Random	Salmon
`Salmon`	long ($k \approx 200{,}000$)	Random	Salmon

Figure 4.8: A comparison of the searches schemes used with the synthetic noise model.

The performance of these different sampling methods is shown in Figure 4.9. For this simulation, `Salmon` requires fewer responses than `Random` to obtain an embedding of a particular quality. In addition, it shows that `NEXT` and `Random` produce embeddings with similar performance (as in prior work [71]). `ARR-100` significantly improves performance and is entirely feasible to implement in NEXT because it searches approximately the same

[20] It's expected that `ARR-100` will search up to 111 queries per head because it actually searches up to 10,000 queries then randomly chooses a head (necessary because of Salmon's decentralized architecture).

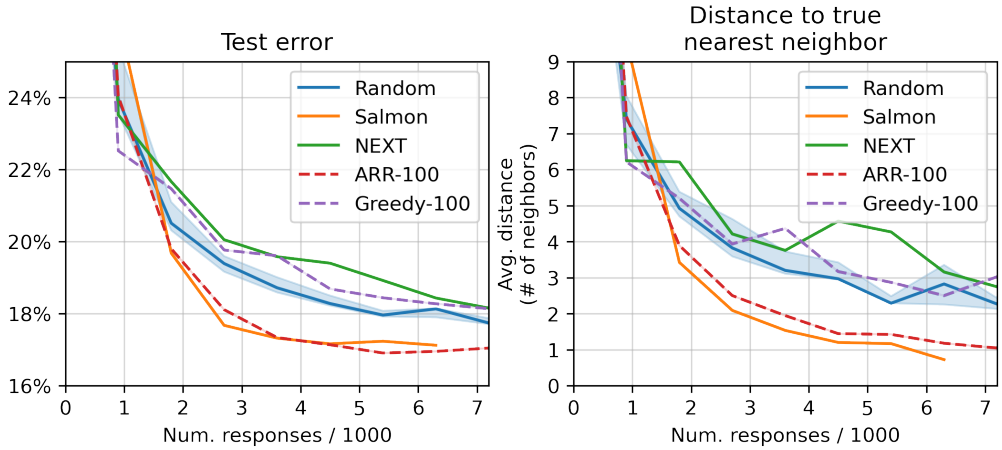


Figure 4.9: A comparison of the five search strategies mentioned in Sec. 4.3.2. The test set consists of 60,000 synthetic responses, and the "average distance to the true nearest neighbor (NN)" graph on the right is a measure of embedding quality (possible because the ground truth embedding is known). Every search except `Random` had one run, which is why no shaded region or error bars are shown. The shaded region for `Random` represents the boundary between the 20th and 80th percentiles over 5 runs with different random seeds.

number of queries.[21] Notably, `Salmon` and `ARR-100` have roughly the same performance, despite that `Salmon` performs a much more complete query search. By that measure, perhaps a key competency of Salmon, enabling long-running query searches is not relevant.[22]

4.4 Conclusion

We have designed a software system that can effectively run active ML algorithms for crowdsourcing in the presence of slow query searches and/or

[21] Fig. 4.3 shows an improved version of NEXT that had been accelerated by a factor of roughly 3.

[22] That would be relevant if the priority scheme changes shown in Appendix E.3 exhibited better performance.

model updates. In addition, we have illustrated good active ML performance with a class of active ML algorithms that have relatively slow query searches in prior work [64, 158]. Our system enables near exhaustive query searches. However, exhaustive searches present some experimental design challenges (constant query heads), and we illustrate that random head selection works around this issue. Luckily, our system is ready to incorporate more complex model updates [165]. Experiments illustrate that this strategy exhibits good performance with human responses. In addition, we have presented some evidence that random head selection would work well in systems developed in prior work [71].

The most immediate future work involves incorporating adaptive step size methods to more efficiently generate the embeddings from responses [101, 102], methods specifically designed for online learning [65], or tuning one of the batch size increase schedules mentioned in Section 3.1. Future work might involve resolving the issues with information gain maximization in exhaustive query searches, some of which are mentioned in Appendices E.1 and E.3. In addition, future work involves generalizing Salmon to other problems of interest (such as "dueling bandits" [13]).

5 CONCLUSION

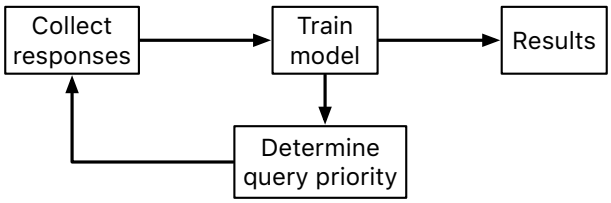


Figure 5.1: The data flow of active ML. For passive ML, each query is equally important and there is no feedback between collection of responses and model training.

Every component of Fig. 1.3 (repeated in Fig. 5.1) has been accelerated in this dissertation. Though there is much context for each problem, the core contributions in relation with Fig. 1.3 are listed below:

1. Chapter 2 determines a "query priority" scheme when using a distributed system for a particular active ML algorithm for hyperparameter optimization. This priority scheme means that **data scientists can produce models of a particular quality at least 3 times quicker** than a passive method.
2. Chapter 3 focuses on some architecture questions for SGD, an extremely common technique for model training and very common in the "model training" block in Fig. 1.3. These architecture questions focus on inter-worker communication and allocating data to model updates. Addressing these issues means that **data scientists can finish their ML model training 2 to 3 times quicker** with a distributed system.
3. Chapter 4 deploys the entire data flow in Fig. 1.3 to crowdsourcing audiences, and focuses on cases where the query searches and/or model updates are not fast. As a result, **social scientists only need to**

collect less than half of the responses required by random sampling.

In total, every chapter/section involves an active/adaptive algorithm[1] or the use of Dask, a Python tool for distributed computation[2] (with the exception of Section 3.3 which provides a general framework for distributed ML model training). At the end of the day, items (1) and (3) both have vaguely similar active algorithm that specifies multiple queries need answering; however, the priority schemes that exhibit the best performance are rather different (highest scoring vs. random as detailed in Section 2.4.1 and Appendix E.3 respectively).

In total, every component of the core loop in Fig. 1.3 has been accelerated in this dissertation, and the whole loop has been effectively deployed to crowdsourcing audiences. Relevant benefits include accelerations to ML model training, ML model selection/hyperparameter optimization and reducing the number of responses required from crowdsourcing audiences. □

[1] Chapters 2 and 4 and Section 3.1.
[2] Chapters 2 and 4 and Section 3.2

BIBLIOGRAPHY

[1] Martin Abadi, Ashish Agarwal, Paul Barham, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. Technical report, Alphabet, Inc., 2015. URL: http://download.tensorflow.org/paper/whitepaper2015.pdf.

[2] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, Savannah, GA, November 2016. USENIX Association. URL: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi.

[3] Alekh Agarwal, Sarah Bird, Markus Cozowicz, Luong Hoang, John Langford, Stephen Lee, Jiaji Li, Dan Melamed, Gal Oshri, Oswaldo Ribas, et al. Making contextual decisions with low technical debt. *arXiv preprint arXiv:1606.03966*, 2016. URL: https://arxiv.org/pdf/1606.03966.pdf.

[4] Sameer Agarwal, Josh Wills, Lawrence Cayton, Gert Lanckriet, David Kriegman, and Serge Belongie. Generalized non-metric multidimensional scaling. In Marina Meila and Xiaotong Shen, editors, *Proceedings of the Eleventh International Conference on Artificial Intelligence and Statistics*, volume 2 of *Proceedings of Machine Learning Research*, pages 11–18, San Juan, Puerto Rico, 21–24 Mar 2007. PMLR. URL: https://proceedings.mlr.press/v2/agarwal07a.html.

[5] Alham Fikri Aji and Kenneth Heafield. Sparse communication for distributed gradient descent. In *EMNLP*, pages 440–445, 2017. doi:10.18653/v1/D17-1045.

[6] S Alexander. Transient weight misadjustment properties for the finite precision LMS algorithm. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 35(9):1250–1258, 1987. doi:10.1109/TASSP.1987.1165279.

[7] Dan Alistarh, Demjan Grubic, Jerry Li, Ryota Tomioka, and Milan Vojnovic. QSGD: Communication-efficient SGD via gradient quantization and encoding. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017. URL: https://proceedings.neurips.cc/paper/2017/file/6c340f25839e6acdc73414517203f5f0-Paper.pdf.

[8] Gene M Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of 1967 joint computer conference*, pages 483–485. ACM, 1967. URL: https://www-inst.cs.berkeley.edu/~n252/sp07/Papers/Amdahl.pdf, doi:10.1145/1465482.1465560.

[9] Sylvain Arlot, Alain Celisse, et al. A survey of cross-validation procedures for model selection. *Statistics surveys*, 4:40–79, 2010. doi:10.1214/09-SS054.

[10] Peter Auer. Using confidence bounds for exploitation-exploration trade-offs. *Journal of Machine Learning Research*, 3(Nov):397–422, 2002. URL: https://www.jmlr.org/papers/v3/auer02a.html.

[11] Lukas Balles, Javier Romero, and Philipp Hennig. Coupling adaptive batch sizes with learning rates. In *33rd Conference on Uncertainty in*

Artificial Intelligence (UAI 2017), pages 675–684. Curran Associates, Inc., 2017. URL: http://auai.org/uai2017/proceedings/papers/141.pdf.

[12] Mikhail Belkin, Daniel J Hsu, and Partha Mitra. Overfitting or perfect fitting? risk bounds for classification and regression rules that interpolate. In *Advances in Neural Information Processing Systems*, pages 2300–2311, 2018. URL: https://papers.nips.cc/paper/2018/hash/e22312179bf43e61576081a2f250f845-Abstract.html.

[13] Viktor Bengs, Robert Busa-Fekete, Adil El Mesaoudi-Paul, and Eyke Haellermeier. Preference-based online learning with dueling bandits: A survey. *Journal of Machine Learning Research*, 22(7):1–108, 2021. URL: http://jmlr.org/papers/v22/18-546.html.

[14] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13(Feb):281–281, 2012. URL: http://jmlr.csail.mit.edu/papers/v13/bergstra12a.html.

[15] James S. Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. In J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. Pereira, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 24*, pages 2546–2554. Curran Associates, Inc., 2011. URL: http://papers.nips.cc/paper/4443-algorithms-for-hyper-parameter-optimization.pdf.

[16] José Carlos M Bermudez and Neil J Bershad. A nonlinear analytical model for the quantized LMS algorithm-the arbitrary step size case. *IEEE Transactions on Signal Processing*, 44(5):1175–1183, 1996. doi:10.1109/78.502330.

[17] Jeremy Bernstein, Yu-Xiang Wang, Kamyar Azizzadenesheli, and Animashree Anandkumar. signSGD: Compressed optimisation for

non-convex problems. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 560–569. PMLR, 10–15 Jul 2018. URL: https://proceedings.mlr.press/v80/bernstein18a.html.

[18] Léon Bottou. Large-scale machine learning with stochastic gradient descent. In Yves Lechevallier and Gilbert Saporta, editors, *Proceedings of the 19th International Conference on Computational Statistics (COMPSTAT'2010)*, pages 177–187. Springer, Paris, France, August 2010. URL: http://leon.bottou.org/papers/bottou-2010.

[19] Léon Bottou. Stochastic gradient tricks. In Grégoire Montavon, Genevieve B. Orr, and Klaus-Robert Müller, editors, *Neural Networks, Tricks of the Trade, Reloaded*, Lecture Notes in Computer Science (LNCS 7700), pages 430–445. Springer, 2012. URL: http://leon.bottou.org/papers/bottou-tricks-2012.

[20] Léon Bottou, Frank E Curtis, and Jorge Nocedal. Optimization methods for large-scale machine learning. *SIAM Review*, 60:223–223, 2018. doi:10.1137/16M1080173.

[21] Léon Bottou. Online learning and stochastic approximations. *On-line learning in neural networks*, 17(9):142, 1998. URL: https://leon.bottou.org/publications/pdf/online-1998.pdf.

[22] Stephen Boyd, , and Lieven Vandenberghe. *Convex optimization*. Cambridge university press, 2004. ISBN: 978-0-521-83378-3. URL: https://web.stanford.edu/~boyd/cvxbook/bv_cvxbook.pdf.

[23] Jamie Brew, Joseph Holt, Lalit Jain, Robert Mankoff, Liam Marshall, Robert Nowak, Rahul Parhi, and Scott Sievert. A robot and comedian walk into a bar, and... aha! Poster:

https://stsievert.com/research/2019/AHA.pdf, Description: https://neurips2019creativity.github.io/doc/robot_and_comedian.pdf, December 2019. Advances in Neural Information Processing Systems, Workshop for "Machine Learning for Creativity and Design".

[24] Sébastien Bubeck et al. Convex optimization: Algorithms and complexity. *Foundations and Trends® in Machine Learning*, 8(3-4):231–357, 2015. URL: http://sbubeck.com/Bubeck15.pdf, doi:10.1561/2200000050.

[25] Lars Buitinck, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, Peter Prettenhofer, Alexandre Gramfort, Jaques Grobler, Robert Layton, Jake VanderPlas, Arnaud Joly, Brian Holt, and Gaël Varoquaux. API design for machine learning software: experiences from the scikit-learn project. In *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, pages 108–122, 2013. URL: https://hal.inria.fr/hal-00856511.

[26] Richard H Byrd, Gillian M Chin, Jorge Nocedal, and Yuchen Wu. Sample size selection in optimization methods for machine learning. *Mathematical programming*, 134(1):127–155, 2012. doi:10.1007/s10107-012-0572-5.

[27] David Campion. Text classification: Be lazy, use prodigy, 2018. URL: https://medium.com/@david.campion/text-classification-be-lazy-use-prodigy-b0f9d00e9495.

[28] Gregory Canal, Stefano Fenu, and Christopher Rozell. Active ordinal querying for tuplewise similarity learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 3332–3340, 2020. doi:10.1609/aaai.v34i04.5734.

[29] Gregory Canal, Andy Massimino, Mark Davenport, and Christopher Rozell. Active embedding search via noisy paired comparisons. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 902–911. PMLR, 09–15 Jun 2019. URL: https://proceedings.mlr.press/v97/canal19a.html.

[30] Venkat Chandrasekaran, Benjamin Recht, Pablo A Parrilo, and Alan S Willsky. The convex geometry of linear inverse problems. *Foundations of Computational mathematics*, 12(6):805–849, 2012. doi:10.1007/s10208-012-9135-7.

[31] Olivier Chapelle and Lihong Li. An empirical evaluation of thompson sampling. In J. Shawe-Taylor, R. Zemel, P. Bartlett, F. Pereira, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 24. Curran Associates, Inc., 2011. URL: https://proceedings.neurips.cc/paper/2011/file/e53a0a2978c28872a4505bdb51db06dc-Paper.pdf.

[32] Zachary Charles and Dimitris Papailiopoulos. Stability and generalization of learning algorithms that converge to global optima. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 745–754. PMLR, 10–15 Jul 2018. URL: http://proceedings.mlr.press/v80/charles18a.html.

[33] Pratik Chaudhari, Anna Choromanska, Stefano Soatto, Yann LeCun, Carlo Baldassi, Christian Borgs, Jennifer Chayes, Levent Sagun, and Riccardo Zecchina. Entropy-SGD: biasing gradient descent into wide valleys. 2019(12), dec 2019. doi:10.1088/1742-5468/ab39d9.

[34] Chia-Yu Chen, Jungwook Choi, Daniel Brand, Ankur Agrawal, Wei Zhang, and Kailash Gopalakrishnan. AdaComp : Adaptive residual gradient compression for data-parallel distributed training. volume 32, Apr. 2018. URL: https://ojs.aaai.org/index.php/AAAI/article/view/11728.

[35] Rob Chew, Michael Wenger, Caroline Kery, Jason Nance, Keith Richards, Emily Hadley, and Peter Baumgartner. Smart: An open source data labeling platform for supervised learning. *Journal of Machine Learning Research*, 20(82):1–5, 2019. URL: http://jmlr.org/papers/v20/18-859.html.

[36] Kanchan Chowdhury, Ankita Sharma, and Arun Deepak Chandrasekar. Evaluating deep learning in SystemML using layer-wise adaptive rate scaling (LARS) optimizer. *arXiv preprint arXiv:2102.03018*, 2021. URL: https://arxiv.org/abs/2102.03018.

[37] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. Fast and accurate deep network learning by exponential linear units (elus). *arXiv preprint arXiv:1511.07289*, 2015. URL: https://arxiv.org/abs/1511.07289.

[38] Andrew Cotter, Ohad Shamir, Nati Srebro, and Karthik Sridharan. Better mini-batch algorithms via accelerated gradient methods. In *Advances in neural information processing systems*, pages 1647–1655, 2011.

[39] Jason Jinquan Dai, Yiheng Wang, Xin Qiu, Ding Ding, Yao Zhang, Yanzhang Wang, Xianyan Jia, Cherry Li Zhang, Yan Wan, Zhichao Li, Jiao Wang, Shengsheng Huang, Zhongyuan Wu, Yang Wang, Yuhao Yang, Bowen She, Dongjie Shi, Qi Lu, Kai Huang, and Guoqiong Song. Bigdl: A distributed deep learning framework for big data. In

Proceedings of the ACM Symposium on Cloud Computing, SoCC '19, page 50–60, 2019. doi:10.1145/3357223.3362707.

[40] Lisandro D Dalcin, Rodrigo R Paz, Pablo A Kler, and Alejandro Cosimo. Parallel distributed computing using Python. *Advances in Water Resources*, 34(9):1124–1139, 2011. doi:10.1016/j.advwatres.2011.04.013.

[41] Dask Development Team. *Dask: Library for dynamic task scheduling*, 2016. URL: https://dask.org.

[42] Soham De, Abhay Yadav, David Jacobs, and Tom Goldstein. Automated Inference with Adaptive Batches. In Aarti Singh and Jerry Zhu, editors, *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics*, volume 54 of *Proceedings of Machine Learning Research*, pages 1504–1513. PMLR, 20–22 Apr 2017. URL: https://proceedings.mlr.press/v54/de17a.html.

[43] Christopher De Sa, Matthew Feldman, Christopher Ré, and Kunle Olukotun. Understanding and optimizing asynchronous low-precision stochastic gradient descent. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pages 561–574. ACM, 2017. doi:10.1145/3079856.3080248.

[44] 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. URL: https://arxiv.org/abs/1803.03383.

[45] Christopher M De Sa, Ce Zhang, Kunle Olukotun, Christopher Ré, and Christopher Ré. Taming the wild: A unified analysis of hogwild-style algorithms. In C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 28. Curran Associates, Inc.,

2015. URL: https://proceedings.neurips.cc/paper/2015/hash/98986c005e5def2da341b4e0627d4712-Abstract.html.

[46] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc' aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, Quoc Le, and Andrew Ng. Large scale distributed deep networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in neural information processing systems*, volume 25, pages 1223–1231. Curran Associates, Inc., 2012. URL: https://proceedings.neurips.cc/paper/2012/hash/6aca97005c68f1206823815f66102863-Abstract.html.

