Beyond Model Efficiency: Data Optimizations for Machine Learning Systems

Michael Roman Kuchnik

CMU-CS-23-119

May 2023

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

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

This research was sponsored by Google, Los Alamos National Laboratory under award number 394903, Semiconductor Research Corp. under award number 2018JU2779, U.S. Army Contracting Office under award number W911NF20D0002, the National Defense Science and Engineering Graduate Fellowship, Cloud TPUs from Google's TPU Research Cloud, and research credits from Google Cloud Platform. This material is based upon work supported by the U.S. Army Research Office and the U.S. Army Futures Command under Contract No. W911NF-20-D-0002. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity. We thank the members and companies of the PDL Consortium: Amazon, Google, Hewlett Packard Enterprise, Hitachi Ltd., IBM Research, Intel Corporation, Meta, Microsoft Research, NetApp, Inc., Oracle Corporation, Pure Storage, Salesforce, Samsung Semiconductor Inc., Seagate Technology, Two Sigma, and Western Digital for their interest, insights, feedback, and support.

Keywords: Machine Learning Systems

Abstract

The field of machine learning, particularly deep learning, has witnessed tremendous recent advances due to improvements in algorithms, compute, and datasets. Systems built to support deep learning have primarily targeted computations used to produce the learned model. This thesis proposes to instead focus on the role of data in both training and validation. For the first part of the thesis, we focus on training data, demonstrating that the data pipeline responsible for training data is a prime target for performance considerations. To aid in addressing performance issues, we introduce a form of data subsampling in the space of data transformations, a reduced fidelity I/O format, and a system for automatically tuning data pipeline performance knobs. In the second part of the thesis, motivated by the trend toward increasingly large and expressive models, we turn to the validation setting, developing a system for automatically querying and validating a large language model's behavior with standard regular expressions. We conclude with future work in the space of data systems for machine learning.

Acknowledgments

To start, I am grateful for my advisors, George Amvrosiadis and Virginia Smith, for giving me the freedom to pursue topics that I found interesting. I entered graduate school thinking I wanted to work on machine learning systems. For a variety of reasons, my PhD journey was more “squiggly” than straight—knocking me off course into a figurative “middle-of-nowhere”. The patience and support my advisors have given me in steering the ship back on course is truly admirable, especially when there was only a vague idea of where I was supposed to go. I’d also like to thank the National Defense Science and Engineering Graduate (NDSEG) Fellowship for its generous support for a large part of my PhD.

I am grateful for the folks I’ve met and interacted with throughout my long years at Carnegie Mellon University (CMU). I’m grateful for my officemates in GHC 7001: David Witmer, Nika Haghtalab, David Wajc, Daniel Anderson, and Souradip Ghosh. While most of your research went right over my head, I will miss your company, conversation, and mentorship. David Witmer and Nika Haghtalab, in particular, had the “pleasure” of being my initial officemates when I just arrived at CMU and had an endless list of probably annoying questions. I am honored to have been part of two labs through my PhD, the Parallel Data Lab (PDL) and the Forge Lab. The PDL and those involved were one of the true constants throughout my PhD. The Forge Lab exposed me to a different research style and environment, and I’m grateful for the weekly group discussions that were had. Lastly, for time spent outside CMU, I’d like to thank the Thursday night dinner group for providing a consistent checkpoint for the particularly long weeks.

I am grateful for those who taught and inspired me throughout my career. For practical reasons, I’ll focus on mentorship experiences relevant to my graduate studies as I remember it. My interest in research can be attributed to Linda Wills, who provided me with mentorship and a research project as an undergraduate, ultimately leading me to pursue graduate school. Garth Gibson, James Ahrens, Elisabeth Moore, Nathan DeBardeleben, and Chuck Cranor were my guides through various aspects of High Performance Computing as well as my early career. Tri Dao answered probably too many of my naive machine learning questions. Abutalib Aghayev helped me pivot toward storage systems at a critical point in my career. Finally, Ana Klimovic and Jiří Šimša were my mentors during and after my internship at Google—this thesis would not be complete without their help. Lastly, I’d like to thank my thesis committee members, Tianqi Chen, Greg Ganger, and Paul Barham, for generously providing their guidance on shaping this work.

I am extremely grateful for my parents, Jan and Brygida. It is without a doubt that I owe my education and much more to their love, sacrifices, and support. I’d like to thank my brother, Matt, for simultaneously testing the limits of my patience while also providing an unending source of friendship. I’d also like to thank my extended family, who, perhaps, are too understanding for my limited visits as I finish this work. Lastly, I’d like to thank my best friend and partner, Deanna, for lovingly supporting me through this journey. You’ve somehow tolerated my midnight “clacking”, impending deadlines, and general disarray for years while we lived in the “box”. I couldn’t have done it without you!