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

[48] Aditya Devarakonda, Maxim Naumov, and Michael Garland. Adabatch: Adaptive batch sizes for training deep neural networks. *arXiv preprint arXiv:1712.02029*, 2017. URL: https://arxiv.org/abs/1712.02029.

[49] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159, 2011. URL: http://jmlr.org/papers/v12/duchi11a.html.

[50] Stefan Falkner, Aaron Klein, and Frank Hutter. BOHB: Robust and efficient hyperparameter optimization at scale. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 1437–1446, Stockholmsmässan, Stockholm

Sweden, 10–15 Jul 2018. PMLR. URL: http://proceedings.mlr.press/v80/falkner18a.html.

[51] Sarah Filippi, Olivier Cappe, Aurélien Garivier, and Csaba Szepesvári. Parametric bandits: The generalized linear case. In J. Lafferty, C. Williams, J. Shawe-Taylor, R. Zemel, and A. Culotta, editors, *Advances in Neural Information Processing Systems*, volume 23. Curran Associates, Inc., 2010. URL: https://proceedings.neurips.cc/paper/2010/file/c2626d850c80ea07e7511bbae4c76f4b-Paper.pdf.

[52] Michael P Friedlander and Mark Schmidt. Hybrid deterministic-stochastic methods for data fitting. *SIAM Journal on Scientific Computing*, 34(3):A1380–A1405, 2012. doi:10.1137/110830629.

[53] Nidham Gazagnadou, Robert Gower, and Joseph Salmon. Optimal mini-batch and step sizes for SAGA. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 2142–2150. PMLR, 09–15 Jun 2019. URL: http://proceedings.mlr.press/v97/gazagnadou19a.html.

[54] Saeed Ghadimi and Guanghui Lan. Stochastic first-and zeroth-order methods for nonconvex stochastic programming. *SIAM Journal on Optimization*, 23(4):2341–2368, 2013. doi:10.1137/120880811.

[55] R Gitlin, J Mazo, and M Taylor. On the design of gradient algorithms for digitally implemented adaptive filters. *IEEE Transactions on Circuit Theory*, 20(2):125–136, 1973. doi:10.1109/TCT.1973.1083627.

[56] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In Yee Whye Teh and Mike Titterington, editors, *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, volume 9

of *Proceedings of Machine Learning Research*, pages 249–256, Chia Laguna Resort, Sardinia, Italy, 13–15 May 2010. PMLR. URL: https://proceedings.mlr.press/v9/glorot10a.html.

[57] Noah Golmant, Nikita Vemuri, Zhewei Yao, Vladimir Feinberg, Amir Gholami, Kai Rothauge, Michael W Mahoney, and Joseph Gonzalez. On the computational inefficiency of large batch sizes for stochastic gradient descent. *arXiv preprint arXiv:1811.12941*, 2018. URL: https://arxiv.org/abs/1811.12941.

[58] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2018. URL: https://arxiv.org/pdf/1706.02677.pdf.

[59] Andreas Griewank. On automatic differentiation. *Mathematical Programming: recent developments and applications*, 6:83–107, 1989. URL: https://ftp.mcs.anl.gov/pub/tech_reports/reports/P10.pdf.

[60] Demjan Grubic, Leo Tam, Dan Alistarh, and Ce Zhang. Synchronous multi-GPU deep learning with low-precision communication: An experimental study. In *Proceedings of the 21st International Conference on Extending Database Technology*, pages 145 – 156, 2018. doi:10.3929/ethz-b-000319485.

[61] Elad Hazan, Adam Kalai, Satyen Kale, and Amit Agarwal. Logarithmic regret algorithms for online convex optimization. In *Proceedings of the 19th Annual Conference on Learning Theory*, COLT'06, page 499–513. Springer-Verlag, 2006. doi:10.1007/11776420_37.

[62] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE*

international conference on computer vision, pages 1026–1034, 2015. URL: https://openaccess.thecvf.com/content_iccv_2015/html/He_Delving_Deep_into_ICCV_2015_paper.html.

[63] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*, pages 770–778, June 2016. URL: https://openaccess.thecvf.com/content_cvpr_2016/html/He_Deep_Residual_Learning_CVPR_2016_paper.html.

[64] Eric Heim, Matthew Berger, Lee Seversky, and Milos Hauskrecht. Active perceptual similarity modeling with auxiliary information. *arXiv preprint arXiv:1511.02254*, 2015. URL: https://arxiv.org/pdf/1511.02254.pdf.

[65] Eric Heim, Matthew Berger, Lee M Seversky, and Milos Hauskrecht. Efficient online relative comparison kernel learning. In *Proceedings of the 2015 SIAM International Conference on Data Mining*, pages 271–279. SIAM, 2015. doi:10.1137/1.9781611974010.31.

[66] Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q. Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*, volume 1, page 3, July 2017. URL: https://openaccess.thecvf.com/content_cvpr_2017/html/Huang_Densely_Connected_Convolutional_CVPR_2017_paper.html.

[67] Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *International Conference on Learning and Intelligent Optimization*, pages 507–523. Springer, 2011. doi:10.1007/978-3-642-25566-3_40.

[68] Max Jaderberg, Andrea Vedaldi, and Andrew Zisserman. Speeding up convolutional neural networks with low rank expansions. In *Proceedings of the British Machine Vision Conference*. BMVA Press, 2014. doi:10.5244/C.28.88.

[69] Lalit Jain, Kevin G Jamieson, and Rob Nowak. Finite sample prediction and recovery bounds for ordinal embedding. In D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 29, pages 2711–2719. Curran Associates, Inc., 2016. URL: https://papers.nips.cc/paper/2016/hash/4e0d67e54ad6626e957d15b08ae128a6-Abstract.html.

[70] Kevin Jamieson, Matthew Malloy, Robert Nowak, and Sébastien Bubeck. lil' ucb : An optimal exploration algorithm for multi-armed bandits. In Maria Florina Balcan, Vitaly Feldman, and Csaba Szepesvári, editors, *Proceedings of The 27th Conference on Learning Theory*, volume 35 of *Proceedings of Machine Learning Research*, pages 423–439, Barcelona, Spain, 13–15 Jun 2014. PMLR. URL: https://proceedings.mlr.press/v35/jamieson14.html.

[71] Kevin G Jamieson, Lalit Jain, Chris Fernandez, Nicholas J. Glattard, and Rob Nowak. NEXT: A System for Real-World Development, Evaluation, and Application of Active Learning. In C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 28. Curran Associates, Inc., 2015. URL: https://proceedings.neurips.cc/paper/2015/file/89ae0fe22c47d374bc9350ef99e01685-Paper.pdf.

[72] Kevin G Jamieson and Robert D Nowak. Low-dimensional embedding using adaptively selected ordinal data. In *2011 49th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, page 1077–1077, 2011. doi:10.1109/Allerton.2011.6120287.

[73] Stanislaw Jastrzebski, Zachary Kenton, Devansh Arpit, Nicolas Ballas, Asja Fischer, Yoshua Bengio, and Amos Storkey. Three factors influencing minima in sgd. *arXiv preprint arXiv:1711.04623*, 2017. URL: https://arxiv.org/abs/1711.04623.

[74] Kaiyi Ji, Zhe Wang, Bowen Weng, Yi Zhou, Wei Zhang, and Yingbin Liang. History-gradient aided batch size adaptation for variance reduced algorithms. In Hal Daumé III and Aarti Singh, editors, *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 4762–4772. PMLR, 13–18 Jul 2020. URL: https://proceedings.mlr.press/v119/ji20a.html.

[75] Xianyan Jia, Shutao Song, Wei He, Yangzihao Wang, Haidong Rong, Feihu Zhou, Liqiang Xie, Zhenyu Guo, Yuanzhou Yang, and Liwei Yu. Highly scalable deep learning training system with mixed-precision: Training imagenet in four minutes. *arXiv preprint arXiv:1807.11205*, 2018. URL: https://arxiv.org/abs/1807.11205.

[76] Chi Jin, Rong Ge, Praneeth Netrapalli, Sham M. Kakade, and Michael I. Jordan. How to escape saddle points efficiently. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 1724–1732. PMLR, 06–11 Aug 2017. URL: https://proceedings.mlr.press/v70/jin17a.html.

[77] Rie Johnson and Tong Zhang. Accelerating stochastic gradient descent using predictive variance reduction. In C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 26. Curran Associates, Inc., 2013. URL: https://proceedings.neurips.cc/paper/2013/file/ac1dd209cbcc5e5d1c6e28598e8cbbe8-Paper.pdf.

[78] Tyler Johnson, Pulkit Agrawal, Haijie Gu, and Carlos Guestrin. AdaScale SGD: A user-friendly algorithm for distributed training. In Hal Daumé III and Aarti Singh, editors, *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 4911–4920. PMLR, 13–18 Jul 2020. URL: https://proceedings.mlr.press/v119/johnson20a.html.

[79] Kwang-Sung Jun, Aniruddha Bhargava, Robert Nowak, and Rebecca Willett. Scalable generalized linear bandits: Online computation and hashing. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 98–108, 2017. URL: https://proceedings.neurips.cc/paper/2017/file/28dd2c7955ce926456240b2ff0100bde-Paper.pdf.

[80] Hamed Karimi, Julie Nutini, and Mark Schmidt. Linear convergence of gradient and proximal-gradient methods under the Polyak-łojasiewicz condition. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, page 795–795, 2016. doi:10.1007/978-3-319-46128-1_50.

[81] Sumeet Katariya, Lalit Jain, Nandana Sengupta, James Evans, and Robert Nowak. Adaptive sampling for coarse ranking. In Amos Storkey and Fernando Perez-Cruz, editors, *Proceedings of the Twenty-First International Conference on Artificial Intelligence and Statistics*, volume 84 of *Proceedings of Machine Learning Research*, pages 1839–1848. PMLR, 09–11 Apr 2018. URL: https://proceedings.mlr.press/v84/katariya18a.html.

[82] Emilie Kaufmann, Olivier Cappé, and Aurélien Garivier. On the complexity of best-arm identification in multi-armed bandit models. *Journal of Machine Learning Research*, 17(1):1–42, 2016. URL: http://jmlr.org/papers/v17/kaufman16a.html.

[83] 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. *arXiv preprint arXiv:1609.04836*, 2016. URL: https://arxiv.org/abs/1609.04836.

[84] Ahmed Khaled, Othmane Sebbouh, Nicolas Loizou, Robert M Gower, and Peter Richtárik. Unified analysis of stochastic gradient methods for composite convex and smooth optimization. *arXiv preprint arXiv:2006.11573*, 2020. URL: https://arxiv.org/abs/2006.11573.

[85] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014. URL: https://arxiv.org/pdf/1412.6980.pdf.

[86] A. Klein, S. Falkner, N. Mansur, and F. Hutter. Robo: A flexible and robust bayesian optimization framework in python. In *NIPS 2017 Bayesian Optimization Workshop*, December 2017. URL: https://github.com/automl/RoBO.

[87] Aaron Klein, Stefan Falkner, Simon Bartels, Philipp Hennig, and Frank Hutter. Fast bayesian optimization of machine learning hyperparameters on large datasets. *arXiv preprint arXiv:1605.07079*, 2016. URL: https://arxiv.org/abs/1605.07079.

[88] Jakub Konečnỳ, H Brendan McMahan, Felix X Yu, Peter Richtárik, Ananda Theertha Suresh, and Dave Bacon. Federated learning: Strategies for improving communication efficiency. In *NIPS Workshop on Private Multi-Party Machine Learning*, 2016. URL: https://arxiv.org/abs/1610.05492.

[89] Jakub Konečnỳ and Peter Richtárik. Randomized distributed mean estimation: Accuracy vs communication. *Frontiers in Applied Mathematics and Statistics*, 4:62, 2018. doi:10.3389/fams.2018.00062.

[90] Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images (chapter 3). Technical report, University of Toronto, 2009. URL: https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf.

[91] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012. URL: https://papers.nips.cc/paper/2012/hash/c399862d3b9d6b76c8436e924a68c45b-Abstract.html.

[92] Manuel Lagunas, Sandra Malpica, Ana Serrano, Elena Garces, Diego Gutierrez, and Belen Masia. A similarity measure for material appearance. *ACM Trans. Graph.*, 38(4), July 2019. doi:10.1145/3306346.3323036.

[93] Rémi Leblond, Fabian Pedregosa, and Simon Lacoste-Julien. ASAGA: Asynchronous Parallel SAGA. In Aarti Singh and Jerry Zhu, editors, *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics*, volume 54 of *Proceedings of Machine Learning Research*, pages 46–54. PMLR, 20–22 Apr 2017. URL: https://proceedings.mlr.press/v54/leblond17a.html.

[94] Lihong Li, Yu Lu, and Dengyong Zhou. Provably optimal algorithms for generalized linear contextual bandits. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine*

Learning Research, pages 2071–2080. PMLR, 06–11 Aug 2017. URL: https://proceedings.mlr.press/v70/li17c.html.

[95] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. *Journal of Machine Learning Research*, 18(185):1–52, 2018. URL: http://jmlr.org/papers/v18/16-558.html.

[96] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 583–598, Broomfield, CO, October 2014. USENIX Association. URL: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/li_mu.

[97] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. Pytorch distributed: Experiences on accelerating data parallel training. *Proc. VLDB Endow.*, 13(12):3005–3018, 2020. URL: https://research.fb.com/wp-content/uploads/2020/08/PyTorch-Distributed-Experiences-on-Accelerating-Data-Parallel-Training.pdf, doi:10.14778/3415478.3415530.

[98] Yujun Lin, Song Han, Huizi Mao, Yu Wang, and William Dally. Deep gradient compression: Reducing the communication bandwidth for distributed training. 2018. URL: https://openreview.net/pdf?id=SkhQHMW0W.

[99] Yu-Chen Lo, Stefano E Rensi, Wen Torng, and Russ B Altman. Ma-

chine learning in chemoinformatics and drug discovery. *Drug discovery today*, 23(8):1538–1546, 2018. doi:10.1016/j.drudis.2018.05.010.

[100] Michael Lohaus, Philipp Hennig, and Ulrike von Luxburg. Uncertainty estimates for ordinal embeddings. *arXiv preprint arXiv:1906.11655*, 2019. URL: https://arxiv.org/abs/1906.11655.

[101] Ke Ma, Jinshan Zeng, Jiechao Xiong, Qianqian Xu, Xiaochun Cao, Wei Liu, and Yuan Yao. Stochastic non-convex ordinal embedding with stabilized barzilai-borwein step size. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018. URL: https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/16082/16595.

[102] Ke Ma, Jinshan Zeng, Jiechao Xiong, Qianqian Xu, Xiaochun Cao, Wei Liu, and Yuan Yao. Fast stochastic ordinal embedding with variance reduction and adaptive step size. *IEEE Transactions on Knowledge and Data Engineering*, 33(6):2467–2478, 2021. doi:10.1109/TKDE.2019.2956700.

[103] Andrew L. Maas, Awni Y. Hannun, and Andrew Y. Ng. Rectifier nonlinearities improve neural network acoustic models. In *ICML Workshop on Deep Learning for Audio, Speech and Language Processing*, 2013. URL: https://ai.stanford.edu/~amaas/papers/relu_hybrid_icml2013_final.pdf.

[104] Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-SNE. *Journal of machine learning research*, 9(Nov):2579–2605, 2008. URL: http://jmlr.csail.mit.edu/papers/v9/vandermaaten08a.html.

[105] Maren Mahsereci and Philipp Hennig. Probabilistic line searches for stochastic optimization. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in*

Neural Information Processing Systems 28, pages 181–189. Curran Associates, Inc., 2015. URL: http://papers.nips.cc/paper/5753-probabilistic-line-searches-for-stochastic-optimization.pdf.

[106] Horia Mania, Xinghao Pan, Dimitris Papailiopoulos, Benjamin Recht, Kannan Ramchandran, and Michael I Jordan. Perturbed iterate analysis for asynchronous stochastic optimization. *SIAM Journal on Optimization*, 27(4):2202–2229, 2017. doi:10.1137/16M1057000.

[107] Donald W. Marquardt and Ronald D. Snee. Ridge regression in practice. *The American Statistician*, 29(1):3–20, 1975. doi:10.1080/00031305.1975.10479105.

[108] Jared D Martin, Adrienne Wood, William TL Cox, Scott Sievert, Robert Nowak, et al. Evidence for distinct facial signals of reward, affiliation, and dominance from both perception and production tasks. *Affective Science*, pages 1–17, 2021. doi:10.1007/s42761-020-00024-8.

[109] Blake Mason, Martina A Rau, and Robert Nowak. Cognitive task analysis for implicit knowledge about visual representations with similarity learning methods. *Cognitive science*, 43(9):e12744, 2019. doi:10.1111/cogs.12744.

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

[111] Noboru Murata. *A statistical study of on-line learning*, page 63–92. 1998. URL: https://www.researchgate.net/profile/Noboru-Murata/publication/2666659_A_Statistical_Study_on_On-line_Learning/links/09e4150b0273b4b7e8000000/A-Statistical-Study-on-On-line-Learning.pdf.

[112] Robert F Murphy. An active role for machine learning in drug development. *Nature chemical biology*, 7(6):327, 2011. doi:10.1038/nchembio.576.

[113] Vinod Nair and Geoffrey E. Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814, 2010. URL: https://icml.cc/Conferences/2010/papers/432.pdf.

[114] Arkadiui Semenovich Nemirovsky and David Borisovich Yudin. Problem complexity and method efficiency in optimization. *SIAM Review*, 27, 1983. doi:10.1137/1027074.

[115] Yurii Nesterov. *Introductory lectures on convex optimization: A basic course*, volume 87. Springer Science & Business Media, 2013.

[116] Yuval Netzer, Tao Wang, Adam Coates, Alessandro Bissacco, Bo Wu, and Andrew Y Ng. Reading digits in natural images with unsupervised feature learning. In *NIPS Workshop on Deep Learning and Unsupervised Feature Learning 2011*, 2011. URL: http://ufldl.stanford.edu/housenumbers/nips2011_housenumbers.pdf.

[117] Phuong Ha Nguyen, Lam Nguyen, and Marten van Dijk. Tight dimension independent lower bound on the expected convergence rate for diminishing step sizes in sgd. In *Advances in Neural Information Processing Systems*, pages 3665–3674, 2019. URL: https://proceedings.neurips.cc/paper/2019/hash/deb54ffb41e085fd7f69a75b6359c989-Abstract.html.

[118] Feng Niu, Benjamin Recht, Christopher Re, and Stephen J. Wright. Hogwild!: A lock-free approach to parallelizing stochastic gradient descent. In J. Shawe-Taylor, R. Zemel, P. Bartlett, F. Pereira, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 24. Curran Associates, Inc., 2011. URL: https://papers.nips.cc/paper/2011/hash/218a0aefd1d1a4be65601cc6ddc1520e-Abstract.html.

[119] Adeola Ogunleye and Qing-Guo Wang. Xgboost model for chronic kidney disease diagnosis. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 17(6):2131–2140, 2020. doi:10.1109/TCBB.2019.2911071.

[120] Genevieve B Orr. Removing noise in on-line search using adaptive batch sizes. In *Advances in Neural Information Processing Systems*, pages 232–238, 1997. URL: https://papers.nips.cc/paper/1996/file/cd758e8f59dfdf06a852adad277986ca-Paper.pdf.

[121] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. In *Neural Information Processing Systems, Workshop on Autodiff*, 2017. URL: https://openreview.net/forum?id=BJJsrmfCZ.

[122] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural*

Information Processing Systems, volume 32. Curran Associates, Inc., 2019. URL: https://proceedings.neurips.cc/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html.

[123] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, and Vincent Dubourg. Scikit-learn: Machine learning in python. *Journal of machine learning research*, 12(Oct):2825–2830, 2011. URL: http://jmlr.csail.mit.edu/papers/v12/pedregosa11a.html.

[124] Michael P. Perrone, Haidar Khan, Changhoan Kim, Anastasios Kyrillidis, Jerry Quinn, and Valentina Salapura. Optimal mini-batch size selection for fast gradient descent. *CoRR*, abs/1911.06459, 2019. URL: http://arxiv.org/abs/1911.06459.

[125] Boris T Polyak and Anatoli B Juditsky. Acceleration of stochastic approximation by averaging. *SIAM Journal on Control and Optimization*, 30(4):838–838, 1992. doi:10.1137/0330046.

[126] Boris Teodorovich Polyak. Gradient methods for minimizing of functionals. *USSR Computational Mathematics and Mathematical Physics*, 3(4):643–653, 1963. doi:10.1016/0041-5553(63)90382-3.

[127] Lutz Prechelt. Automatic early stopping using cross validation: quantifying the criteria. *Neural Networks*, 11(4):761–767, 1998. doi:10.1016/S0893-6080(98)00010-0.

[128] Hang Qi, Evan R Sparks, and Ameet Talwalkar. Paleo: A performance model for deep neural networks. In *ICLR (Poster)*, 2017. URL: https://talwalkarlab.github.io/paleo/.

[129] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving language understanding by generative pre-training. Blog

post: https://openai.com/blog/language-unsupervised/, 2018. URL: https://cdn.openai.com/research-covers/language-unsupervised/language_understanding_paper.pdf.

[130] Pranav Rajpurkar, Jeremy Irvin, Kaylie Zhu, Brandon Yang, Hershel Mehta, Tony Duan, Daisy Ding, Aarti Bagul, Curtis Langlotz, Katie Shpanskaya, et al. CheXNet: Radiologist-level pneumonia detection on chest x-rays with deep learning. *arXiv preprint arXiv:1711.05225*, 2017. URL: https://arxiv.org/pdf/1711.05225.pdf.

[131] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. XNOR-Net: Imagenet classification using binary convolutional neural networks. In *European Conference on Computer Vision*, pages 525–542. Springer, 2016. doi:10.1007/978-3-319-46493-0_32.

[132] Martina A Rau, Blake Mason, and Robert Nowak. How to model implicit knowledge? similarity learning methods to assess perceptions of visual representations. *International Educational Data Mining Society*, 2016. doi:10.1111/cogs.12744.

[133] Benjamin Recht, Rebecca Roelofs, Ludwig Schmidt, and Vaishaal Shankar. Do ImageNet classifiers generalize to ImageNet? In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 5389–5400. PMLR, 09–15 Jun 2019. URL: https://proceedings.mlr.press/v97/recht19a.html.

[134] Sashank J. Reddi, Ahmed Hefny, Suvrit Sra, Barnabas Poczos, and Alex Smola. Stochastic variance reduction for nonconvex optimization. In Maria Florina Balcan and Kilian Q. Weinberger, editors, *Proceedings of The 33rd International Conference on Machine Learning (ICML)*,

volume 48 of *Proceedings of Machine Learning Research*, pages 314–323, New York, New York, USA, 20–22 Jun 2016. PMLR. URL: https://proceedings.mlr.press/v48/reddi16.html.

[135] Cèdric Renggli, Saleh Ashkboos, Mehdi Aghagolzadeh, Dan Alistarh, and Torsten Hoefler. SparCML: High-performance sparse communication for machine learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3295500.3356222.

[136] Herbert Robbins and Sutton Monro. A stochastic approximation method. *The annals of mathematical statistics*, page 400–407, 1951. URL: https://www.jstor.org/stable/pdf/2236626.pdf.

[137] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015. doi:10.1007/s11263-015-0816-y.

[138] Christopher De Sa, Christopher Re, and Kunle Olukotun. Global convergence of stochastic gradient descent for some non-convex matrix problems. In Francis Bach and David Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 2332–2341, Lille, France, 07–09 Jul 2015. PMLR. URL: https://proceedings.mlr.press/v37/sa15.html.

[139] Tara N Sainath, Brian Kingsbury, Vikas Sindhwani, Ebru Arisoy, and Bhuvana Ramabhadran. Low-rank matrix factorization for deep neural network training with high-dimensional output targets. In *Acoustics,*

Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on, pages 6655–6659. IEEE, 2013. doi:10.1109/ICASSP.2013.6638949.

[140] Ruslan Salakhutdinov. Deep learning tutorial at the Simons Institute, Berkeley, 2017, 2017. URL: https://simons.berkeley.edu/talks/ruslan-salakhutdinov-01-26-2017-1.

[141] Tom Schaul, Sixin Zhang, and Yann LeCun. No more pesky learning rates. In Sanjoy Dasgupta and David McAllester, editors, *Proceedings of the 30th International Conference on Machine Learning*, volume 28 of *Proceedings of Machine Learning Research*, pages 343–351, Atlanta, Georgia, USA, 17–19 Jun 2013. PMLR. URL: https://proceedings.mlr.press/v28/schaul13.html.

[142] Mark Schmidt, Nicolas Le Roux, and Francis Bach. Minimizing finite sums with the stochastic average gradient. *Mathematical Programming*, page 83–112, 2017. doi:10.1007/s10107-016-1030-6.

[143] Wilko Schwarting, Javier Alonso-Mora, and Daniela Rus. Planning and decision-making for autonomous vehicles. *Annual Review of Control, Robotics, and Autonomous Systems*, 1:187–210, 2018. doi:10.1146/annurev-control-060117-105157.

[144] Frank Seide, Hao Fu, Jasha Droppo, Gang Li, and Dong Yu. 1-bit stochastic gradient descent and its application to data-parallel distributed training of speech DNNs. In *Fifteenth Annual Conference of the International Speech Communication Association*, 2014. URL: https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/IS140694.pdf.

[145] Ayon Sen, Purav Patel, Martina A Rau, Blake Mason, Robert Nowak, Timothy T Rogers, and Xiaojin Zhu. Machine beats

human at sequencing visuals for perceptual-fluency practice. In *International Educational Data Mining Society*. ERIC, 2018. URL: http://educationaldatamining.org/files/conferences/EDM2018/papers/EDM2018_paper_94.pdf.

[146] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in TensorFlow. *arXiv preprint arXiv:1802.05799*, 2018. URL: https://arxiv.org/pdf/1802.05799.pdf.

[147] Burr Settles. *Active learning literature survey*, volume 52. University of Wisconsin, Madison, 2010. URL: http://digital.library.wisc.edu/1793/60660.

[148] Shai Shalev-Shwartz et al. *Online Learning and Online convex optimization*, volume 4. Now Publishers Inc., 2012. doi:10.1561/2200000018.

[149] Scott Sievert, Daniel Ross, Lalit Jain, Kevin Jamieson, Rob Nowak, and Robert Mankoff. NEXT: A system to easily connect crowdsourcing and adaptive data collection. In Katy Huff, David Lippa, Dillon Niederhut, and M Pacer, editors, *Proceedings of the 16th Python in Science Conference*, pages 113 – 119, 2017. doi:10.25080/shinma-7f4c6e7-010.

[150] Scott Sievert and Shrey Shah. Improving the convergence of sgd through adaptive batch sizes. *arXiv preprint arXiv:1910.08222*, 2021. URL: https://arxiv.org/abs/1910.08222.

[151] Leslie N Smith. No more pesky learning rate guessing games. *CoRR, abs/1506.01186*, 5, 2015. URL: https://arxiv.org/abs/1506.01186.

[152] Samuel L. Smith, Pieter-Jan Kindermans, and Quoc V. Le. Don't decay the learning rate, increase the batch size. In *International Conference*

on Learning Representations, 2018. URL: https://openreview.net/forum?id=B1Yy1BxCZ.

[153] Samuel L. Smith and Quoc V. Le. A bayesian perspective on generalization and stochastic gradient descent. In *International Conference on Learning Representations*, 2018. URL: https://openreview.net/forum?id=BJij4yg0Z.

[154] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 2951–2959. Curran Associates, Inc., 2012. URL: http://papers.nips.cc/paper/4522-practical-bayesian-optimization-of-machine-learning-algorithms.pdf.

[155] Evan R. Sparks, Ameet Talwalkar, Virginia Smith, Jey Kottalam, Xinghao Pan, Joseph Gonzalez, Michael J. Franklin, Michael I. Jordan, and Tim Kraska. Mli: An api for distributed machine learning. In *2013 IEEE 13th International Conference on Data Mining*, pages 1187–1192, 2013. doi:10.1109/ICDM.2013.158.

[156] Nikko Strom. Scalable distributed DNN training using commodity GPU cloud computing. In *Sixteenth Annual Conference of the International Speech Communication Association*, 2015. URL: https://www.isca-speech.org/archive_v0/interspeech_2015/papers/i15_1488.pdf.

[157] Ananda Theertha Suresh, Felix X. Yu, Sanjiv Kumar, and H. Brendan McMahan. Distributed mean estimation with limited communication. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 3329–3337. PMLR, 06–11

Aug 2017. URL: https://proceedings.mlr.press/v70/suresh17a.html.

[158] Omer Tamuz, Ce Liu, Serge Belongie, Ohad Shamir, and Adam Tauman Kalai. Adaptively learning the crowd kernel. In *Proceedings of the 28th International Conference on International Conference on Machine Learning*, ICML'11, page 673–680, Madison, WI, USA, 2011. Omnipress. URL: http://www.icml-2011.org/papers/395_icmlpaper.pdf.

[159] Ervin Tanczos, Robert Nowak, and Bob Mankoff. A kl-lucb algorithm for large-scale crowdsourcing. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017. URL: https://proceedings.neurips.cc/paper/2017/file/c02f9de3c2f3040751818aacc7f60b74-Paper.pdf.

[160] Min Tang, Xiaoqiang Luo, and Salim Roukos. Active learning for statistical natural language parsing. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, pages 120–127, Philadelphia, Pennsylvania, USA, July 2002. Association for Computational Linguistics. URL: https://aclanthology.org/P02-1016, doi:10.3115/1073083.1073105.

[161] Robert Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society: Series B (Methodological)*, 58(1):267–288, 1996. doi:10.1111/j.2517-6161.1996.tb02080.x.

[162] Yusuke Tsuzuku, Hiroto Imachi, and Takuya Akiba. Variance-based gradient compression for efficient distributed deep learning. *arXiv preprint arXiv:1802.06058*, 2018. URL: https://arxiv.org/abs/1802.06058.

[163] Laurens Van Der Maaten and Kilian Weinberger. Stochastic triplet embedding. In *2012 IEEE International Workshop on Machine Learning for Signal Processing*, pages 1–6. IEEE, 2012. URL: http://www.cs.cornell.edu/~kilian/papers/stochastictriplet.pdf, doi:10.1109/MLSP.2012.6349720.

[164] Stefan Van Der Walt, S Chris Colbert, and Gael Varoquaux. The numpy array: a structure for efficient numerical computation. *Computing in science & engineering*, 13(2):22–30, 2011. doi:10.1109/MCSE.2011.37.

[165] Leena Chennuru Vankadara, Siavash Haghiri, Michael Lohaus, Faiz Ul Wahab, and Ulrike von Luxburg. Insights into ordinal embedding algorithms: A systematic evaluation. *arXiv preprint arXiv:1912.01666*, 2019. URL: https://arxiv.org/abs/1912.01666.

[166] Hongyi Wang*, Scott Sievert*, Shengchao Liu, Zachary Charles, Dimitris Papailiopoulos, and Stephen Wright. Atomo: Communication-efficient learning via atomic sparsification. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018. URL: https://proceedings.neurips.cc/paper/2018/file/33b3214d792caf311e1f00fd22b392c5-Paper.pdf.

[167] Linnan Wang, Wei Wu, Junyu Zhang, Hang Liu, George Bosilca, Maurice Herlihy, and Rodrigo Fonseca. FFT-based gradient sparsification for the distributed training of deep neural networks. In *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '20, page 113–124, 2020. doi:10.1145/3369583.3392681.

[168] Jianqiao Wangni, Jialei Wang, Ji Liu, and Tong Zhang. Gradient sparsification for communication-efficient distributed optimization. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018. URL: https://proceedings.neurips.cc/paper/2018/file/3328bdf9a4b9504b9398284244fe97c2-Paper.pdf.

[169] Rachel Ward, Xiaoxia Wu, and Leon Bottou. AdaGrad stepsizes: Sharp convergence over nonconvex landscapes. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 6677–6686. PMLR, 09–15 Jun 2019. URL: https://proceedings.mlr.press/v97/ward19a.html.

[170] Martin Wattenberg, Fernanda Viégas, and Ian Johnson. How to use t-SNE effectively. *Distill*, 2016. URL: http://distill.pub/2016/misread-tsne, doi:10.23915/distill.00002.

[171] Wei Wen, Cong Xu, Feng Yan, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. Terngrad: Ternary gradients to reduce communication in distributed deep learning. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017. URL: https://proceedings.neurips.cc/paper/2017/hash/89fcd07f20b6785b92134bd6c1d0fa42-Abstract.html.

[172] Simon Wiesler, Alexander Richard, Ralf Schluter, and Hermann Ney. Mean-normalized stochastic gradient for large-scale deep learning. In *Acoustics, Speech and Signal Processing (ICASSP), 2014 IEEE International Conference on*, pages 180–184. IEEE, 2014. doi:10.1109/ICASSP.2014.6853582.

[173] Ashia C Wilson, Rebecca Roelofs, Mitchell Stern, Nati Srebro, and Benjamin Recht. The marginal value of adaptive gradient methods in machine learning. 30, 2017. URL: https://proceedings.neurips.cc/paper/2017/file/81b3833e2504647f9d794f7d7b9bf341-Paper.pdf.

[174] Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. *arXiv preprint arXiv:1708.07747*, 2017. URL: https://arxiv.org/abs/1708.07747.

[175] Jian Xue, Jinyu Li, and Yifan Gong. Restructuring of deep neural network acoustic models with singular value decomposition. In *Interspeech*, pages 2365–2369, 2013. URL: https://www.microsoft.com/en-us/research/wp-content/uploads/2013/01/svd_v2.pdf.

[176] Dong Yin, Ashwin Pananjady, Max Lam, Dimitris Papailiopoulos, Kannan Ramchandran, and Peter Bartlett. Gradient diversity: a key ingredient for scalable distributed learning. In Amos Storkey and Fernando Perez-Cruz, editors, *Proceedings of the Twenty-First International Conference on Artificial Intelligence and Statistics*, volume 84 of *Proceedings of Machine Learning Research*, pages 1998–2007. PMLR, 09–11 Apr 2018. URL: https://proceedings.mlr.press/v84/yin18a.html.

[177] Chris Ying, Sameer Kumar, Dehao Chen, Tao Wang, and Youlong Cheng. Image classification at supercomputer scale. *arXiv preprint arXiv:1811.06992*, 2018. URL: https://arxiv.org/pdf/1811.06992.pdf.

[178] Yang You, Igor Gitman, and Boris Ginsburg. Large batch training of convolutional networks. *arXiv preprint arXiv:1708.03888*, 2017. URL: https://arxiv.org/abs/1708.03888.

[179] Yang You, Jing Li, Sashank Reddi, Jonathan Hseu, Sanjiv Kumar, Srinadh Bhojanapalli, Xiaodan Song, James Demmel, Kurt Keutzer, and Cho-Jui Hsieh. Large batch optimization for deep learning: Training bert in 76 minutes. In *International Conference on Learning Representations*, 2020. URL: https://openreview.net/forum?id=Syx4wnEtvH.

[180] Sergey Zagoruyko and Nikos Komodakis. Wide residual networks. In *British Machine Vision Conference 2016*. British Machine Vision Association, 2016. doi:10.5244/C.30.87.

[181] Matthew D Zeiler. Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012. URL: https://arxiv.org/pdf/1212.5701.pdf.

[182] C Zhang, S Bengio, M Hardt, B Recht, and O Vinyals. Understanding deep learning requires rethinking generalization. In *International Conference on Learning Representations*, 2017. URL: https://openreview.net/forum?id=Sy8gdB9xx.

[183] Guodong Zhang, Lala Li, Zachary Nado, James Martens, Sushant Sachdeva, George Dahl, Chris Shallue, and Roger B Grosse. Which algorithmic choices matter at which batch sizes? insights from a noisy quadratic model. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019. URL: https://proceedings.neurips.cc/paper/2019/file/e0eacd983971634327ae1819ea8b6214-Paper.pdf.

[184] Hantian Zhang, Jerry Li, Kaan Kara, Dan Alistarh, Ji Liu, and Ce Zhang. ZipML: Training linear models with end-to-end low precision, and a little bit of deep learning. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on*

Machine Learning (ICML), volume 70 of *Proceedings of Machine Learning Research*, pages 4035–4043. PMLR, 06–11 Aug 2017. URL: https://proceedings.mlr.press/v70/zhang17e.html.

[185] Huan Zhang, Cho-Jui Hsieh, and Venkatesh Akella. Hogwild++: A new mechanism for decentralized asynchronous stochastic gradient descent. In *2016 IEEE 16th International Conference on Data Mining (ICDM)*, pages 629–638, 2016. doi:10.1109/ICDM.2016.0074.

[186] Pan Zhou, Xiaotong Yuan, and Jiashi Feng. New insight into hybrid stochastic gradient descent: Beyond with-replacement sampling and convexity. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 1234–1243. Curran Associates, Inc., 2018. URL: http://papers.nips.cc/paper/7399-new-insight-into-hybrid-stochastic-gradient-descent-beyond-with-replace pdf.

[187] 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. URL: https://arxiv.com/abs/1606.06160.

[188] Jingbo Zhu, Huizhen Wang, Tianshun Yao, and Benjamin K Tsou. Active learning with sampling by uncertainty and density for word sense disambiguation and text classification. In *Proceedings of the 22nd International Conference on Computational Linguistics (Coling 2008)*, pages 1137–1144, Manchester, UK, August 2008. Coling 2008 Organizing Committee. URL: https://aclanthology.org/C08-1143.

A HYPERBAND

This appendix mentions some details for our experiments with Hyperband. For complete detail, see https://github.com/stsievert/dask-hyperband-comparison.

A.1 Serial simulation detail

Here's the creation of the dataset:

```
from dask_ml.model_selection import train_test_split
X, y = make_4_circles(num=60e3)
X_train, X_test, y_train, y_test = train_test_split(
   X, y, test_size=int(10e3)
)
```

A visualization of this dataset is in Figure 2.2a.