Contents

Chapter 1

Machine Learning Data Pipelines

Machine learning (ML) has seen a recent boom in applications in the past decade. The field, which focuses on algorithms that improve as a function of data or experience [201], has evolved from a specialized set of applications (e.g., ads [195], recommendation systems [60, 106, 213], spam detection [316]) to being applied to nearly every technical discipline. For example, deep learning is applied to gameplay [261, 286], protein folding [143], robotics [80], a range of natural language processing tasks [43, 55], and is projected to reach a point of ubiquity that may lead to major economic disruptions [87].

At the forefront of this revolution has been the subfield of deep learning [108, 173]. Deep learning uses a cascade of *layers*—mathematical operations—to assemble a model. The layers are jointly learned such that earlier layers simplify the task at hand for subsequent layers. Deep networks, while perhaps less theoretically understood than other ML or AI approaches, have shown that methods that are general yet computationally taxing eventually dominate algorithms that take advantage of additional specialization [268]. Such computationally taxing, yet general, methods have benefited from trends such as Moore's Law [209]—the exponential scaling of hardware performance—as well as hardware and software specialization [165, 275]. Today's plethora of deep learning software raises a case that deep learning is perhaps even *more accessible* today than the alternatives—training a state-of-the-art model merely requires access to model specification code, which is usually open-source and readily available. The core of deep learning technology has been both commoditized and democratized, allowing anyone to benefit from human-years of research and development.

However, while routine aspects of using deep learning are more accessible, there are still fundamental problems that remain to be solved and impact the downstream performance of many applications. One way to taxonomize these problems (and their corresponding solutions) is by grouping them between the three areas of 1) *ML algorithms*, 2) *compute*, and 3) *data*. Each of these three areas have been optimized to continue advancements in the field and have been cited as key sources that led to the rise of deep learning [35]. For instance, a lack of *training data* and *compute*, are attributed to the decline of deep networks in the early 2000s [35]. The lack of these factors was perhaps not rectified until a decade later, when record performance was achieved on the 2012 ImageNet Large Scale Visual Recognition Challenge (ILSVRC) competition [71], The winning submission, AlexNet [156], was a deep convolutional neural network (CNN) and had been trained on a million images with the help of graphics processing units (GPUs). There have also been advances in *ML algorithms*, which have made learning more efficient. For instance, ReLU activations and dropout, mathematical operations widely used to accelerate learning, were key algorithmic components of the 2012 submission [156, 173]. When these advancements were combined, the resulting model had surpassed the next best submission by a significant 10% absolute error and started a revolution in computer vision [173]. Today's recent trends in natural language processing can be similarly viewed as a core algorithmic innovation [285] being scaled to massive data and compute [43, 55], resulting in predictable advancements in performance.

At the forefront of democratizing machine learning are *machine learning systems* [239]. These systems encompass and solve sufficiently formulaic parts of the machine learning methodology, enabling practitioners to focus their time on other problems. If ML algorithms, compute, and data are the pillars holding up modern ML, then machine learning systems are the tools used to put them into place. Today's systems include facilities for symbolic manipulation of mathematical expressions, portability across a variety of hardware platforms, distributed execution, and libraries pre-packaged with commonly used utilities and mathematical expressions [11, 49, 96, 222]. The choice of functionality to support in these frameworks goes beyond a matter of convenience—support can change the direction of the field as a whole [29, 128]—thus motivating studying which avenues future ML systems should invest in.

As part of this thesis, we explore several directions for new or revised functionality in the modern machine learning stack, placing *a focus on the treatment of data* throughout the stack. Data is important to study because it is the most dynamic of the three problem areas—data can always be refined to cover additional samples, additional features, or certain types of behavior, while a model (and compute) is necessarily fixed to function over some type of data. Furthermore, optimizations to data can result in significant application gains, spurring research in *data-centric AI* [3]. However, the lunch is not free and changes to data are easier said than done. A lack of theoretical understanding means that, for any new type of machine learning task, a practitioner will likely have to test what combination of data works best. Without proper data abstractions, a single change in the task may cause a practitioner to manually assess and tune the characteristics of the application's data. Not only is it tedious to tune aspects of the data, the act of characterizing the model's as well as the system's performance as a function of data requires expertise in machine learning and systems, respectively—expertise that is typically split among different groups of people. If the goal of ML systems is to support practitioners in solving repetitive problems, then it is reasonable to expect ML systems to enable rapid configuration and prototyping with respect to data. Simply put, data pipelines should be *first-class citizens* in the ML systems stack—they should not be an afterthought bolted on to tools that support models and compute.

The organization of the rest of this chapter is as follows. First, we give an overview of how machine learning systems are built and evaluated (§1.1). We then provide an overview of how the workloads found in machine learning are fundamentally changing, splitting the community into two (§ 1.2) and motivating a fundamentally different treatment of machine learning systems. We then turn to motivation for the thesis, revisiting the significance of data in the current machine learning environment (§ 1.3). Finally, we introduce the thesis statement and provide an overview into the chapters of this document (§1.4). Readers familiar with the current state of machine learning and machine learning systems may skip the "textbook material" of Section 1.1 and Section 1.2 and move directly to Section 1.3.