Scikit-learn's fully-connected neural network is used, their `MLPClassifier` which has several hyperparameters. Only one affects the architecture of the best model: `hidden_layer_sizes`, which controls the number of layers and number of neurons in each layer.

There are 5 values for the hyperparameter. It is varied so the neural network has 24 neurons but varies the network depth and the width of each layer. Two choices are 12 neurons in 2 layers or 6 neurons in four layers. One choice has 12 neurons in the first layer, 6 in the second, and 3 in third and fourth layers.

The other six hyperparameters control finding the best model and do not influence model architecture. 3 of these hyperparameters are continuous and 3 are discrete (of which there are 10 unique combinations). Details are in Appendix A.3. These hyperparameters include the batch size, learning rate (and decay schedule) and a regularization parameter:

```
from sklearn.neural_network import MLPClassifier
model = MLPClassifier(...)
params = {'batch_size': [32, 64, ..., 512], ...}
print(params.keys())
# dict_keys([
#     "batch_size",  # 5 choices
#     "learning_rate",  # 2 choices
#     "hidden_layer_sizes",  # 5 choices
#     "alpha",  # cnts
#     "power_t",  # cnts
#     "momentum",  # cnts
#     "learning_rate_init"  # cnts
# ])
```

A.1.1 Usage: rule of thumb on `HyperbandSearchCV`'s inputs

Now that we've specified the search space and model architecture, let's create our `HyperbandSearchCV` object. As discussed in Section 2.4.2, `HyperbandSearchCV` only requires two parameters besides the model and data as discussed above: the number of `partial_fit` calls for each model (`max_iter`) and the number of examples each call to `partial_fit` sees (which is implicit via the Dask array chunk size `chunks`). These inputs control how many hyperparameter values are considered and how long to train the models.

The values for `max_iter` and `chunks` can be specified by a rule-of-thumb once the number of parameter to be sampled and the number of examples required to be seen by at least one model, `n_examples`. This rule of thumb is below:

```
n_examples = 50 * len(X_train)
n_params = 299
```

```
# The rule-of-thumb to determine inputs
max_iter = n_params
chunks = n_examples // n_params
```

The value of 299 is chosen because `n_params` is only approximate and the Dask array evenly chunks evenly with that value. Creation of a `HyperbandSearchCV` object and the Dask array is simple with this:

```
from dask_ml.model_selection import HyperbandSearchCV
search = HyperbandSearchCV(
   model, params,
   max_iter=max_iter, aggressiveness=4)

X_train = da.from_array(X_train, chunks=chunks)
y_train = da.from_array(y_train, chunks=chunks)
search.fit(X_train, y_train)
```

`aggressiveness=4` is chosen because this is my first time optimizing these hyperparameters – I only made one small edit to the hyperparameter search space.[1] With `max_iter`, no model sees more than `n_examples` examples as desired and Hyperband evaluates (approximately) `n_params` hyperparameter combinations.[2]

A.2 Parallel experiment detail

The inputs and desired outputs are given in Figure 2.4. This is an especially difficult problem because the noise variance varies slightly between images.

[1] For personal curiosity, I changed total number of neurons to 24 from 20 to allow the `[12, 6, 3, 3]` configuration.

[2] Exact specification is available through the `metadata` attribute of `HyperbandSearchCV`.

To protect against this, a shallow neural network is used that's slightly more complex than a linear model. This means hyperparameter optimization is not simple.

Specifically, this section will find the best hyperparameters for a model created in PyTorch [121] (with the wrapper Skorch[3] for an image denoising task. Again, some detail is mentioned in Appendix A.3 and complete details can be found at https://github.com/stsievert/dask-hyperband-comparison.

A.2.1 Model architecture & Hyperparameters

Autoencoders are a type of neural network useful for image denoising. They reduce the dimensionality of the input before expanding to the original dimension, which is similar to a lossy compression. Let's create that model and the images it will denoise:

```
# custom model definition with PyTorch
from autoencoder import Autoencoder
from dask_ml.model_selection import train_test_split
import skorch  # scikit-learn API wrapper for PyTorch

model = skorch.NeuralNetRegressor(Autoencoder, ...)

X, y = noisy_mnist(augment=5)
X_train, X_test, y_train, y_test = train_test_split(
   X, y, test_size=0.05)
```

Of course, this is a neural network so there are many hyperparameters to tune. Only one hyperparameter affects the model architecture: `estimator__activation`, which specifies the activation the neural network should use. This hyperparameter is varied between 4 different choices, all

[3] https://github.com/skorch-dev/skorch

different types of the rectified linear unit (ReLU) [113], including the leaky ReLU [103], parametric ReLU [62] and exponential linear units (ELU) [37].

The other hyperparameters all control finding the optimal model after the architecture is fixed. These hyperparameters include 3 discrete hyperparameters (with 160 unique combinations) and 3 continuous hyperparameters. Some of these hyperparameters include choices on the optimizer to use (SGD [18] or Adam [85]), initialization, regularization and optimizer hyperparameters like learning rate or momentum. Here's a brief summary:

```
params = {'optimizer': ['SGD', 'Adam'], ...}
print(params.keys())
# dict_keys([
#     "optimizer",  # 2 choices
#     "batch_size",  # 5 choices
#     "module__init",  # 4 choices
#     "module__activation",  # 4 choices
#     "optimizer__lr",  # cnts
#     "optimizer__momentum",  # cnts
#     "optimizer__weight_decay"  # cnts
# ])
```

Details are in Appendix A.3. Now, let's create our `HyperbandSearchCV` object: `HyperbandSearchCV` supports specifying `patience=True` to make a decision on how long to wait to see if scores stop increasing, as mentioned above. Let's create a `HyperbandSearchCV` object that stops training non-improving models.

A.2.2 Usage: plateau specification for non-improving models

```
from dask_ml.model_selection import HyperbandSearchCV
search = HyperbandSearchCV(
```

```
    model, params, max_iter=max_iter, patience=True)
search.fit(X_train, y_train)
```

The current implementation uses `patience=True` to choose a high value of `patience=max_iter // 3`. This is most useful for the least adaptive bracket of Hyperband (which trains a couple models to completion) and mirrors the patience of the second least adaptive bracket in Hyperband.

In these experiments, `patience=max_iter // 3` has no effect on performance. If `patience=max_iter // 6` for these experiments, there is a moderate effect on performance (`patience=max_iter // 6` obtains a model with validation loss 0.0637 instead of 0.0630 like `patience=max_iter // 3` and `patience=False`).

A.3 Hyperparameter search spaces

A.3.1 Serial Simulation

Here are some of the other hyperparameters tuned, alongside descriptions of their default values and the values chosen for tuning.

- `alpha`, a regularization term that can affect generalization. This value defaults to 10^{-4} and is tuned logarithmically between 10^{-6} and 10^{-3}
- `batch_size`, the number of examples used to approximate the gradient at each optimization iteration. This value defaults to 200 and is chosen to be one of $[32, 64, \ldots, 512]$.
- `learning_rate` controls the learning rate decay scheme, either constant or via the "`invscaling`" scheme, which has the learning rate decay like γ_0/t^p where p and γ_0 are also tuned. γ_0 defaults to 10^{-3} and is tuned logarithmically between 10^{-4} and 10^{-2}. p defaults to 0.5 and is tuned between 0.1 and 0.9.
- `momentum`, the amount of momentum to include in Nesterov's momentum [115]. This value is chosen between 0 and 1.

The learning rate scheduler used is not Adam [85] because it claims to be most useful without tuning and has reportedly has marginal gain [173].

A.3.2 Parallel Experiments

Here are some of the other hyperparameters tuned:

- `optimizer`: which optimization method should be used for training? Choices are stochastic gradient descent (SGD) [18] and Adam [85]. SGD is chosen with 5/7th probability.
- `estimator__init`: how should the estimator be initialized before training? Choices are Xavier [56] and Kaiming [62] initialization.
- `batch_size`: how many examples should the optimizer use to approximate the gradient? Choices are $[32, 64, \ldots, 512]$.
- `weight_decay`: how much of a particular type of regularization should the neural net have? Regularization helps control how well the model performs on unseen data. This value is chosen to be zero 1/6th of the time, and if not zero chosen uniformly at random between 10^{-5} and 10^{-3} logarithmically.
- `optimizer__lr`: what learning rate should the optimizer use? This is the most basic hyperparameter for the optimizer. This value is tuned between $10^{-1.5}$ and 10^{1} after some initial tuning.
- `optimizer__momentum`, which is a hyper-parameter for the SGD optimizer to incorporate Nesterov momentum [115]. This value is tuned between 0 and 1.

B.1 Gradient diversity bounds

Yin et al. introduced a measure of gradient dissimilarity called "gradient diversity" [176]. When the gradients for each examples are orthogonal, then the gradient diversity $\Delta_k = 1$ and when all the gradients are exactly the same, then $\Delta_k = 1/n$. The definition is repeated below:

Definition 6. The gradient diversity of a model $\boldsymbol{w}$ with respect to F is given by

$$\Delta(\boldsymbol{w}) := \frac{\sum_{i=1}^n \|\nabla f_i(\boldsymbol{w})\|_2^2}{\|\sum_{i=1}^n \nabla f_i(\boldsymbol{w})\|_2^2} = \frac{\sum_{i=1}^n \|\nabla f_i(\boldsymbol{w})\|_2^2}{\sum_{i=1}^n \|\nabla f_i(\boldsymbol{w})\|_2^2 + \sum_{i\neq j} \langle f_i(\boldsymbol{w}), f_j(\boldsymbol{w})\rangle}. \tag{B.1}$$

when $f_i(\boldsymbol{w}) = f(\boldsymbol{w}; \boldsymbol{x}_i)$. Let $\Delta_k := \Delta(\boldsymbol{w}_k)$ given iterates $\{\boldsymbol{w}_i\}_{i=1}^T$.

Yin et al. show that serial SGD and mini-batch SGD produce similar results with the same number of gradient evaluations [176, Theorem 3]. In this result, the batch size must obey a bound proportional to the maximum gradient diversity over *all* iterates. Let's see how gradient diversity changes as an optimization proceeds:

Theorem 11. *If F is β-smooth, the gradient diversity Δ_k obeys $\Delta_k \geq c/\|\boldsymbol{w}_k - \boldsymbol{w}^\star\|_2^2$ for $c = M_L^2/\beta^2 n$.*

Theorem 12. *If F is α-strongly convex, the gradient diversity Δ_k obeys $\Delta_k \leq c/\|\boldsymbol{w}_k - \boldsymbol{w}^\star\|_2^2$ for $c = M_U^2/\alpha^2 n$.*

Corollary 13. *If F is α-PL, then the gradient diversity Δ_k obeys $\Delta_k \leq c/(F(\boldsymbol{w}_k) - F^\star)$ for $c = M_U^2/2\alpha n$.*

Straightforward proofs of the above are given in Appendix B.1.1 and B.1.2. These proofs will rely on these statements:

Lemma 14. *If a function f is λ-strongly convex, then f is also λ-PL.*

Corollary 15 (from Lemma 1 on [176])**.** *Let $\boldsymbol{w}_k$ be a model after k updates. Let $\boldsymbol{w}_{k+1}$ be the model after a mini-batch iteration given by Equation 3.2 with batch size $B_k \leq n\delta\Delta_k + 1$ for an arbitrary δ. Then,*

$$\mathbb{E}\left[\|\boldsymbol{w}_{k+1} - \boldsymbol{w}^\star\|_2^2 \;\middle|\; \boldsymbol{w}_k\right] \leq \|\boldsymbol{w}_k - \boldsymbol{w}^\star\|_2^2 - 2\gamma_k \langle \nabla F(\boldsymbol{w}_k), \boldsymbol{w}_k - \boldsymbol{w}^\star \rangle + \frac{(1+\delta)\gamma^2 M^2(\boldsymbol{w}_k)}{B_k}$$

with equality when there are no projections.

Proof is in Appendix B.1.3.

B.1.1 Proof of Theorem 11

Proof. First, let's expand the gradient diversity term and exploit that $\nabla F(\boldsymbol{w}^\star) = 0$ when $\boldsymbol{w}^\star$ is a local minimizer or saddle point:

$$\begin{aligned}
\Delta_k &= \frac{\sum_i \|\nabla f_i(\boldsymbol{w}_k)\|_2^2}{\|\sum_i \nabla f_i(\boldsymbol{w}_k)\|_2^2} \\
&= \frac{\sum_i \|\nabla f_i(\boldsymbol{w}_k)\|_2^2}{\|n\nabla F(\boldsymbol{w}_k)\|_2^2} \\
&= \frac{\frac{1}{n}\sum_i \|\nabla f_i(\boldsymbol{w}_k)\|_2^2}{n\,\|\nabla F(\boldsymbol{w}_k) - \nabla F(\boldsymbol{w}^\star)\|_2^2}
\end{aligned}$$

Because F is β-smooth, $\|\nabla F(\boldsymbol{w}_1) - \nabla F(\boldsymbol{w}_2)\| \leq \beta\,\|\boldsymbol{w}_1 - \boldsymbol{w}_2\|$. Then,

$$\Delta_k = \frac{M^2(\boldsymbol{w}_k)}{n\,\|\nabla F(\boldsymbol{w}_k) - \nabla F(\boldsymbol{w}^\star)\|_2^2} \geq \frac{M^2(\boldsymbol{w}_k)}{n\beta^2\,\|\boldsymbol{w}_k - \boldsymbol{w}^\star\|_2^2} \geq \frac{M_L^2}{n\beta^2\,\|\boldsymbol{w}_k - \boldsymbol{w}^\star\|_2^2}$$

□

B.1.2 Proof of Theorem 12

Proof. Now, define expand gradient diversity and take advantage that $\nabla F(x^\star) = 0$ when $x^\star$ is a local minima or saddle point:

$$\begin{aligned}
\Delta_k &= \frac{\sum_i \|\nabla f_i(\boldsymbol{w}_k)\|_2^2}{\|\sum_i \nabla f_i(\boldsymbol{w}_k)\|_2^2} \\
&= \frac{\frac{1}{n}\sum_i \|\nabla f_i(\boldsymbol{w}_k)\|_2^2}{n\,\|\nabla F(\boldsymbol{w}_k)\|_2^2} \\
&= \frac{M^2(\boldsymbol{w}_k)}{n\,\|\nabla F(\boldsymbol{w}_k)\|_2^2} \\
&\leq \frac{M^2(\boldsymbol{w}_k)}{2\alpha n\,(F(\boldsymbol{w}_k) - F(\boldsymbol{w}^\star))}
\end{aligned}$$

In the context of Theorem 12, the function F is assumed to be α-strongly convex. This implies that the function F is also α-PL as shown in Lemma 14. With this, the fact that strongly convex functions grow at least quadratically can be used, so

$$\frac{M^2(\boldsymbol{w}_k)}{2\alpha n\,(F(\boldsymbol{w}_k) - F(\boldsymbol{w}^\star))} \leq \frac{M^2(\boldsymbol{w}_k)}{\alpha^2 n\,\|\boldsymbol{w}_k - \boldsymbol{w}^\star\|_2^2}$$

Then, by definition of M^2 and M_U^2, there's also

$$\Delta_k \leq \frac{M_U^2}{2\alpha n\,(F(\boldsymbol{w}_k) - F(\boldsymbol{w}^\star))} \leq \frac{M_U^2}{\alpha^2 n\,\|\boldsymbol{w}_k - \boldsymbol{w}^\star\|_2^2}$$

□

B.1.3 Proof of Lemma 14

There is a brief proof of this in Appendix B of [80]. It is expanded here for completeness.

Proof. Recall that λ-strongly convex means $\forall \boldsymbol{x} \in \mathbb{R}^d$ and $\forall \boldsymbol{y} \in \mathbb{R}^d$

$$f(\boldsymbol{y}) \geq f(\boldsymbol{x}) + \nabla f(\boldsymbol{x})^T(\boldsymbol{y} - \boldsymbol{x}) + \frac{\lambda}{2}\|\boldsymbol{y} - \boldsymbol{x}\|_2^2$$

and λ-PL means that $\frac{1}{2}\|\nabla f(\boldsymbol{x})\|_2^2 \geq \lambda(f(\boldsymbol{x}) - f(\boldsymbol{x}^\star))$.

Let's start off with the definition of strong convexity, and define $g(\boldsymbol{y}) = \nabla f(\boldsymbol{x})^T(\boldsymbol{y} - \boldsymbol{x}) + \frac{\lambda}{2}\|\boldsymbol{y} - \boldsymbol{x}\|_2^2$. Then, it's simple to see that

$$\begin{aligned} f(\boldsymbol{x}^\star) - f(\boldsymbol{x}) &\geq \nabla f(\boldsymbol{x})^T(\boldsymbol{x}^\star - \boldsymbol{x}) + \frac{\lambda}{2}\|\boldsymbol{x}^\star - \boldsymbol{x}\|_2^2 \\ &\geq \min_{\boldsymbol{y}} g(\boldsymbol{y}) \end{aligned}$$

g is a convex function, so the minimum can be obtained by setting $\nabla g(\boldsymbol{y}) = 0$. When the minimum of $g(\boldsymbol{y})$ is found, $\boldsymbol{y} = \boldsymbol{x} - \frac{1}{\lambda}\nabla f(\boldsymbol{x})$. That means that

$$\begin{aligned} \min_{\boldsymbol{y}} g(\boldsymbol{y}) &= g(\boldsymbol{x} - \lambda^{-1}\nabla f(\boldsymbol{x})) \\ &= \frac{-1}{\lambda}\|f(\boldsymbol{x})\|_2^2 + \frac{1}{2\lambda}\|\nabla f(\boldsymbol{x})\|_2^2 \\ &\geq \frac{-1}{2\lambda}\|\nabla f(\boldsymbol{x})\|_2^2 \end{aligned}$$

because $\boldsymbol{y} - \boldsymbol{x} = \frac{-1}{\lambda}\nabla f(\boldsymbol{x})$.