1.1 Machine Learning Systems

This thesis places a heavy emphasis on the design of machine learning systems, which solve certain parts of the machine learning methodology. In this section, we specify the focus of this thesis in the language of both machine learning and systems (§1.1.1). We begin by introducing machine learning from the theoretical perspective so that we can theoretically define the scope of this thesis. We then switch to describing the components of machine learning systems in § 1.1.2, allowing us to describe the focus of this thesis in terms of logical components. Afterwords, we define how machine learning systems optimizing certain components can be evaluated (§ 1.1.3).

1.1.1 Machine Learning Theory

Machine learning's goal is to *fit* a model to data such that the model is expected to perform well on future, potentially unseen, data. Since this thesis focuses on deep learning, we briefly review the mathematical language used in that area of machine learning. Let $(\boldsymbol{x}, \boldsymbol{y}) \sim \mathcal{D}$ be examples of inputs, $\boldsymbol{x}$, and labels, $\boldsymbol{y}$, drawn from the training distribution, $\mathcal{D}$. The inputs, $\boldsymbol{x} \in \mathbb{X}$, can be imagined to be a vector representing some real-world data, such as images, audio, text, and the labels, $\boldsymbol{y} \in \mathbb{Y}$, can be thought of as metadata about $\boldsymbol{x}$ (e.g., what object is contained in the image) or data which is somehow missing from $\boldsymbol{x}$ (e.g., pixels that are left unspecified)[1]. The goal is to learn a function $f : \mathbb{X} \rightarrow \mathbb{Y}$ that is accurate in predicting the label, $\boldsymbol{y}$, from the input, $\boldsymbol{x}$. Specifically, we seek a function $f \in \mathbb{F}$ that minimizes the expected *loss*, $\mathcal{L}$, on unseen test data, $\mathcal{D}_{\text{test}}$: $\min_{f \in \mathbb{F}} \mathbb{E}_{(\boldsymbol{x}_{\text{test}}, \boldsymbol{y}_{\text{test}}) \sim \mathcal{D}_{\text{test}}} \mathcal{L}(f(\boldsymbol{x}_{\text{test}}), \boldsymbol{y}_{\text{test}})$. The loss, $\mathcal{L}$, is a function mapping from two labels to a real, non-negative, number: $\mathcal{L} : \mathbb{Y} \times \mathbb{Y} \rightarrow \mathbb{R}^{+}$. If the loss is binary—1 for incorrect predictions and 0 for correct predictions—we are finding the function f that gives us maximum accuracy. In the general case, the loss can be intuitively thought of as a "distance" between the predicted label and the actual label, and it can be Euclidean distance in the case of regression, for example. Since $\mathcal{L}$ can be expected to be 0 when $f(\boldsymbol{x}) = \boldsymbol{y}$, a perfect predictive function, f, would have 0 loss. However, such a perfect function is not necessarily achievable because the relationship between labels and inputs in $(\boldsymbol{x}_{\text{test}}, \boldsymbol{y}_{\text{test}}) \sim \mathcal{D}_{\text{test}}$ may have elements of true randomness (e.g., noise or labeling error)—in that case we want the expected loss to be as low as we can expect it to be.

There are a few issues with this formulation. First, data from $\mathcal{D}_{\text{test}}$ is supposed to be unseen—at the very least, we aren't supposed to have labels. Thus, we cannot directly fit a model to data from $\mathcal{D}_{\text{test}}$. We can, however, assume that $\mathcal{D}$ and $\mathcal{D}_{\text{test}}$ are identical distributions. Thus, we can directly minimize over the training set: $\min_{f \in \mathbb{F}} \mathbb{E}_{(\boldsymbol{x}, \boldsymbol{y}) \sim \mathcal{D}} \mathcal{L}(f(\boldsymbol{x}), \boldsymbol{y})$. Second, we don't have access to $\mathcal{D}$ in general; instead, we have some representative samples $\left\{(\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)}) \sim \mathcal{D} : \ \forall i \in \{1, 2, \ldots, N\}\right\}$. We can thus minimize expected loss on $\mathcal{D}$ in the finite sample case: $\min_{f \in \mathbb{F}} \frac{1}{N} \sum_{\left\{(\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)}) \sim \mathcal{D} : \ \forall i \in \{1, 2, \ldots, N\}\right\}} \mathcal{L}(f(\boldsymbol{x}^{(i)}), \boldsymbol{y}^{(i)})$. Third, with a sufficiently powerful model, we can simply memorize parts of $\mathcal{D}$ e.g., if $f \in \mathbb{F}$ is a hashtable that stores all samples. Memorization is undesirable because it is a cause of *overfitting* and *poor generalization*—where the model fits to noise in the data more than an actual trend. The hashtable has an especially degenerate case of generalization because it is undefined outside the training set and thus has no

[1]Note that we treat both $\boldsymbol{x}$ and $\boldsymbol{y}$ as vectors for generality, though $\boldsymbol{y}$ may be a scalar for some tasks.

generalization. However, if we instead use the nearest sample's label (i.e., *nearest neighbor*), we can see that only a single sample is responsible for the prediction—thus, the prediction cannot be said robust to small changes in the data. To avoid overfitting, we can empirically evaluate over an additional "validation" split of data, $\mathcal{D}_{\text{valid}}$, which functions as $\mathcal{D}_{\text{test}}$, but has ground-truth labels to estimate the validation loss. If the model performs well on $\mathcal{D}_{\text{valid}}$, we can expect it will also perform well on $\mathcal{D}_{\text{test}}$. Forth, we haven't defined $f \in \mathbb{F}$ and how it is found. If $\mathbb{F}$ is very big (or infinite), we can't feasibly find f by simply iterating over $\mathbb{F}$—we need a better algorithm, which means we need to make some assumptions on the structure of $\mathbb{F}$. For deep learning, the key idea is to use a differentiable loss, $\mathcal{L}$, combined with a differentiable model, $f(\boldsymbol{x}; \boldsymbol{\theta})$, such that calculus can be applied to find changes in $\boldsymbol{\theta}$ that reduce the loss.

Remarks on Seemingly Unspecified Data Algorithms

The theory described above focuses extensively on how to fit the model. However, *it never defined how to create the data*. The algorithm for computing the training data, $(\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)}) \sim \mathcal{D}$, is not defined because it is application dependent—no single treatment exists—leaving a number of questions for practitioners. Where do $(\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)})$ come from? Are they easily computable? What order do these samples arrive in—is i arbitrary? If a list of samples defines the training distribution $\mathcal{D}$, then are we sampling from the list uniformly? Can we resample the same samples multiple times? If the data is media data, what is the impact of quality and resolution? What domain knowledge can we incorporate into the data to make the problem easier to learn—can we encode *invariances* into the problem by transforming samples in a way that preserves their labels (e.g., an image rotation)? Should we normalize the data in some way? Since most of these decisions are not independent, how do they compose with each other and what order do you apply them in? In abstracting away implementation details of data, we've only specified the solution for a general class of modeling problems. However, this thesis attempts to demonstrate that the "implementation details" of data matter, both from a machine learning point of view as well as a systems one—a practitioner faces many potential design and implementation decisions when it comes to data.

ML-Oriented Thesis Theme: *A theme of this thesis is investigating the design and performance of training data* $(\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)}) \sim \mathcal{D}$ *as well as testing data* $(\boldsymbol{x}^{(i)}_{test}, \boldsymbol{y}^{(i)}_{test}) \sim \mathcal{D}_{\text{test}}$. *The system's performance implications of sampling from the training set are given in §2 and §5, a method for choosing samples (i.e., values of* i*) is given in §3, a method for approximating* $\mathcal{D}$ *is covered in §4, and a system for specifying* $\mathcal{D}_{\text{test}}$ *is given in §6. From a machine learning point of view, this thesis is interested in addressing tradeoffs between system performance and model generalization from the point of view of data.*

1.1.2 The Machine Learning Components

A machine learning configuration can be broken up into a few components. A configuration is typically involved in an interactive *workflow* [32], where the preferable configurations are found via trial and error. In essence, the theory presented in §1.1.1 must be broken down into a process or algorithm that is fully or semi-automated. We outline the major components involved in training below and show the flow of data in Figure 1.1. In short, data stored as raw bytes is transformed into a matrix (or matrix-like object), and the matrix can be efficiently operated over with an accelerator. This thesis primarily focuses on the first two blocks.

Data and Task. The *data* is a set of examples, each of which has a fixed set of features and, optionally, labels. For example, a dataset may be an album of cat and dog images, where the features are the pixels of each image, and the label is whether an image is of a cat or dog. The input to a machine learning algorithm is *training data*, and held-out (unseen) data used to evaluate the algorithm is *validation data* or *test data*. The machine learning *task* is to predict some features or labels for the unseen data as a function of the training data (e.g., predictions to label unlabeled data when trained on labeled data). The data and task are entirely determined by the practitioner and vary significantly. Domain experts may have knowledge of task-specific *invariances*, which are captured by *augmenting* training or testing data to improve the model's performance by effectively increasing its dataset size. Typical examples of augmentations are translations, rotations, and crops for image classification, and their benefits include increasing the effective size of training data as well as reducing variations in test-time predictions [156]. The system responsible for feeding input into the model is the *data loader* or *input pipeline*. Note that while the term "data pipeline" is overloaded to cover any process that moves data, for the purpose of this thesis, we treat it as synonymous with data loader and input pipeline—this thesis is only concerned with data systems interacting with a model during training or testing.