□

B.2 Convergence proofs

This section will analyze the convergence rate of mini-batch SGD on $F(\boldsymbol{w})$. In this, at every iteration k, B_k examples are drawn uniformly at random with repetition via $i_1^{(k)}, \ldots, i_{B_k}^{(k)}$ from the possible example indices $\{1, \ldots, n\}$. Let $S_k = \{i_1^{(k)}, \ldots, i_{B_k}^{(k)}\}$. The model is updated with $\boldsymbol{w}_{k+1} = \boldsymbol{w}_k - \gamma_k \widehat{\boldsymbol{g}}_k$ where

$$\widehat{\boldsymbol{g}}_k = \frac{1}{B_k}\sum_{i \in S_k} \nabla f_i(\boldsymbol{w}_k).$$

Note that $\mathbb{E}[\widehat{\boldsymbol{g}}_k] = \nabla F(\boldsymbol{w}_k)$. Now, let's prove Lemma 4, repeated here for convenience:

Lemma 16. *Let $c' = c/M_U^2$. When the gradient estimate $\widehat{\boldsymbol{g}}_k = {}^1/_{B_k} \sum_{i=1}^{B_k} \nabla f_{i_s}(\boldsymbol{w}_k)$ is created with batch size B_k in Eq. 3.3 with i_s chosen uniformly at random, then the expected variance*

$$\mathbb{E}\left[\|\nabla F(\boldsymbol{w}_k) - \boldsymbol{g}_k\|_2^2 \;\middle|\; \boldsymbol{w}_k\right] \leq \frac{F(\boldsymbol{w}_k) - F^\star}{c'}.$$

Proof.

$$\begin{aligned}
\mathbb{E}\left[\|\nabla F(\boldsymbol{w}_k) - \widehat{\boldsymbol{g}}_k\|_2^2 \;\middle|\; \boldsymbol{w}_k\right] &= \mathbb{E}\left[\|\nabla F(\boldsymbol{w}_k)\|_2^2 + \|\widehat{\boldsymbol{g}}_k\|_2^2 - 2\left\langle \nabla F(\boldsymbol{w}_k), \widehat{\boldsymbol{g}}_k \right\rangle \;\middle|\; \boldsymbol{w}_k\right] \\
&= \mathbb{E}\left[\left\| \frac{1}{B_k} \sum_{i=1}^{B_k} \nabla f_{i_k}(\boldsymbol{w}_k) \right\|_2^2 \;\middle|\; \boldsymbol{w}_k\right] - \|\nabla F(\boldsymbol{w}_k)\|_2^2 \\
&= \frac{\mathbb{E}\left[\|\nabla f(\boldsymbol{w}_k)\|_2^2 \;\middle|\; \boldsymbol{w}_k\right]}{B_k} + \frac{B_k - 1}{B_k} \|\nabla F(\boldsymbol{w}_k)\|_2^2 - \|\nabla F(\boldsymbol{w}_k)\|_2^2 \\
&\leq \frac{\mathbb{E}\left[M^2(\boldsymbol{w}_k) \mid \boldsymbol{w}_k\right]}{B_k} \\
&\leq \frac{\mathbb{E}\left[M^2(\boldsymbol{w}_k) \mid \boldsymbol{w}_k\right] (F(\boldsymbol{w}_k) - F^\star)}{c' M_U^2} \\
&\leq \frac{\mathbb{E}\left[F_k - F^\star\right]}{c'}
\end{aligned}$$

when the batch size $B_k = \lceil c(F(\boldsymbol{w}_k) - F^\star)^{-1} \rceil$ and with $c = c' M_U^2$.

□

B.2.1 Proof of Theorem 2

Proof. From definition of β-smooth (Definition 2) and with the mini-batch SGD iterations,

$$F(\boldsymbol{w}_{k+1}) \leq F(\boldsymbol{w}_k) - \gamma \left\langle \nabla F(\boldsymbol{w}_k), \frac{1}{B} \sum_{i=1}^{B} \nabla f_{s_i}(\boldsymbol{w}) \right\rangle + \frac{\beta \gamma^2}{2} \left\| \frac{1}{B} \sum_{i=1}^{B} \nabla f_{s_i}(\boldsymbol{w}_k) \right\|_2^2$$

Wrapping with conditional expectation and noticing that $\left\langle \sum_{i=1}^{B} \boldsymbol{a}_i, \sum_{i=1}^{B} \boldsymbol{a}_i \right\rangle = \sum_{i=1}^{B} \|\boldsymbol{a}_i\|^2 + \sum_{i=1}^{B} \sum_{j=1, j\neq i}^{B} \langle \boldsymbol{a}_i, \boldsymbol{a}_j \rangle$ and defining $\Omega_k := F(\boldsymbol{w}_k) - F(\boldsymbol{w}^\star)$, and $\overline{\Omega}_k := \Omega_k - \gamma \|\nabla F(\boldsymbol{w}_k)\|_2^2$ then

$$
\begin{aligned}
\mathbb{E}\left[\Omega_{k+1} \mid \boldsymbol{w}_k\right] &\leq \overline{\Omega}_k + \frac{\beta\gamma^2}{2}\mathbb{E}\left[\left\| \frac{1}{B}\sum_{i=1}^{B} \nabla f_{s_i}(\boldsymbol{w}_k) \right\|_2^2 \,\middle|\, \boldsymbol{w}_k\right] \\
&= \overline{\Omega}_k + \frac{\beta\gamma^2}{2}\mathbb{E}\left[\left\langle \frac{1}{B}\sum_{i=1}^{B} \nabla f_{i_s}(\boldsymbol{w}_k), \frac{1}{B}\sum_{i=1}^{B} \nabla f_{i_s}(\boldsymbol{w}_k) \right\rangle \,\middle|\, \boldsymbol{w}_k\right] \\
&= \overline{\Omega}_k + \frac{\beta\gamma^2}{2B^2}\mathbb{E}\left[\sum_{i=1}^{B} \|\nabla f_{i_s}(\boldsymbol{w}_k)\|_2^2 + \sum_{i\neq j}^{B} \langle \nabla f_{i_s}(\boldsymbol{w}_k), \nabla f_{j_s}(\boldsymbol{w}_k)\rangle \,\middle|\, \boldsymbol{w}_k\right] \\
&= \overline{\Omega}_k + \frac{\beta\gamma^2}{2}\left(\frac{\mathbb{E}\left[\|\nabla f(\boldsymbol{w}_k)\|_2^2\right]}{B} + \frac{B-1}{B}\|\nabla F(\boldsymbol{w}_k)\|_2^2\right) \\
&= \overline{\Omega}_k + \frac{\beta\gamma^2}{2}\left(\frac{M^2(\boldsymbol{w}_k)}{B} + \frac{B-1}{B}\|\nabla F(\boldsymbol{w}_k)\|_2^2\right) \\
&\leq \Omega_k + \|\nabla F(\boldsymbol{w}_k)\|_2^2\left(\frac{\beta\gamma^2}{2} - \gamma\right) + \frac{\beta\gamma^2}{2}\frac{F(\boldsymbol{w}_k) - F(\boldsymbol{w}^\star)}{c'}
\end{aligned}
$$

when $c = c' M_U^2$ by Lemma 4. Choose $\gamma < \frac{2}{\beta}$ so $\frac{\beta\gamma^2}{2} - \gamma < 0$. Then because F is α-PL,

$$
\begin{aligned}
&\leq F(\boldsymbol{w}_k) - F^\star - \left(\gamma - \frac{\beta\gamma^2}{2}\right) \cdot 2\alpha(F(\boldsymbol{w}_k) - F(\boldsymbol{w}^\star)) + \frac{\beta\gamma^2}{2}\frac{F(\boldsymbol{w}_k) - F(\boldsymbol{w}^\star)}{c'} \\
&= \left(1 - 2\alpha\gamma + 2\alpha\frac{\beta\gamma^2}{2} + \frac{\beta\gamma^2}{2c'}\right)(F(\boldsymbol{w}_k) - F(\boldsymbol{w}^\star)) \\
&= \left(1 - a\gamma + b\gamma^2\right)(F(\boldsymbol{w}_k) - F(\boldsymbol{w}^\star))
\end{aligned}
$$

when $a = 2\alpha$ and $b = \beta\left(\alpha + \frac{1}{2c'}\right)$. Choose the step size $\gamma = a/2b = \alpha/[\beta\left(\alpha + \frac{1}{2c'}\right)] < 1/\beta$. Then

$$= \left(1 - \frac{a^2}{4b}\right)(F(\boldsymbol{w}_k) - F(\boldsymbol{w}^\star))$$
$$= \left(1 - \frac{\alpha^2}{\beta\left(\alpha + \frac{1}{2c'}\right)}\right)(F(\boldsymbol{w}_k) - F(\boldsymbol{w}^\star))$$

This holds for any k. Then, by law of iterated expectation:

$$\begin{aligned}
\mathbb{E}\left[F(\boldsymbol{w}_2) - F^\star \mid \boldsymbol{w}_0\right] &= \mathbb{E}\left[\mathbb{E}\left[F(\boldsymbol{w}_2) - F^\star \mid \boldsymbol{w}_1\right] \mid \boldsymbol{w}_k\right] \\
&\leq \mathbb{E}\left[(1-r)\mathbb{E}\left[F(\boldsymbol{w}_1) - F^\star \mid \boldsymbol{w}_1\right] \mid \boldsymbol{w}_0\right] \\
&= (1-r)\mathbb{E}\left[F(\boldsymbol{w}_1) - F^\star \mid \boldsymbol{w}_0\right] \\
&\leq (1-r)\mathbb{E}\left[(1-r)(F(\boldsymbol{w}_0) - F^\star) \mid \boldsymbol{w}_0\right] \\
&= (1-r)^2(F(\boldsymbol{w}_0) - F^\star)
\end{aligned}$$

when $r := \left(1 - \frac{\alpha^2}{\beta\left(\alpha + \frac{1}{2c'}\right)}\right)$. Continuing this process to iteration T,

$$\mathbb{E}\left[F(\boldsymbol{w}_T) - F(\boldsymbol{w}^\star) \mid \boldsymbol{w}_0\right] \leq \left(1 - \frac{\alpha^2}{\beta\left(\alpha + \frac{1}{2c'}\right)}\right)^T (F(\boldsymbol{w}_0) - F(\boldsymbol{w}^\star))$$

Noticing that $1 - x \leq e^{-x}$ for all $x \geq 0$, $\mathbb{E}\left[F(\boldsymbol{w}_T) - F(\boldsymbol{w}^\star)\right] \leq \epsilon$ when

$$T \geq \log\left(\frac{F(\boldsymbol{w}_0) - F(\boldsymbol{w}^\star)}{\epsilon}\right)\left(\frac{\beta\left(\alpha + \frac{1}{2c'}\right)}{\alpha^2}\right) \tag{B.2}$$

□

B.2.2 Proof of Theorem 3

Proof. Suppose we use a step-size of $\gamma = 1/(\beta + 1/\eta)$ for $\eta > 0$. Then, we have the following relation, extracted from the proof of Theorem 6.3 of Bubeck et al. [24].

$$\mathbb{E}[F(\boldsymbol{w}_{k+1})-F^\star] \leq \frac{(\beta+1/\eta)}{2}\left(\mathbb{E}\|\boldsymbol{w}_k-\boldsymbol{w}^*\|-\mathbb{E}\|\boldsymbol{w}_{k+1}-\boldsymbol{w}^*\|\right)+\frac{\eta}{2}\mathbb{E}\|\nabla F(\boldsymbol{w}_k)-\widehat{\boldsymbol{g}}_k\|^2.$$

By Lemma 4, and taking $\eta = c'$, we have

$$\begin{aligned}\mathbb{E}[F(\boldsymbol{w}_{k+1}) - F^\star] &\leq \frac{(\beta+1/\eta)}{2}\left(\mathbb{E}\|\boldsymbol{w}_k-\boldsymbol{w}^*\|-\mathbb{E}\|\boldsymbol{w}_{k+1}-\boldsymbol{w}^*\|\right)+\frac{\eta}{2}\frac{\mathbb{E}[F(\boldsymbol{w}_k)-F^\star]}{c'}\\ &= \frac{(\beta+1/c')}{2}\left(\mathbb{E}\|\boldsymbol{w}_k-\boldsymbol{w}^*\|-\mathbb{E}\|\boldsymbol{w}_{k+1}-\boldsymbol{w}^*\|\right)+\frac{1}{2}\mathbb{E}[F(\boldsymbol{w}_k)-F^\star].\end{aligned}$$

Summing $k = 0$ to $k = T-1$ we have

$$\begin{aligned}\sum_{k=0}^{T-1}\mathbb{E}[F(\boldsymbol{w}_{k+1}) - F^\star] &\leq \frac{(\beta+1/c')}{2}\left(\mathbb{E}\|\boldsymbol{w}_0-\boldsymbol{w}^*\|-\mathbb{E}\|\boldsymbol{w}_{k+1}-\boldsymbol{w}^*\|\right)+\frac{1}{2}\sum_{k=0}^{T-1}\mathbb{E}[F(\boldsymbol{w}_k)-F^\star]\\ &\leq \frac{(\beta+1/c')}{2}R^2+\frac{1}{2}\sum_{k=0}^{T-1}\mathbb{E}[F(\boldsymbol{w}_k)-F^\star].\end{aligned}$$

Rearranging, we have

$$\begin{aligned}\sum_{k=0}^{T-1}\mathbb{E}[F(\boldsymbol{w}_{k+1}) - F^\star] &= (\beta+1/c')R^2+F(\boldsymbol{w}_0)-F^\star-2(F(\boldsymbol{w}_T)-F^\star)\\ &\leq (\beta+1/c')R^2+F(\boldsymbol{w}_0)-F^\star\end{aligned}$$

This implies the desired result after applying the law of iterated expectation and convexity. □

B.2.3 Proof of Theorem 5

Proof. By definition of β-smooth,

$$F(\boldsymbol{w}_{k+1}) \leq F(\boldsymbol{w}_k)+\langle\nabla F(\boldsymbol{w}_k),\boldsymbol{w}_{k+1}-\boldsymbol{w}_k\rangle+\frac{\beta}{2}\|\boldsymbol{w}_{k+1}-\boldsymbol{w}_k\|_2^2$$

Then substitution of $\boldsymbol{w}_{k+1} = \boldsymbol{w}_k - \frac{\gamma}{B_k}\sum_{i=1}^{B_k}\nabla f_{i_s}(\boldsymbol{w}_k)$, the following is obtained:

$$\gamma\left\langle \nabla F(\boldsymbol{w}_k), \frac{1}{B_k}\sum_{i=1}^{B_k}\nabla f_{i_s}(\boldsymbol{w}_k)\right\rangle \leq F_k - F_{k+1} + \frac{\beta\gamma^2}{2}\left\|\frac{1}{B_k}\sum_{i=1}^{B_k}\nabla f_{i_s}(\boldsymbol{w}_k)\right\|_2^2$$

Wrapping in conditional expectation given $\boldsymbol{w}_k$ and using the shorthand $\nabla f_i := \nabla f_i(\boldsymbol{w}) = \nabla f(\boldsymbol{w}; \boldsymbol{z}_i)$

$$\begin{aligned}
\gamma\left\|\nabla F(\boldsymbol{w}_k)\right\|_2^2 &\leq \mathbb{E}\left[F_k - F_{k+1} \mid \boldsymbol{w}_k\right] + \frac{\beta\gamma^2}{2}\mathbb{E}\left[\left\|\frac{1}{B_k}\sum_{i=1}^{B_k}\nabla f_{i_s}(\boldsymbol{w}_k)\right\|_2^2 \,\middle|\, \boldsymbol{w}_k\right] \\
&\leq \mathbb{E}\left[F_k - F_{k+1} \mid \boldsymbol{w}_k\right] + \frac{\beta\gamma^2}{2B_k^2}\mathbb{E}\left[\sum_{i=1}^{B_k}\sum_{j=1}^{B_k}\langle\nabla f_{i_s}, \nabla f_{j_s}\rangle \,\middle|\, \boldsymbol{w}_k\right] \\
&\leq \mathbb{E}\left[F_k - F_{k+1} \mid \boldsymbol{w}_k\right] + \frac{\beta\gamma^2}{2B_k^2}\mathbb{E}\left[\sum_{i=1}^{B_k}\left\|\nabla f_{i_s}\right\|_2^2 + \sum_{i=1}^{B_k}\sum_{j=1}^{B_k}\langle\nabla f_{i_s}, \nabla f_{j_s}\rangle \,\middle|\, \boldsymbol{w}_k\right] \\
&\leq \mathbb{E}\left[F_k - F_{k+1}\right] + \frac{\beta\gamma^2}{2}\left(\frac{M^2(\boldsymbol{w}_k)}{B_k} + \frac{B_k - 1}{B_k}\left\|\nabla F(\boldsymbol{w}_k)\right\|_2^2\right)
\end{aligned}$$

because the indices i_s and j_s are chosen independently and $\mathbb{E}\left[\nabla f_{i_s}(\boldsymbol{w})\right] = F(\boldsymbol{w})$. Then, substituting the definition of B_k in Eq. 3.4,

$$\leq \mathbb{E}\left[F_k - F_{k+1}\right] + \frac{\beta\gamma^2}{2}\left(\frac{\left\|\nabla F(\boldsymbol{w}_k)\right\|_2^2}{c} + \left\|\nabla F(\boldsymbol{w}_k)\right\|_2^2\right)$$

when $c = c'M_L^2$. Then this inequality is obtained after rearranging:

$$\left\|\nabla F(\boldsymbol{w}_k)\right\|_2^2\left(\gamma - \frac{\gamma^2\beta}{2}(c'^{-1} + 1)\right) \leq \mathbb{E}\left[F_k - F_{k+1}\right]$$

Then with this result and iterated expectation

$$\begin{aligned}\min_{k=0,\ldots,T-1} \|\nabla F(\boldsymbol{w}_k)\|_2^2 &\leq \frac{1}{T}\sum_{k=0}^{T-1} \|\nabla F(\boldsymbol{w}_k)\|_2^2 \\ &\leq \frac{F_0 - F^\star}{T}\left(\gamma - \frac{\gamma^2\beta}{2}(c'^{-1}+1)\right)^{-1} \\ &\leq \frac{F_0 - F^\star}{T} 2\beta\left(\frac{1}{c'}+1\right)\end{aligned}$$

when $\gamma = \beta^{-1}c/(c + M_L^2))$.

□

Corollary 17.

$$\sum_{k=0}^{T-1} \|\nabla F(\boldsymbol{w}_k)\|_2^2 \leq 2\beta\left(F_0 - F^\star\right)\left(\frac{1}{c'}+1\right)$$

B.3 Proofs for required number of examples

The number of examples required to be processed is the sum of batch sizes, $\sum_{i=1}^{T} B_i$ over T iterations. This section will assume an oracle provides the batch size B_i to provide bounds on the number of examples to provide bounds on $\sum_{i=1}^{T} B_i$.