Model and Optimizer. The *model* is the part of a machine learning that learns from the data. Given an input, the model returns a prediction of the output. Models encompass everything from decision trees to logistic regression to deep neural networks. This thesis focuses on models that have a fixed set of parameters, which can be tuned by an *optimizer*. The optimizer's goal is to maximize the model's performance on the task; examples include batch gradient descent, stochastic gradient descent, and ADAM [41, 151]. *Training* is the part of the machine learning workflow where the optimizer uses training data to update the model. The subsequent *testing* phase treats the model's parameters as fixed, removing any need for the optimizer—this phase runs over the testing data to form predictions for unknown features or labels. While the model and optimizer are separately configurable, we will refer to the combination of both as just "model" for short. The choice of modeling framework is up to the practitioner, though practitioners often limit themselves to well-studied models that have been shown to robustly generalize across a variety of datasets and tasks.

Hardware and Software. Each of the above pieces needs to be run on a physical computer, composed of *hardware* components. Examples include a disk drive for storing data, a central processing unit (CPU) for performing data manipulation and orchestration, and an accelerator, such as a graphics processing unit (GPU) [215], tensor processing unit (TPU) [109, 140, 165], field programmable array (FPGA), or some other application-specific integrated circuit (ASIC) [137, 248]. *Software* is what enables such a hardware setup to be programmed to do a variety of tasks—enabling the same hardware setup to run an effectively infinite number of combinations of data, task, and model. While software is an integral part of all layers of modern computer systems, we pay particular attention to *machine learning frameworks*, which abstract common application logic into a ML-specific language or library. Frameworks such as `TensorFlow` [11], `JAX` [96], `PyTorch` [222], and `MXNet` [49] provide practitioners with utilities to create and run *tensors*, which generalize matrices, as well as other mathematical functions. The frameworks either implicitly or explicitly model how to schedule and execute these operations on a variety of hardware and

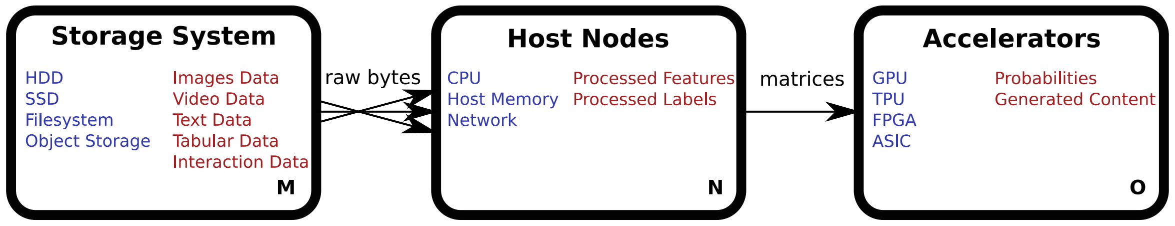


Figure 1.1: The flow of data from storage to host nodes to accelerators. Storage may be disaggregated from host nodes (communication represented by crossed arrows), which are potentially distributed but directly connected to their own accelerators (represented by a straight arrow). Data may be compressed in storage, decompressed and processed in the host nodes, and finally passed as numeric matrices into the accelerators. Text in blue represents hardware and systems. Text in red represents data. The bottom right hand corner of each block represents that if the system is represented as a graph with vertices and edges $G = (V, E)$, each of these blocks may have the number of vertices $|V|$ equal to M, N, or O, respectively—E is otherwise constrained to allow the shown inter-block communication with potential intra-block communication.

can do so automatically. In turn, users do not have to understand how to best execute matrix-heavy workloads, but rather express them in a form similar to their mathematical expression.

Systems-Oriented Thesis Theme: *A theme of this thesis is investigating the design and performance of training data pipelines as well as testing data pipelines. We investigate the extent to which training pipelines impact datacenter performance in §2 and we propose to fix a class of performance issues with automatic tuning in §5. We additionally investigate the role of approximation in training pipelines by selectively omitting the generation of samples in §3 as well as selectively downsampling training images in §4. Finally, we provide a system for specifying a user-defined test set by using regular expression programming with §6. From a systems point of view, this thesis is interested in abstractions that reduce data pipeline configuration from the point of view of the user.*

1.1.3 The Metrics

Practitioners are concerned with two broad metrics when evaluating configurations of machine learning components: the system's *utility* compared to its *cost*.

- **Utility.** First, how good is the resulting model's statistical performance i.e., its *utility*? In the simplest case, this refers to the model's *generalization*, or the ability of the model to perform on the test data. Typical metrics used to measure generalization are accuracy and residuals. For simplicity, we will refer to all forms of generalization as "accuracy", and they may encompass both learning effects as well as error introduced by the implementation (e.g., reduced precision [199]). For model testing, we are interested in how accurately the test reflects the true value of the intended measurements (e.g., if accuracy metrics for a model reflect performance in production). Note that models can have high utility yet may be incorrect due to unaccounted for dimensions of performance.
- **Cost.** Second, how much does a workflow *cost*, in terms of resources? Cost can include hardware time, programmer effort, energy, training job duration, or test construction effort.