B.3.1 Proof of Corollaries 6 and 7

These proofs require another lemma that will be used in both proofs:

Lemma 18. *If a model is trained so the loss difference from optimal $F(\boldsymbol{w}) - F^\star \in [\epsilon/2, \epsilon]$, then $4B_0(F(\boldsymbol{w}_0) - F^\star)T/\epsilon$ examples need to be processed when there are T model updates the initial batch size is B_0.*

Proof of Corollary 6

Proof. This case requires $T \geq c_{\alpha,\beta} \log(\delta_0/\epsilon)$ iterations for some constant c when F is α-PL and β-smooth by Equation B.2 when $\delta_0 = F(\boldsymbol{w}_0) - F^\star$. Applying Lemma 18 gives that the adaptive batch size scheme in Eq. (3.3) requires no more then the number of examples

$$\sum_{k=0}^{T-1} B_k \leq \frac{\log(\delta_0/\epsilon)}{\epsilon} \cdot 4c_{\alpha,\beta} B_0 \delta_0$$

□

Proof of Corollary 7

Proof. This case requires $T \geq r_\beta/\epsilon$ iterations when F is convex and β-smooth by Theorem 2. Applying Lemma 18 gives that the adaptive batch size scheme in Eq. (3.3) requires no more then the number of examples

$$\sum_{k=0}^{T-1} B_k \leq \frac{1}{\epsilon^2} \cdot 4r_\beta B_0 \delta_0$$

when $\delta_0 := F(\boldsymbol{w}_0) - F^\star$. □

B.3.2 Proof of Lemma 18

Proof.

$$\begin{aligned}
\sum_{k=1}^{T} B_k &= \sum_{k=1}^{T} \left\lceil \frac{B_0(F(\boldsymbol{w}_0) - F^\star)}{F(\boldsymbol{w}_k) - F^\star} \right\rceil \\
&\leq 2B_0(F(\boldsymbol{w}_0) - F^\star) \sum_{k=1}^{T} \frac{1}{F(\boldsymbol{w}_k) - F^\star} \\
&\leq 4B_0(F(\boldsymbol{w}_0) - F^\star)T/\epsilon
\end{aligned}$$

□

B.3.3 Proof of Corollary 8

Proof. Following the proof of Lemma 18,

$$\begin{aligned}\sum_{k=1}^{T} B_k &= \sum_{k=1}^{T} \left\lceil \frac{c}{\|\nabla F(\boldsymbol{w}_k)\|_2^2} \right\rceil \\ &\leq 2c \sum_{k=1}^{T} \frac{1}{\|\nabla F(\boldsymbol{w}_k)\|_2^2} \\ &\leq 4cT/\epsilon \\ &\leq 4cr/\epsilon^3\end{aligned}$$

using Theorem 5 when $\|\nabla F(\boldsymbol{w}_k)\| \leq \epsilon$ (and not when $\|\nabla F(\boldsymbol{w}_k)\|_2^2 \leq \epsilon$). □

B.4 Experiment details

PyTorch [121] is used to implement all optimization.

B.4.1 Simulation with synthetic dataset

All optimizers use learning rate $\gamma = 2.5 \cdot 10^{-3}$ unless explicitly noted otherwise.

- **SGD with adaptive batch sizes.** Batch size: $B_k = \lceil B_0(F(\boldsymbol{w}_0) - F^\star)(F(\boldsymbol{w}_k) - F^\star)^{-1} \rceil$, $B_0 = 2$.
- **SGD with decaying step sizes**: Static batch size $B = 64$, decaying step size $\gamma_k = 10\gamma/k$ at iteration k [111].
- **AdaGrad** is used with a batch size of $B = 64$ and PyTorch 1.1's default hyperparameters, $\gamma = 0.01$ and 0 for all other hyperparameters.
- **Gradient descent**. No other hyperparameters are required past learning rate.

These hyperparameters were not tuned past ensuring the convergence of each optimizer.

B.4.2 Experiment with Fashion MNIST

Fashion MNIST is a dataset with 60,000 training examples and 10,000 testing examples. Each example includes a 28×28 image that falls in one of 10 classes (e.g., "coat" or "bag") [174]. The standard pre-processing in PyTorch's MNIST example is used.[1] The CNN in the example will used, which has about 111,000 parameters that specify 3 convolutional layers with max-pooling and 2 fully-connected layers, with ReLU activations after every layer.

The hyperparameter optimization process followed the data flow below for each optimizer:

- Randomly sample hyperparameters, and train the model for 200 epochs on 80% of the training set (using the remaining 20% for validation).
- Refine hyperparameters based on the hyperparameters that had validation loss within 0.005 of the minimum, and had fewer model updates than the mean number of model updates.
- Repeat steps 1 and 2 until satisfied with validation performance.
- Manually choose one set of hyperparameters for each optimizer, and train for 200 epochs with the entire training set, and report performance on the test set.

Step (4) has only been run once for RadaDamp. For GeoDamp, we sampled at least 268 hyperparameters, and for AdaGrad we sampled at least

[1] The transform at http://github.com/pytorch/examples/.../mnist/main.py#L105 is used; the resulting pixels value have a mean of 0.504 and a standard deviation of 1.14, not zero mean and unit variance as is typical for preprocessing. The model used has about 110 thousand parameters and includes biases in all layers, likely resolving any issues.

179 hyperparameters. We spent a while on step (3) for RadaDamp[2] Both GeoDamp and AdaGrad required fewer iterations of step (3).

Hyperparameter sampling space, and tuned values are below. After some initial sampling, the learning rate is fixed at to be 0.005 and initial/maximum batch sizes to be 256/1024 respectively for all optimizers. We tuned the value of weight decay more for RadaDamp, and set it to be 0.003 for all optimizers.

With those fixed hyperparameters, in our last run of hyperparameter optimization we sampled from these hyperparameters:

- AdaGrad:
  - Batch size: [16, 32, 64, 128, 256] **(tuned value: 256)**
- GeoDamp:
  - Damping delay (epochs): (2, 5, 10, 20, 30, 60] **(tuned value: 10)**
  - Damping factor: log-uniform between 1 to 10 **(tuned value: 1.219231)**
- RadaDamp:
  - "Dwell": [1, 10, 20, 30, 50, 100] **(tuned value: 1)**
  - Memory ρ: [0.95, 0.99, 0.995, 0.999] **(tuned value: 0.999)**

"Dwell" is the frequency at which to update the batch size; if dwell= 7, then the batch size will be updated every 7 model updates. Because we found the best value of dwell to be 1, it is not included in the description of Algorithm 1.

GeoDamp and RadaDamp change the batch size/learning rate for SGD with Nesterov momentum (and a momentum value 0.9). GeoDamp-LR aka SGD and RadaDamp-LR change the learning rate by the same amount

[2]Primarily to tune the regularization balance between loss and gradient norm, λ. We didn't have much success with large λ.

the batch size would have changed; if RadaDamp increases the batch size by a factor of d, RadaDamp-LR will decay the learning rate by a factor of d instead. When the maximum batch size is reached for RadaDamp and GeoDamp, the learning rate is decayed instead of the batch size increasing by the same scheme.

If the damping factor is d and the damping delay is e epochs, the batch size increases by a factor of d or the step size decays by a factor of d every e epochs.

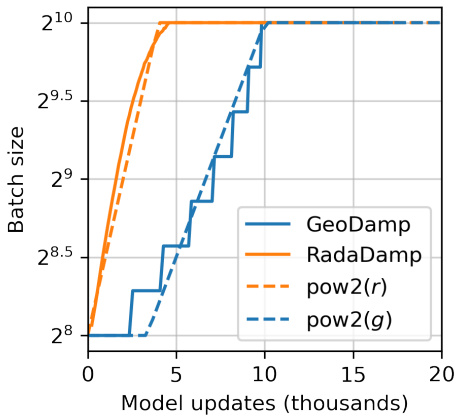


Figure B.1: The batch size against the number of model updates u. Here, $r = 0.5u/10^3 + 8$ and $g = 0.3u/10^3 + 7$.

C TRAINING PYTORCH MODELS FASTER WITH DASK

C.1 Example usage

First, let's create a standard PyTorch model. This is a simple definition; a more complicated model or one that uses GPUs can easily be substituted.

```
import torch.nn as nn
import torch.nn.functional as F

class HiddenLayer(nn.Module):
    def __init__(self, features=4, hidden=2, out=1):
        super().__init__()
        self.hidden = nn.Linear(features, hidden)
        self.out = nn.Linear(hidden, out)

    def forward(self, x, *args, **kwargs):
        return self.out(F.relu(self.hidden(x)))
```

Now, let's create our optimizer:

```
from adadamp import DaskRegressor
import torch.optim as optim

est = DaskRegressor(
    module=HiddenLayer, module__features=10,
    optimizer=optim.Adadelta,
    optimizer__weight_decay=1e-7,
    max_epochs=10
)
```

So far, a PyTorch model and optimizer have been specified. As per the Scikit-learn API, we specify parameters for the model/optimizer with double underscores, so in our example "HiddenLayer(features=10)" will be created. We can set the batch size increase parameters at initialization if desired, or inside `set_params`.

```
from adadamp.dampers import GeoDamp
est.set_params(
    batch_size=GeoDamp, batch_size__delay=60,
    batch_size__factor=5)
```

This will increase the batch size by a factor of 5 every 60 epochs, which is used in the experiments. Now, we can train:

```
from sklearn.datasets import make_regression
X, y = make_regression(n_features=10)
X = torch.from_numpy(X.astype("float32"))
y = torch.from_numpy(y.astype("float32")).reshape(-1, 1)

est.fit(X, y)
```

C.2 Future work

C.2.1 Architecture

Fundamentally, the model weights can be either be held on a master node (centralized), or on every node (decentralized). Respectively, these storage architectures typically use point-to-point communication or an "all-reduce" communication. Both centralized [2, 96] and decentralized [97, 146] communication architectures are common.

Future work is to avoid the overhead introduced by manually having Dask control the model update workflow. With any synchronous centralized

system, the time required for any one model update is composed of the time required for the following tasks:

1. Broadcasting the model from the master node to all workers
2. Finishing gradient computation on all workers.
3. Communicating gradients back to master node.
4. Various overhead tasks (e.g., serialization, worker scheduling, etc).
5. Computing the model update after all gradients are computed & gathered.

Items (1), (3) and (4) are a large concern in our implementation. Decentralized communication has the advantage of eliminating item (1) and most of item (4), and mitigates (3) with a smarter communication strategy (all-reduce vs. point-to-point). Item (2) is still a concern with straggler nodes [46], but recent work has achieved "near-linear scalability with 256 GPUs" in a homogeneous computing environment [97]. Items (2) and (5) can be avoided with asynchronous methods (e.g., [118, 185]).

That is, most of the concerns in our implementation will be resolved with a distributed communication strategy. The PyTorch distributed communication package uses a synchronous decentralized strategy, so the model is communicated to each worker and gradients are sent between workers with an all-reduce scheme [97]. It has some machine learning specific features to reduce the communication time, including performing both computation and communication concurrently as layer gradients become available [97, Sec. 3.2.3].

The software library dask-pytorch-ddp[1] allows use of the PyTorch decentralized communications [97] with Dask clusters, and is a thin wrapper around PyTorch's distributed communication package. Future work will likely involve ensuring training can efficiently use a variable number of workers.

[1] https://github.com/saturncloud/dask-pytorch-ddp

C.2.2 Simulations

We have simulated the expected gain from the work of enabling decentralized communication with two networks that use a decentralized all-reduce strategy:

- `decentralized-medium` It assumes an a network with inter-worker bandwidth of 54Gb/s and a latency of 0.05μs.
- `centralized` uses a centralized communication strategy (as implemented) and the same network as `decentralized-medium`.
- `decentralized-high` has the same network as `decentralized-medium` but has an inter-worker bandwidth of 800Gb/s and a latency of 0.025μs.

To provide baseline performance, we also show the results with the current implementation:

- `centralized` uses the same network as `decentralized-medium` but with the centralized communication scheme that is currently implemented.

Table C.1: Simulations that indicate how the training time (in minutes) will change under different architectures and networks. The "centralized" architecture is the currently implemented architecture, and has the same numbers as "training time" in Table 3.7.

Maximum batch size	Centralized	Decentralized (moderate)	Decentralized (high)
5.1k (*2)	69.9	45.1	43.5
3.2k	107.2	67.7	65.5
16k	107.5	67.7	65.7
640	116.9	73.6	71.8
128	200.2	121.7	121.5

`decentralized-medium` is most applicable for clusters that have decent bandwidth between nodes. It's also applicable to for certain cases when Amazon EC2 is used with one GPU per worker,[2] or workers have a very moderate Infiniband setup.[3] `decentralized-high` is a simulation of the network used by the PyTorch developers to illustrate their distributed communication [97]. We have run simulations to illustrate the effects of these networks. Of course, changing the underlying networks does not affect the number of epochs or model updates, so Figs. 3.6 and 3.7 also apply here.

A summary of how different networks affect training time is shown in Table C.1. We show the training time for a particular network (`decentralized-moderate`) in Figure C.1; `decentralized-high` shows similar performance as illustrated in Table C.1. A visualization of Table C.1 is shown in Figure 3.8. This shows how network quality affects the performance of different optimization methods in Figure C.1. Clearly, the optimization method (and the maximum number of workers) is more important than the network.

Finally, let's show how the number of Dask workers affects the time required to complete a single epoch with a constant batch size. This simulation will use the `decentralized-high` network and has the advantage of removing any overhead. The results in Figure C.2 show that the speedups start saturating around 128 examples/worker for the model used with a batch size of 512. Larger batch sizes will likely mirror this performance – computation is bottleneck with this model/dataset/hardware.

[2] 50Gb/s and 25Gb/s networks can be obtained with `g4dn.8xlarge` and `g4dn.xlarge` instances respectively. `g4dn.xlarge` machines have 1 GPU each and are the least expensive for a fixed number of FLOPs on the GPU.

[3] A 2011 Infiniband setup with 4 links (https://en.wikipedia.org/wiki/InfiniBand#Performance)

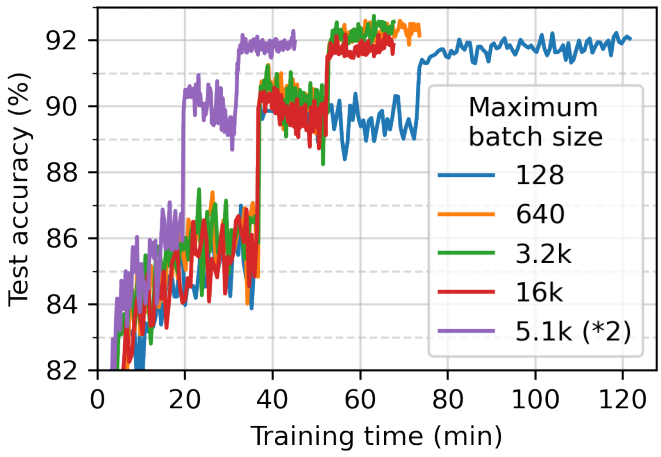


Figure C.1: The training time required for different optimizers under the `decentralized-moderate` network.

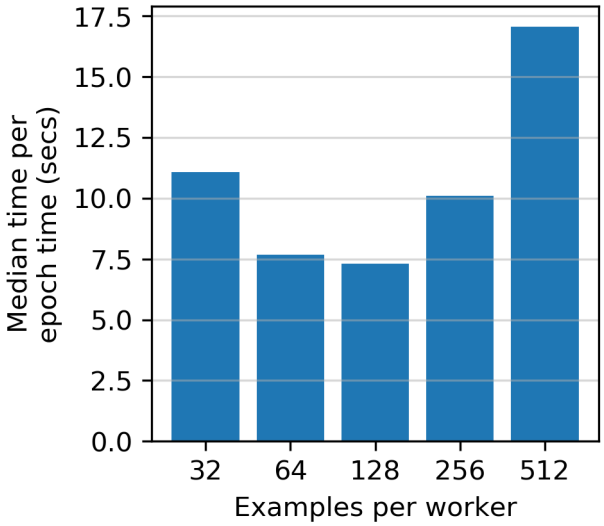


Figure C.2: The median time to complete a pass through the training set with a batch size of 512. As expected, the speedups diminish when there is little computation and much communication (say with 32 examples per worker).

C.3 Loss vs. time

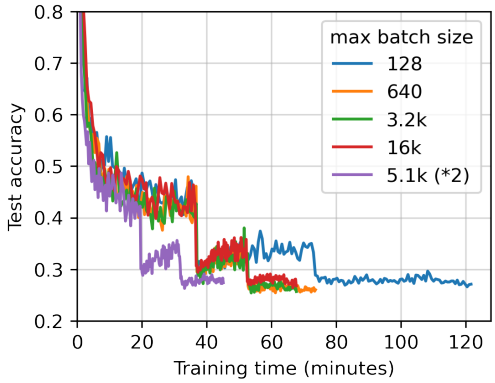


Figure C.3: The training time required for different optimizers under the `decentralized-moderate` network.

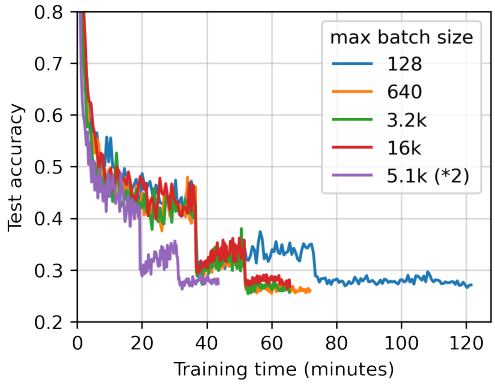


Figure C.4: The training time required for different optimizers under the `decentralized-high` network.

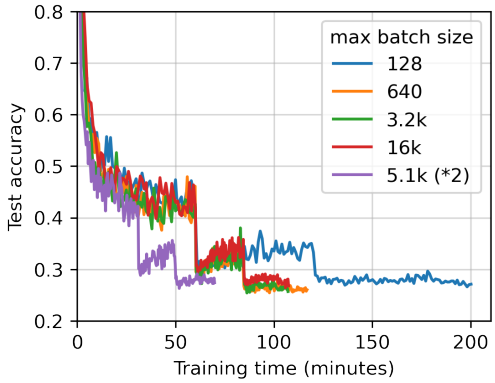


Figure C.5: The training time required for different optimizers under the `centralized` network.

D.1 Rigorous statement

Let $(V, \langle \cdot, \cdot \rangle)$ be an inner product space over $\mathbb{R}$ and let $\| \cdot \|$ denote the induced norm on V. In what follows, you may think of $\boldsymbol{g}$ as a stochastic gradient of the function we wish to optimize. An *atomic decomposition* of $\boldsymbol{g}$ is any decomposition of the form $\boldsymbol{g} = \sum_{\boldsymbol{a} \in \mathcal{A}} \lambda_{\boldsymbol{a}} \boldsymbol{a}$ for some set of atoms $\mathcal{A} \subseteq V$. Intuitively, $\mathcal{A}$ consists of simple building blocks. We will assume that for all $\boldsymbol{a} \in \mathcal{A}$, $\|\boldsymbol{a}\| = 1$, as this can be achieved by a positive rescaling of the $\lambda_{\boldsymbol{a}}$.

An example of an atomic decomposition is the entry-wise decomposition $\boldsymbol{g} = \sum_i \boldsymbol{g}_i e_i$ where $\{\boldsymbol{e}_i\}_{i=1}^n$ is the standard basis. More generally, any orthonormal basis of V gives rise to a unique atomic decomposition of $\boldsymbol{g}$. While we focus on finite dimensional vectors, one could use Fourier and wavelet decompositions in this framework. When considering matrices, the singular value decomposition gives an atomic decomposition in the set of rank-1 matrices. More general atomic decompositions have found uses in a variety of situations, including solving linear inverse problems [30].

We are interested in finding an approximation to g with fewer atoms. Our primary motivation is that this reduces communication costs, as we only need to send atoms with non-zero weights. We can use whichever decomposition is most amenable for sparsification. For instance, if X is a low rank matrix, then its singular value decomposition is naturally sparse, so we can save communication costs by sparsifying its singular value decomposition instead of its entries.

Suppose $\mathcal{A} = \{\boldsymbol{a}_i\}_{i=1}^n$ and we have an atomic decomposition $g = \sum_{i=1}^n \lambda_i \boldsymbol{a}_i$. We wish to find an unbiased estimator $\widehat{\boldsymbol{g}}$ of $\boldsymbol{g}$ that is sparse in these atoms, and with small variance. Since $\widehat{\boldsymbol{g}}$ is unbiased, minimizing its variance is equivalent to minimizing $\mathbb{E}[\|\widehat{\boldsymbol{g}}\|^2]$. We use Eq. 3.7 for our

estimator. We have the following lemma about $\widehat{\boldsymbol{g}}$ in Eq. 3.7:

Lemma 19. $\mathbb{E}[\widehat{\boldsymbol{g}}] = g$ *and* $\mathbb{E}[\|\widehat{\boldsymbol{g}}\|^2] = \sum_{i=1}^n \lambda_i^2 p_i^{-1} + \sum_{i \neq j} \lambda_i \lambda_j \langle \boldsymbol{a}_i, \boldsymbol{a}_j \rangle$.

Let $\boldsymbol{\lambda} = [\lambda_1, \ldots, \lambda_n]^T$, $\boldsymbol{p} = [p_1, \ldots, p_n]^T$. In order to ensure that this estimator is sparse, we fix some *sparsity budget* s. That is, we require $\sum_i p_i = s$. This is a *sparsity on average* constraint. We wish to minimize $\mathbb{E}[\|\widehat{\boldsymbol{g}}\|^2]$ subject to this constraint. By Lemma 19, this is equivalent to Eq. (3.8), rephrased below:

$$\min_{\boldsymbol{p}} \sum_{i=1}^n \frac{\lambda_i^2}{p_i} \quad \text{s.t.} \quad \forall i,\ 0 < p_i \leq 1, \quad \sum_{i=1}^n p_i = s. \tag{D.1}$$

An equivalent form of this problem was presented in [89] (Section 6.1). The authors considered this problem for entry-wise sparsification and found a closed-form solution for $s \leq \|\boldsymbol{\lambda}\|_1 / \|\boldsymbol{\lambda}\|_\infty$. We give a version of their result but extend it to all s. A similar optimization problem was given in [168], which instead minimizes sparsity subject to a variance constraint.

Algorithm 2 ATOMO probabilities

procedure ATOMO($\boldsymbol{\lambda} \in \mathbb{R}^n$ with $|\lambda_1| \geq \ldots |\lambda_n|$; sparsity budget s such that $0 < s \leq n$.)
  $i = 1$
  while $i \leq n$ **do**
  if $|\lambda_i| s \leq \sum_{j=i}^n |\lambda_i|$ **then**
    for $k = i, \ldots, n$ **do**
      $p_k = |\lambda_k| s \left(\sum_{j=i}^n |\lambda_i| \right)^{-1}$
    $i = n + 1$
  else
    $p_i = 1, s = s - 1$
    $i = i + 1$
return $\boldsymbol{p} \in \mathbb{R}^n$ solving Eq. (3.8)

We will show that Algorithm 2 produces $\boldsymbol{p} \in \mathbb{R}^n$ solving Eq. (3.8). While we show in Appendix D.3 that this can be derived via the KKT conditions,

we focus on an alternative method relaxes Eq. (3.8) to better understand its structure. This approach also analyzes the variance achieved by solving Eq. (3.8) more directly.

Note that Eq. (3.8) has non-empty feasible set only for $0 < s \leq n$. Define $f(\boldsymbol{p}) := \sum_{i=1}^{n} \lambda_i^2/p_i$. To understand how to solve Eq. (3.8), we first consider the following relaxation:

$$\min_{p} \sum_{i=1}^{n} \frac{\lambda_i^2}{p_i} \quad \text{s.t.} \quad \forall i,\ 0 < p_i, \quad \sum_{i=1}^{n} p_i = s. \tag{D.2}$$

We have the following lemma about the solutions to Eq. (D.2), first shown in [89].

Lemma 20 ([89])**.** *Any feasible vector p to Eq. (D.2) satisfies $f(\boldsymbol{p}) \geq \frac{1}{s}\|\boldsymbol{\lambda}\|_1^2$. This is achieved iff $p_i = \frac{|\lambda_i| s}{\|\boldsymbol{\lambda}\|_1}$.*

Lemma 20 implies that if we ignore the constraint that $p_i \leq 1$, then the optimal p is achieved by setting $p_i = |\lambda_i| s/\|\lambda\|_1$. If the quantity in the right-hand side is greater than 1, this does not give us an actual probability. This leads to the following definition.

Definition 21. *An atomic decomposition $\boldsymbol{g} = \sum_{i=1}^{n} \lambda_i \boldsymbol{a}_i$ is s-unbalanced at entry i if $|\lambda_i| s > \|\lambda\|_1$.*

We say that g is s-balanced otherwise. Clearly, an atomic decomposition is s-balanced iff $s \leq \|\boldsymbol{\lambda}\|_1/\|\boldsymbol{\lambda}\|_\infty$. Lemma 20 gives us the optimal way to sparsify s-balanced vectors, since the optimal p for Eq. (D.2) is feasible for Eq. (3.8). If $\boldsymbol{g}$ is s-unbalanced at entry j, we cannot assign this p_j as it is larger than 1. In the following lemma, we show that in $p_j = 1$ is optimal in this setting.

Lemma 22. *Suppose that $\boldsymbol{g}$ is s-unbalanced at entry j and that $\boldsymbol{q}$ is feasible in Eq. (3.8). Then $\exists\ \boldsymbol{p}$ that is feasible such that $f(\boldsymbol{p}) \leq f(\boldsymbol{q})$ and $p_j = 1$.*

Let $\phi(\boldsymbol{g}) = \sum_{i \neq j} \lambda_i \lambda_j \langle \boldsymbol{a}_i, \boldsymbol{a}_j \rangle$. Lemmas 20 and 22 imply the following theorem about solutions to Eq. (3.8).

Theorem 23. *If g is s-balanced, then $\mathbb{E}[\|\widehat{\boldsymbol{g}}\|^2] \geq s^{-1}\|\boldsymbol{\lambda}\|_1^2 + \phi(\boldsymbol{g})$ with equality if and only if $p_i = |\lambda_i| s / \|\boldsymbol{\lambda}\|_1$. If $\boldsymbol{g}$ is s-unbalanced, then $\mathbb{E}[\|\widehat{\boldsymbol{g}}\|^2] > s^{-1}\|\boldsymbol{\lambda}\|_1^2 + \phi(\boldsymbol{g})$ and is minimized by $\boldsymbol{p}$ with $p_j = 1$ where $j = \arg\max_{i=1,\ldots,n} |\lambda_i|$.*

Due to the sorting requirement in the input, Algorithm 2 requires $O(n \log n)$ operations. In Appendix D.3 we describe a variant that uses only $O(sn)$ operations. Thus, we can solve Eq. (3.8) in $O(\min\{n, s\} \log(n))$ operations.

D.2 Proofs

D.2.1 Proof of Lemma 20

Proof. Suppose we have some p satisfying the conditions in Eq. (D.2). We define two auxiliary vectors $\boldsymbol{\alpha}, \boldsymbol{\beta} \in \mathbb{R}^n$ by

$$\alpha_i = \frac{|\lambda_i|}{\sqrt{p_i}}.$$
$$\beta_i = \sqrt{p_i}.$$

Then note that using the fact that $\sum_i p_i = s$, we have

$$f(\boldsymbol{p}) = \sum_{i=1}^n \frac{\lambda_i^2}{p_i} = \left(\sum_{i=1}^n \frac{\lambda_i^2}{p_i}\right)\left(\frac{1}{s}\sum_{i=1}^n p_i\right) = \frac{1}{s}\left(\sum_{i=1}^n \frac{\lambda_i^2}{p_i}\right)\left(\sum_{i=1}^n p_i\right) = \frac{1}{s}\|\boldsymbol{\alpha}\|_2^2\| \\ bmbeta\|_2^2.$$

By the Cauchy-Schwarz inequality, this implies

$$f(\boldsymbol{p}) = \frac{1}{s}\|\boldsymbol{\alpha}\|_2^2\|\boldsymbol{\beta}\|_2^2 \geq \frac{1}{s}|\langle \boldsymbol{\alpha}, \boldsymbol{\beta} \rangle|^2 = \frac{1}{s}\left(\sum_{i=1}^n |\lambda_i|\right)^2 = \frac{1}{s}\|\boldsymbol{\lambda}\|_1^2. \tag{D.3}$$

This proves the first part of Lemma 20. In order to have $f(\boldsymbol{p}) = \frac{1}{s}\|\boldsymbol{\lambda}\|_1^2$, Eq. (D.3) implies that we need

$$|\langle \boldsymbol{\alpha}, \boldsymbol{\beta} \rangle| = \|\boldsymbol{\alpha}\|_2 \|\boldsymbol{\beta}\|_2.$$

By the Cauchy-Schwarz inequality, this occurs iff $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ are linearly dependent. Therefore, $c\boldsymbol{\alpha} = \boldsymbol{\beta}$ for some constant c. Solving, this implies $p_i = c|\lambda_i|$. Since $\sum_{i=1}^n p_i = s$, we have

$$c\|\lambda\|_1 = \sum_{i=1}^n c|\lambda_i| = \sum_{i=1}^n p_i = s.$$

Therefore, $c = \|\boldsymbol{\lambda}\|_1/s$, which implies the second part of the theorem. □

D.2.2 Proof of Lemma 22

Fix $\boldsymbol{q}$ that is feasible in Eq. (3.8). To prove Lemma 22 we will require a lemma. Given the atomic decomposition $\boldsymbol{g} = \sum_{i=1}^n \lambda_i \boldsymbol{a}_i$, we say that λ is s-unbalanced at i if $|\lambda_i|s > \|\boldsymbol{\lambda}\|_1$, which is equivalent to $\boldsymbol{g}$ being unbalanced in this atomic decomposition at i. For notational simplicity, we will assume that $\boldsymbol{\lambda}$ is s-unbalanced at $i = 1$. Let $A \subseteq \{2, \ldots, n\}$. We define the following notation:

$$s_A = \sum_{i \in A} q_i.$$

$$f_A(\boldsymbol{q}) = \sum_{i \in A} \frac{\lambda_i^2}{q_i}.$$

$$(\boldsymbol{\lambda}_A)_i = \begin{cases} \lambda_i, & \text{for } i \in A, \\ 0, & \text{for } i \notin A. \end{cases}$$

Note that under this notation, Lemma 20 implies that for all $p > 0$,

$$f_A(\boldsymbol{p}) \geq \frac{1}{s_A}\|\boldsymbol{\lambda}_A\|_1^2. \tag{D.4}$$

Lemma 24. *Suppose that q is feasible and that there is some set $A \subseteq \{2, \ldots, n\}$ such that*

1. *$\boldsymbol{\lambda}_A$ is $(s_A + q_1 - 1)$-balanced.*
2. *$|\boldsymbol{\lambda}_1|(s_A + q_1 - 1) > \|\boldsymbol{\lambda}_A\|_1$.*

Then there is a vector p that is feasible satisfying $f(\boldsymbol{p}) \leq f(\boldsymbol{q})$ and $p_1 = 1$.

Proof. Suppose that such a set A exists. Let $B = \{2, \ldots, n\} \backslash A$. Note that we have

$$f(\boldsymbol{q}) = \sum_{i=1}^{n} \frac{\lambda_i^2}{q_i} = \frac{\lambda_1^2}{q_1} + f_A(\boldsymbol{q}) + f_B(\boldsymbol{q}).$$

By Eq. (D.4), this implies

$$f(\boldsymbol{q}) \geq \frac{\lambda_1^2}{q_1} + \frac{1}{s_A}\|\boldsymbol{\lambda}_A\|_1^2 + f_B(\boldsymbol{q}). \tag{D.5}$$

Define $\boldsymbol{p}$ as follows.

$$p_i = \begin{cases} 1, & \text{for } i = 1, \\ \dfrac{|\lambda_i|(s_A + q_1 - 1)}{\|\boldsymbol{\lambda}_A\|_1}, & \text{for } i \in A, \\ q_i, & \text{for } i \notin A. \end{cases}$$

Note that by Assumption 1 and Lemma 20, we have

$$f_A(\boldsymbol{p}) = \frac{1}{s_A + q_1 - 1}\|\boldsymbol{\lambda}_A\|_1^2.$$

Since $p_i = q_i$ for $i \in B$, we have $f_B(\boldsymbol{p}) = f_B(\boldsymbol{q})$. Therefore,

$$f(\boldsymbol{p}) = \lambda_1^2 + \frac{1}{s_A + q_1 - 1}\|\boldsymbol{\lambda}_A\|_1^2 + f_B(\boldsymbol{q}). \tag{D.6}$$

Combining Eq. (D.5) and Eq. (D.6), we have

$$\begin{aligned} f(\boldsymbol{q}) - f(\boldsymbol{p}) &= \lambda_1^2\left(\frac{1}{q_1} - 1\right) + \|\boldsymbol{\lambda}_A\|_1^2\left(\frac{1}{s_A} - \frac{1}{s_A + q_1 - 1}\right) \\ &= \lambda_1^2\left(\frac{1 - q_1}{q_1}\right) + \|\boldsymbol{\lambda}_A\|_1^2\left(\frac{q_1 - 1}{s_A(s_A + q_1 - 1)}\right). \end{aligned}$$

Combining this with Assumption 2, we have

$$f(\boldsymbol{q}) - f(\boldsymbol{p}) \geq \frac{\|\boldsymbol{\lambda}_A\|_1^2}{(s_A + q_1 - 1)^2}\left(\frac{1 - q_1}{q_1}\right) + \|\boldsymbol{\lambda}_A\|_1^2\left(\frac{q_1 - 1}{s_A(r_A + q_1 - 1)}\right). \tag{D.7}$$

To show that the RHS of Eq. (D.7) is at most 0, it suffices to show

$$s_A \geq q_1(s_A + q_1 - 1). \tag{D.8}$$

However, note that since $0 < q_1 < 1$, the RHS of Eq. (D.8) satisfies

$$q_1(s_A + q_1 - 1) = s_A q_1 - q_1(1 - q_1) \leq s_A q_1 \leq s_A.$$

Therefore, Eq. (D.8) holds, completing the proof. □

We can now prove Lemma 22. In the following, we will refer to Conditions 1 and 2, relative to some set A, as the conditions required by Lemma 24.

Proof. We first show this in the case that $n = 2$. Here we have the atomic decomposition

$$\boldsymbol{g} = \lambda_1 \boldsymbol{a}_1 + \lambda_2 \boldsymbol{a}_2.$$

The condition that $\boldsymbol{\lambda}$ is s-unbalanced at $i = 1$ implies

$$|\lambda_1|(s-1) > |\lambda_2|.$$

In particular, this implies $s > 1$. For $A = \{2\}$, Condition 1 is equivalent to

$$|\lambda_2|(s_A + q_1 - 2) \leq 0.$$

Note that $s_A = q_2$ and that $q_1 + q_2 - 2 = s - 2$ by assumption. Since $q_i \leq 1$, we know that $s - 2 \leq 0$ and so Condition 1 holds. Similarly, Condition 2 becomes

$$|\lambda_1|(s-1) > |\lambda_2|$$

which holds by assumption. Therefore, Lemma 22 holds for $n = 2$.

Now suppose that $n > 2$, q is some feasible probability vector, and that λ is s-unbalanced at index 1. We wish to find an A satisfying Conditions 1 and 2. Consider $B = \{2, \ldots, n\}$. Note that for such B, $s_B + q_1 - 1 = s - 1$. By our unbalanced assumption, we know that Condition 2 holds for $B = \{2, \ldots, n\}$. If λ_B is $(s-1)$-balanced, then Lemma 24 implies that we are done.

Assume that λ_B is not $(s-1)$-balanced. After relabeling, we can assume it is unbalanced at $i = 2$. Let $C = \{3, \ldots, n\}$. Therefore,

$$|\lambda_2|(s-2) > \|\boldsymbol{\lambda}_C\|_1. \tag{D.9}$$

Combining this with the s-unbalanced assumption at $i = 1$, we find

$$\begin{aligned}|\lambda_1| &> \frac{\|\boldsymbol{\lambda}_B\|_1}{s-1} \\ &= \frac{|\lambda_2|}{s-1} + \frac{\|\boldsymbol{\lambda}_C\|_1}{s-1} \\ &> \frac{\|\boldsymbol{\lambda}_C\|_1}{(s-1)(s-2)} + \frac{\|\boldsymbol{\lambda}_C\|_1}{s-1} \\ &= \frac{\|\boldsymbol{\lambda}_C\|_1}{s-2}.\end{aligned}$$

Therefore,

$$|\lambda_1|(s - q_2 - 1) \geq |\lambda_1|(s-2) > \|\boldsymbol{\lambda}_C\|_1. \tag{D.10}$$

Let $D = \{1, 3, 4, \ldots, n\} = \{1, \ldots, n\} \backslash \{2\}$. Then note that Eq. (D.10) implies that $\boldsymbol{\lambda}_D$ is $(s - q_2)$-unbalanced at $i = 1$. Inductively applying this theorem, this means that we can find a vector $p' \in \mathbb{R}^{|D|}$ such that $p'_1 = 1$ and $f_D(\boldsymbol{p'}) \leq f_D(\boldsymbol{q})$. Moreover, $s_D(\boldsymbol{p'}) = s - q_2$. Therefore, if we let $\boldsymbol{p}$ be the vector that equals $\boldsymbol{p'}$ on D and with $p_2 = q_2$, we have

$$f(\boldsymbol{p}_2) = f_C(\boldsymbol{p'}) + \frac{\lambda_2^2}{q_2} \leq f_D(\boldsymbol{q}) + \frac{\lambda_2^2}{q_2} = f(\boldsymbol{q}).$$

This proves the desired result. □

D.3 Analysis of ATOMO via the KKT Condtions

In this section we show how to derive Algorithm 2 using the KKT conditions. Recall that we wish to solve the following optimization problem:

$$\min_{\boldsymbol{p}} \; f(\boldsymbol{p}) := \sum_{i=1}^{n} \frac{\lambda_i^2}{p_i} \quad \text{subject to} \;\; \forall i, \; 0 < p_i \leq 1, \quad \sum_{i=1}^{n} p_i = s. \tag{D.11}$$

We first note a few immediate consequences.