These non-monetary costs can often be converted to a monetary value required to complete the workflow.

A configuration is arguably better than an alternative if it performs better in terms of at least one of these metrics, and it is strictly better if it performs better across all metrics. The former is called *Pareto efficient* or *Pareto optimal*. The latter can be referred to as *optimal*, and *Pareto dominates* all other solutions. Machine learning systems should return configurations that are arguably Pareto efficient. For instance, the system may default to using a numeric representation which is lower precision if it delivers higher performance—as long as a higher precision format does not match this performance, the choice is sensible. Users and applications may not treat all configurations equally, so it is also reasonable to expect the system to expose some flexibility in choosing between configurations.

It is worth noting that such a treatment of metrics is theoretical and is often approximated in practice. Today, it is uncommon for systems to be fully automated because there isn't a complete understanding of how configurations impact utility as well as cost. For instance, choosing between numerical representations requires an understanding of application error bounds with respect to numerical approximation error as well as an understanding of the performance properties of the hardware using these different representations. Not only is the set of numerical representations too large to iterative over, systems only have efficient implementations of a few of the representations (e.g., IEEE-754 floating point). When the ingredients required to make an informed decision on tradeoffs do not exist, the system will either have to be configured empirically over a benchmark or heuristically using common defaults. The effort required to find the configuration will still be accounted toward the cost; however, without rigorous analysis, it is not possible to guarantee the solution is optimal. Nevertheless, the set of Pareto optimal configurations is useful in establishing an absolute, rather than a relative, scale of performance—it is, by definition, impossible to dominate the performance of these configurations, making them ideal points of comparison for any system.

1.2 A Simplified Worldview: A Dichotomy of ML Models

Recent trends have disrupted the traditional workflows associated with machine learning. Machine learning practice has made it clear that general, data-driven methods can usually outperform specialized methods [268] once sufficient data is obtained. To simplify: there are two types of models in the world: 1) small, specialized ones and 2) large, general ones. Thus, it is interesting to wonder what would happen if compute and data were scaled to their limits (e.g., training on the entire internet) for a fixed learning algorithm. A sufficiently large model would be able to take advantage of all of the data—potentially solving the learning problem to optimality. If this were to happen—a class of learning algorithms were to be "solved"—then machine learning systems used for training would have limited usefulness—they'd only be used once to train that model. Even if the trained model turns out to be an approximation of the optimal model (e.g., within a factor ϵ of the optimal model's performance), the cost of retraining would be astronomical and it may be hard to justify a full retrain. This thought experiment is reflective of what is happening today with the rise of *large pretrained models*—also known as *foundation models* [40]—in general and Large Language Models (LLMs) in particular.

model type	training time	training cost	training frequency	inference usage
task-specific	minutes to days	\$10-\$1k	hourly to weekly	bulk streaming
large pretrained	weeks to months	\$100k-\$100M	yearly	interactive

Table 1.1: A breakdown of traditional task-specific machine learning models compared to the more recent large pretrained models. Workloads are fundamentally different between the two categories because the investment in large pretrained models is orders of magnitude higher. Large pretrained models are train-once and can be used on a broad set of downstream tasks, albeit with high computational expense.

Large pretrained models make the case that training a large model on a large amount of data using a large amount of compute may outperform a smaller, more specialized model in terms of predictive accuracy. Specifically, the larger model may perform well on tasks that were difficult for smaller models—a term called *emergent ability* [298]. For example, a large pretrained model can be applied in the *zero-shot* setting—where it is only evaluated on the test set without seeing the training set—and it may perform competitively compared to a smaller model that learned directly from the training set [236]. Alternatively, there is the *few-shot* setting, where a handful of examples are given at inference time. This setting is powerful because it allows just enough task-specific tuning to solve complicated tasks—through the process of generating generic text, a large pretrained model can learn the basic rules of arithmetic [43]. The significance of emergent abilities cannot be understated—if certain larger models perform well on many tasks without specialized training—what point is there to train smaller models? In some sense, large pretrained models are the next iteration of bitter lessons [268]—if deep learning trumps hand-engineered features, then large pretrained models trained over massive datasets trump deep learning models limited to smaller, specialized datasets. However, as usual, the lunch is not free—these models can still be outperformed in terms of training cost, inference cost, or application accuracy if enough effort is put into a specialized model. Therefore, the constraints of an application will dictate whether a large pretrained model will be used in production.

We take note of *two broad categories of machine learning models* to help forecast their training workloads and inference usage, as shown in Table 1.1. First, there are traditional machine learning models, which are highly specialized to a task and are generally efficient in terms of number of parameters. We call the models in this bin *task-specific models* because they solve specific well-studied problems used in production, such as recommendations, spam detection, and image classification [122]. Second, there are *large pretrained models* [40], which are much larger than task-specific models in terms of model size and resources utilized. Currently, these models are most popular in natural language processing because the subproblem of understanding the semantics and syntax of grammar is common across most natural language processing tasks, and thus solving it once can translate to many downstream applications. Unfortunately, these models can be so large that a single application inference can be over \$3[2]. With inference being so expensive, large pretrained models are most cost-effective on complex, high-value tasks, such

[2]GPT-4 at \$0.12 per 1000 sampled tokens at a context length of 32k is \$3.84 [217]. ChatGPT is estimated to cost \$700k per day [208].

as writing and programming [87]. In contrast, high-volume, low-value tasks, such as filtering spam, are more appropriately solved with traditional task-specific models. Additionally, since large pretrained models are so costly to train[3], they cannot be easily retrained from scratch and thus necessarily suffer from a "freshness" problem[4]

Overall, it's reasonable to expect application costs to be a dominant factor in the decision to choose between task-specific or large pretrained models. Task-specific models are more amenable to automation and running continuously because they are generally cheaper to train, cheaper to deploy, and easier to monitor and debug. On the other hand, the training of large pretrained models is so high stakes that manual intervention is justified [7, 8]. Since the large pretrained models are trained once, but deployed for a long time, it is reasonable to expect a significant fraction of cost to be incurred logically closer to the application, via inference expenses or runtime bugs.

1.2.1 The Difficulty of Testing: Knowing What You Don't Know

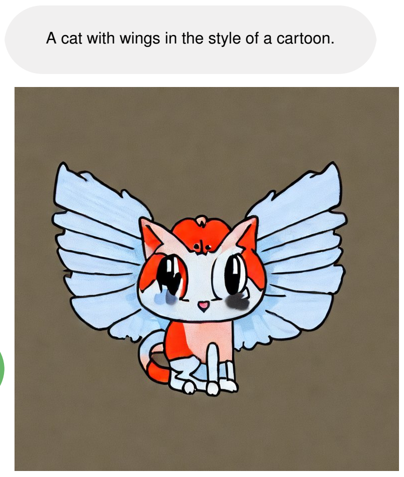
The relevance of this dichotomy to this thesis is that task-specific models are meant to be automated while large pretrained models are meant to be interacted with in a potentially creative way (i.e., exploiting their emergent behavior). In Figure 1.2, we can see the effects that large pretrained models may have on machine learning systems through the lens of data. Data from image classification datasets, such as ImageNet [71], poised a conceptually straightforward task: given an image of an object, return which of the 1000 classes of objects is best represented by it. While this task presented computer vision and machine learning challenges, it was obvious how to check for accuracy—with only 1 correct class for each image, a prediction was correct only if it matched the correct class. In code, this was as simple as `is_correct = y_pred == y_label`, for prediction `y_pred` and class label `y_label`. Similarly, during training, it was easy to reason about the images—for example, a crop or rotation of the image should preserve the class label. However, today's large pretrained models are not only able to be dynamically parameterized with the space of classification labels [237], they're capable of generating free-form examples of both images and text. Stable Diffusion [250] and ChatGPT [5], for example, are capable of forming images and text, respectively, given a prompt specifying what type of output they should have. These outputs are no longer easy to check—they are high-dimensional (e.g., pixels in an image or character strings in text), and we can't make strong assumptions about how transformations to the prompt will change what the model perceives. The combination of these factors means it is hard to predict the error-modes of large pretrained models.

To demonstrate how difficult it is to understand large pretrained model behavior, consider the case of natural language understanding. Humans have long used language to communicate with each other, yet an algorithm for understanding all human language has not yet been found. On the other hand, there is a convincing case that human language is understood by large pretrained models [228]. Therefore, since we do not know of an algorithm that parses human language, we can use large pretrained models to read human language, but we cannot algorithmically determine the correctness of the model behavior. Another form of this issue is in AI alignment, where it is a matter of debate of how to algorithmically specify the value-system of an AI agent [149]. It is

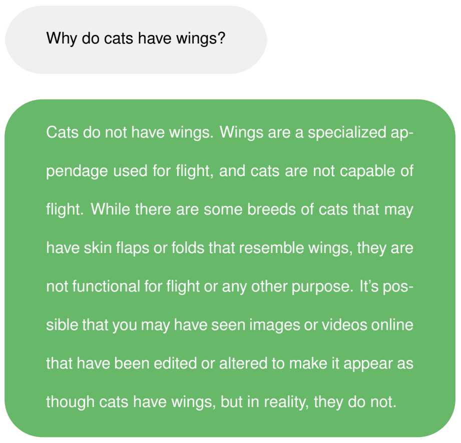
[3]GPT-4 is estimated to cost over $100 million to train [303].

[4]For instance, ChatGPT was released in November 2022 [5], yet it was trained on data from 2021 and before [94].

(a) ImageNet



(b) Stable Diffusion



Why do cats have wings?