1. If $s > n$ then the problem is infeasible. Note that when $s \geq n$, the optimal thing to do is to set all $p_i = 1$, in which case no sparsification takes place.
2. If $\lambda_i = 0$, then $p_i = 0$. This follows from the fact that this p_i does not change the value of $f(\boldsymbol{p})$, and the objective could be decreased by allocating more to the p_j associated to non-zero λ_j. Therefore we can assume that all $\lambda_i \neq 0$.
3. If $|\lambda_i| \geq |\lambda_j| > 0$, then we can assume $p_i \geq p_j$. Otherwise, suppose $p_j > p_i$ but $|\lambda_i| \geq |\lambda_j|$. Let p' denote the vector with p_i, p_j switched. We then have

$$\begin{aligned} f(\boldsymbol{p}) - f(\boldsymbol{p'}) &= \frac{\lambda_i^2 - \lambda_j^2}{p_i} + \frac{\lambda_j^2 - \lambda_i^2}{p_j} \\ &= \lambda_i^2 \left(\frac{1}{p_i} - \frac{1}{p_j} \right) - \lambda_j^2 \left(\frac{1}{p_i} - \frac{1}{p_j} \right) \\ &\geq 0. \end{aligned}$$

We therefore assume $0 < s \leq n$ and $|\lambda_1| \geq |\lambda_2| \geq \ldots \geq |\lambda_n| > 0$. As above we define $\boldsymbol{\lambda} := [\lambda_1, \ldots, \lambda_n]^T$. While the formulation of Eq. (D.11) does not allow direct application of the KKT conditions, since we have a strict inequality of $0 < p_i$, this is fixed with the following lemma.

Lemma 25. *The minimum of Eq. (D.11) is achieved by some p^* satisfying*

$$p_i^* \geq \frac{s\lambda_i^2}{n\|\boldsymbol{\lambda}\|_2^2}.$$

Proof. Define $\overline{\boldsymbol{p}}$ by $\overline{\boldsymbol{p}}_i = s/n$. This vector is clearly feasible in Eq. (D.11). Let $\boldsymbol{p}$ be any feasible vector. If $f(\boldsymbol{p}) \leq f(\boldsymbol{q})$ then for any $i \in [n]$ we have

$$\frac{\lambda_i^2}{p_i} \leq f(\boldsymbol{p}) \leq f(\overline{\boldsymbol{p}}).$$

Therefore, $p_i \geq \lambda_i^2 / f(\overline{\boldsymbol{p}})$. A straightforward computations shows that $f(\overline{\boldsymbol{p}}) = n\|\boldsymbol{\lambda}\|_2^2/s$. Note that this implies that we can restrict to the feasbile set

$$\frac{s\lambda_i^2}{n\|\boldsymbol{\lambda}\|_2^2} \leq p_i \leq 1.$$

This defines a compact region C. Since f is continuous on this set, its maximum value is obtained at some $\boldsymbol{p}^*$. □

The KKT conditions then imply that at any point $\boldsymbol{p}$ solving Eq. (D.11), we must have

$$0 \leq 1 - p_i \perp \mu - \frac{\lambda_i^2}{p_i} \geq 0, \quad i = 1, 2, \ldots, n \tag{D.12}$$

for some $\mu \in \mathbb{R}$. Since $|\lambda_i| > 0$ for all i, we actually must have $\mu > 0$. We therefore have two conditions for all i.

1. $p_i = 1 \implies \mu \geq \lambda_i^2$.
2. $p_i < 1 \implies p_i = |\lambda_i|/\sqrt{\mu}$.

Note that in either case, to have p_1 feasible we must have $\mu \geq \lambda_1^2$. Combining this with the fact that we can always select $p_1 \geq p_2 \geq \ldots \geq p_n$, we obtain the following partial characterization of the solution to Eq. (D.11). For some $n_s \in [n]$, we have $p_1, \ldots, p_{n_s} = 1$ while $p_i = |\lambda_i|/\sqrt{\mu} \in (0, 1)$ for $i = n_s + 1, \ldots, n$. Combining this with the constraint that $\sum_{i=1}^n p_i = s$, we have

$$s = \sum_{i=1}^n p_i = n_s + \sum_{i=n_s+1}^n p_i = n_s + \sum_{i=n_s+1}^n \frac{|\lambda_i|}{\sqrt{\mu}}. \tag{D.13}$$

Rearranging, we obtain

$$\mu = \frac{\left(\sum_{i=n_s+1}^n |\lambda_i|\right)^2}{(s - n_s)^2} \tag{D.14}$$

which then implies that

$$p_i = 1, \quad i = 1, \ldots, n_s, \quad p_i = \frac{|\lambda_i|(s - n_s)}{\sum_{j=n_s+1}^{n} |\lambda_j|}, \quad i = n_s + 1, \ldots, n. \tag{D.15}$$

Thus, we need to select n_s such that the p_i in Eq. (D.15) are bounded above by 1. Let n_s^* denote the first element of $[n]$ for which this holds. Then the condition that $p_i \leq 1$ for $i = n_s^* + 1, \ldots, n$ is exactly the condition that $[\lambda_{n_s^*+1}, \ldots, \lambda_n]$ is $(s - n_s)$-balanced. In particular, Lemma 20 implies that, fixing $p_i = 1$ for $i = 1, \ldots, n_s^*$, the optimal way to assign the remaining p_i is by

$$p_i = \frac{|\lambda_i|(s - n_s^*)}{\sum_{j=n_s^*+1}^{n} |\lambda_j|}.$$

This agrees with Eq. (D.15) for $n_s = n_s^*$. In particular, the minimal value of f occurs at the first value of n_s such that the p_i in Eq. (D.15) are bounded above by 1.

Algorithm 2 scans through the sorted λ_i and finds the first value of n_s for which the probabilities in Eq. (D.15) are in $[0, 1]$, and therefore finds the optimal $\boldsymbol{p}$ for Eq. (D.11). The runtime is dominated by the $O(n \log n)$ sorting cost. It is worth noting that we could perform the algorithm in $O(sn)$ time as well. Instead of sorting and then iterating through the λ_i in order, at each step we could simply select the next largest $|\lambda_i|$ not yet seen and perform an analogous test and update as in the above algorithm. Since we would have to do the selection step at most s times, this leads to an $O(sn)$ complexity algorithm.

D.4 Hyperparameter optimization

We firstly provide results of step size tunning, as shows in Table D.1 we reported stepsize tunning results for all of our experiments. We tuned these step sizes by evaluating many logarithmically spaced step sizes (e.g.,

$2^{-10}, \ldots, 2^{0}$) and evaluated on validation loss.

This step sizes tuning, for 8 gradient coding methods and 3 datasets was only possible because fairly small networks were used.

Table D.1: Tuned stepsizes for the ResNet-18 model and the SVHN dataset.

Optimizer	Learning rate/step size
SVD rank 1	0.1
SVD rank 2	0.125
SVD rank 3	0.125
SVD rank 4	0.125
QSGD 1bit	0.0078125
QSGD 2bit	0.0078125
QSGD 4bit	0.046875
QSGD 8bit	0.125

We empirically study runtime costs of spectral-ATOMO with sparsity budget set at 1, 2, 3, 6 and made comparisons among b-bit QSGD and TernGrad. In some scenarios, spectral-ATOMO attains a higher compression ratio than QSGD and TernGrad. For example, singular value sparsification with sparsity budget 1 may communicate smaller messages than $\{2, 4\}$-bit QSGD and Terngrad.

E.1 Exhaustive query searches

Let's run a query search for the embedding in Figure E.1, which is chosen to be somewhat accurate but not close to perfect. When the information gain of all $N = n(n-1)(n-2)/2$ queries are scored, some *heads* get chosen to be the most informative and have higher information gain than any other head.

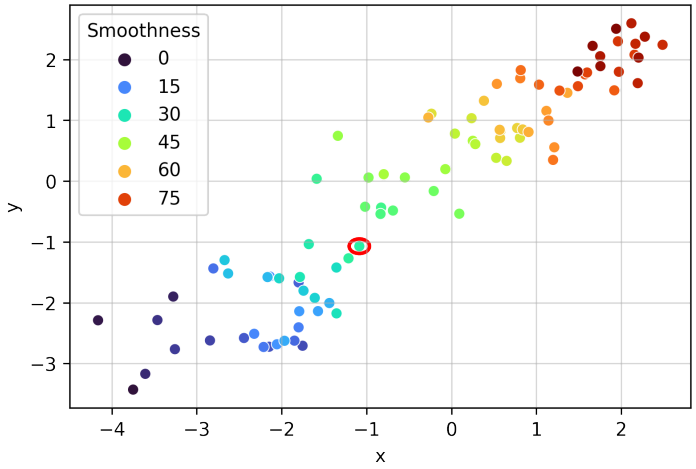


Figure E.1: The embedding on which query searches will be performed. This embedding has been learned from 1,800 human responses to randomly selected queries. Item 32 is circled with a solid red line.

The head with the highest information gain is item 32, and the top 849 queries have head 32. To see how this changes after a model update, let's answer the top 100 scoring queries (all with head 32) with a synthetic noise model and perform a complete model update[1] This embedding is shown in

[1]With 100,000 epochs and warm starting from the embedding in Figure E.1, more complete than Salmon's online embedding.

Fig. E.2, and shows some modifications around item 32, which is shown in Figure E.3.

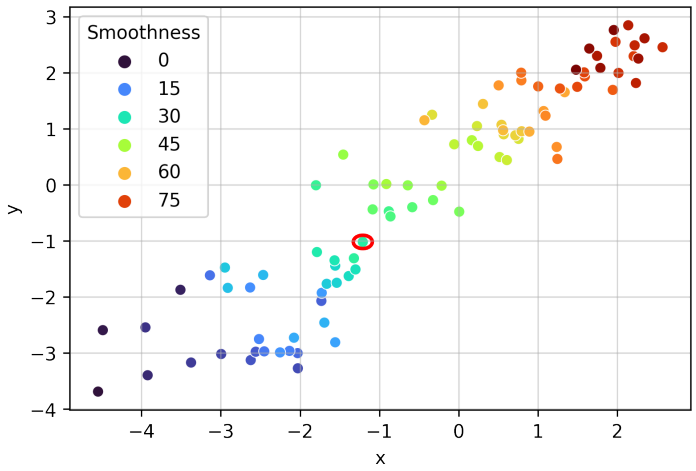


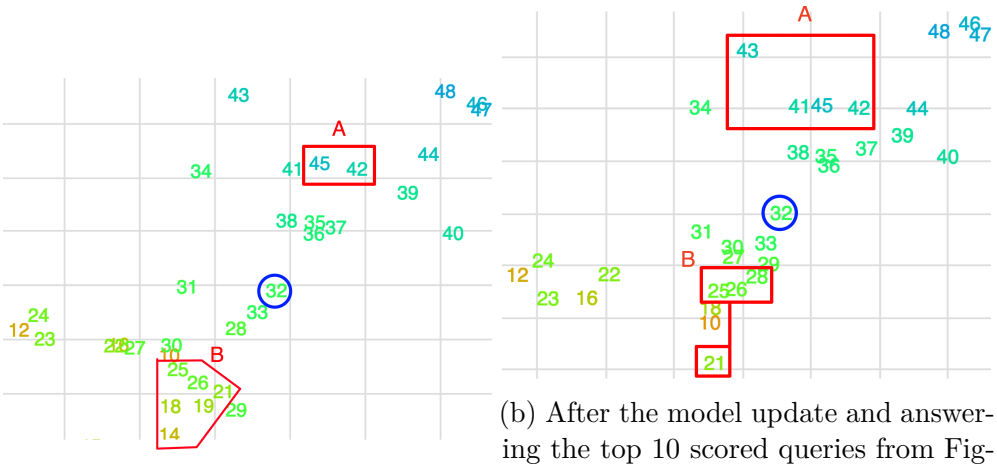
Figure E.2: The embedding on which query searches will be performed. This embedding has been learned from 1,800 human responses to randomly selected queries, with 100 responses from a synthetic noise model. Item 32 is circled with a solid red line.

After the model update, the head of the top 3,916 queries is item 32. Before the model update, only 54% of the top 3,916 queries had a head of item 32. Every one of the 849 questions from before the model update is included in the top 3,916 queries after the model update. A more complete illustration is in Fig. E.4

E.2 Adaptive search tuning

Salmon's search makes three modifications to NEXT's greedy search: search length and random head selection. This section will present simulation results validating each of those choices.

First, let's show how search length changes performance in the greedy search. In Figure E.5, the performance for different "greedy-k" searches



(a) Before the model update.

(b) After the model update and answering the top 10 scored queries from Figure E.3a with a synthetic noise model.

Figure E.3: Before and after the model update, the top 10 queries with the highest information gain had item 32 as head with one bottom from box A and one bottom from box B.

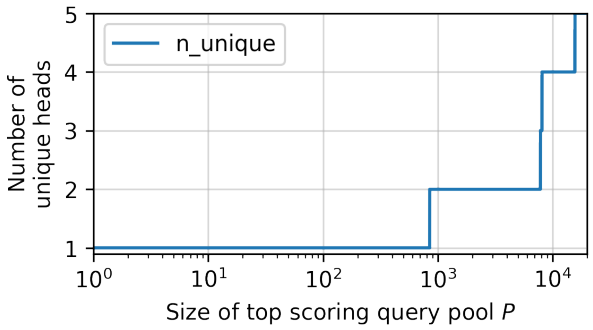


Figure E.4: The number of unique heads in the pool of the P queries with highest information gain. When ranked by information gain, the top 849 queries have the same head, item 32. The top 7,832 queries only have two unique heads, item 32 and item 35.

is shown. In each of these, between $k/3$ and k queries are searched per user (roughly), and the query with maximum information from that list is provided to the synthetic noise model. Figure E.5 shows that moderate searches tend to perform the best; both exhaustive and minimal searches do

not perform well. Perhaps that is due to the behavior shown in Figure E.6, which illustrates that the greedy searches can select the same head repeatedly, especially as the search length increases.

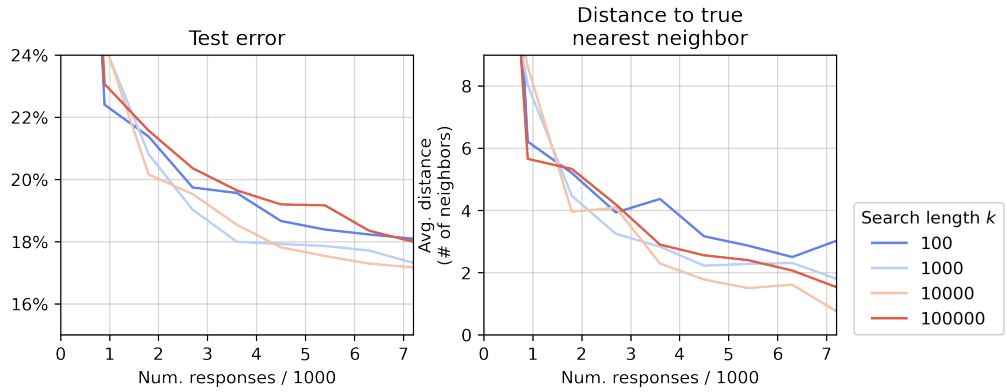


Figure E.5: The performance of different samplers that choose the query with maximum information gain from a list of up to length k.

Figure E.6, which shows different "greedy-k" that searches between roughly $k/3$ and k queries per user.

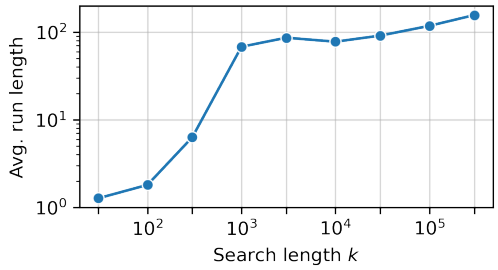


Figure E.6: The average "run" length for search length k. A "run" is defined to be "number of queries with the same head," and the query with maximum information gain is chosen from a list of length k.

Now, let's see how random head selection changes the performance. Again, let's plot different search lengths, and compare with the best of

the "greedy-k" searches in Fig. E.7.[2] Random head selection improves performance over the best "greedy-k" performance, and performs slightly better than any of the "greedy-k" searches. Search length seems somewhat important, but does not appear to be critical. Notably, the implementation of a round-robin scheme is a lot simpler than figuring out the correct value of k.

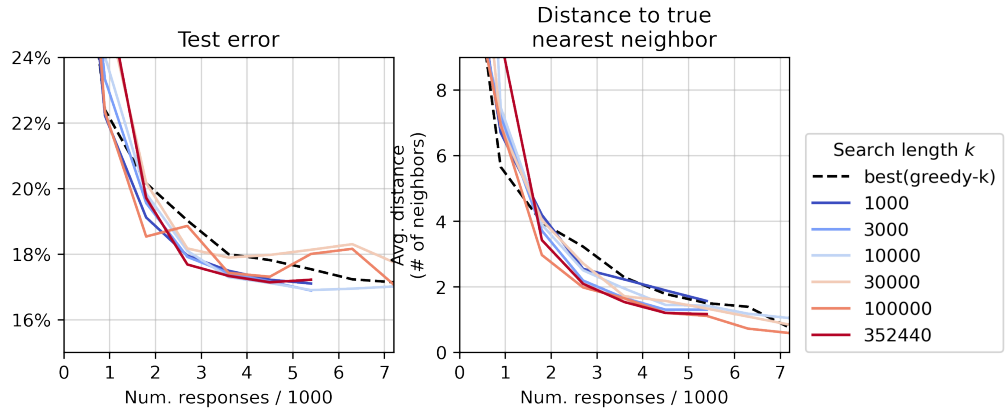


Figure E.7: The performance of different samplers that score k queries and randomly selects a head, then finds the best query with that head. In practice, k/n queries are expected to be searched per head.

E.3 Priority

The `ARR` and `Salmon` schemes in Sections 4.3.1 and 4.3.2 follow this process:

1. Perform an near-exhaustive query search
2. Find the top query for each query head.
3. Randomly assign scores to queries.

[2] In Figure E.5, we only show a subset of the greedy-k searches for simplicity; "best(greedy-k)" is the minimum error/nearest neighbor distance for $k \in \{30, 100, 300, \ldots, 100 \cdot 10^3, 300 \cdot 10^3\}$.

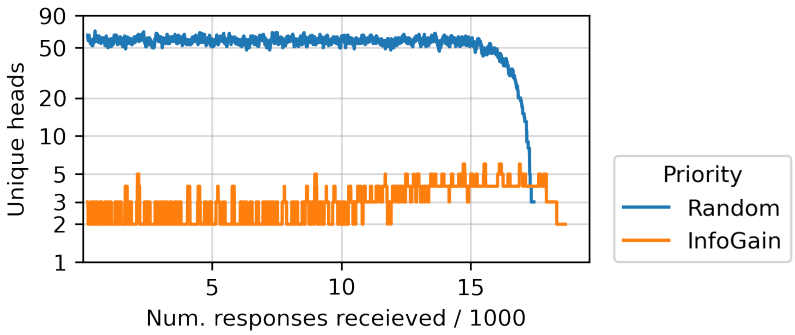


Figure E.8: The number of unique heads in the last 90 adaptively chosen queries for different priority schemes. The number of received responses includes both adaptively and randomly chosen queries.

4. Post those queries to the database.
5. When a query is requested, pop the highest scoring query of the database.

This section will modify step (3), and substitute these choices:

1. `Random`: randomly assign scores to each query.
2. `InfoGain`: set the score to be the query's information gain.

The second choice is very similar to Section 2.4.1. However, it exhibits the same "constant head" behavior in Appendix E.1 as shown in Fig. E.8. When embeddings are generated embeddings from simulated human responses, the embeddings from the `InfoGain` scheme perform worse than random sampling.