Cats do not have wings. Wings are a specialized appendage used for flight, and cats are not capable of flight. While there are some breeds of cats that may have skin flaps or folds that resemble wings, they are not functional for flight or any other purpose. It's possible that you may have seen images or videos online that have been edited or altered to make it appear as though cats have wings, but in reality, they do not.

(c) ChatGPT

Figure 1.2: Various notions of cats according to classification datasets (Figure 1.2a), as well as more recent generative models (Figure 1.2b and Figure 1.2c). While it used to be the case that manipulating data was conceptually straightforward, today's generative models are capable of *producing* arbitrary images and descriptions of objects which stretch the boundaries of free-form creativity. While predictions over Figure 1.2a require a mere equality check for "Persian cat", it is much more difficult to automatically assess the free-form content found in Figure 1.2b and Figure 1.2c. Data systems will have to meet the needs of both specialized and open-ended content in the future during both training and inference.

thus likely that we cannot verify the correctness of large pretrained models in general because the only practical way to do so is to develop algorithms that can be proven to satisfy similar properties without any aspects of learning.

In effect, the "art" and "science" of large pretrained models comes at a disagreement because testing them requires knowing what you don't know. Unlike task-specific models, which have relatively restricted output spaces (e.g., a limited number of classes) and reasonable baselines (e.g., hand-engineered features), it seems fundamentally more difficult to reason about large pretrained model errors. Large pretrained models can make syntactic as well as semantic errors, and applications may desire that the output isn't offensive, nonfactual, or biased—each of which may require extensive testing to rule out. The first chapters of this thesis are set in a world of task-specific models, leaving one chapter on addressing the world of large pretrained models.

1.3 The Case for Data-Centric Machine Learning

A rising trend in the machine learning community is that of *data-centric machine learning* [3, 6, 229]. The observation is that, while models have been the focus of the machine learning community, the bulk of practitioner time is spent on tuning the dataset. From a research point of view, nearly all time is spent assuming the dataset quality is fixed and effort must be spent on improving modeling methods. However, in practice, nearly all time is spent assuming the model is good enough and effort must be spent on improving the data. These trends are again shifting modeling research toward addressing practical problems such as distribution shift, labeling error, privacy, and fairness. From a systems point of view, this paradigm shift means that more effort will be placed on programming aspects of data, placing a larger emphasis on the performance and usability of machine learning data systems.

1.3.1 Historical Context

The work in this thesis began in 2018, 3 years before the term "data-centric AI" [3] was coined. While data-centric ideas were already in existence, they had yet to hit mainstream. Machine learning systems research was predominantly *model-centric*—the narrative at the time was that model architectures and model scale were dominant factors holding back the practice. Accelerators, such as GPUs, were widely available, but it was difficult to automatically parallelize a model among a cluster of networked computers. The *bottleneck* at the time was perceived to be a lack of software to coordinate the parallelization and communication of a model's layers and gradients, motivating work in auto-parallelization and gradient compression. At the time, it was unorthodox to look at data systems for machine learning, because making a model larger would lead to more interesting computational problems and arguably a higher accuracy model. For instance, it is estimated that AlexNet and ResNet-50 each took about a week to train at the time of their publications [113, 156]. These models were considered *huge* given the resources of the time and a practitioner would *of course* want an even bigger model, if given the chance.

The narrative has changed dramatically in the years since, due to advances in both hardware and software. Today, a single compute node typically has $8\times$ high-end accelerators, which are sufficient to train many models without going over the network. Each compute node is connected with high-speed interconnects [140, 141]. The parallelization software is very good—capable of parallelizing models across thousands of accelerators in this environment [165]. Machine learning and HPC clusters are effectively converging—many of the Top-500 supercomputers are tuned for AI workloads, featuring the same GPUs used for ML training [9]. Today, ResNet-50 can be trained in 11.5 seconds to MLPerf Training standards for approximately \$50 [204]—tuning these models is now potentially bottlenecked by the time it takes for a practitioner to simply *think* about the experiment, much less manually optimize the model [5]. Furthermore, the narrative that scaling models will solve all problems has taken a complicated twist. On one hand, experts argue that current architectures are fundamentally flawed [170]. On the other hand, larger models trained over larger datasets are having breakthroughs in more challenging domains, such as natural language processing [43, 55]. Automatic parallelization and training optimizations are as important as ever for these massive models, but for many practical applications, the rush for training larger models seems over.

The community has since adapted to these changes. MLPerf now has a Data-Centric AI leaderboard called DataPerf [193]. Papers are being written about data pipeline bottlenecks [134, 206, 322]. On the large model front, researchers and practitioners are worried that models are *too big* to be realistically trained and deployed outside a few well-funded organizations [40]. Today, it seems more acceptable that the *entire* ML workflow, from data collection to model serving, is

[5]The median 2021 software professional's hourly pay is \$52 [216], approximately the cost of training ResNet-50.

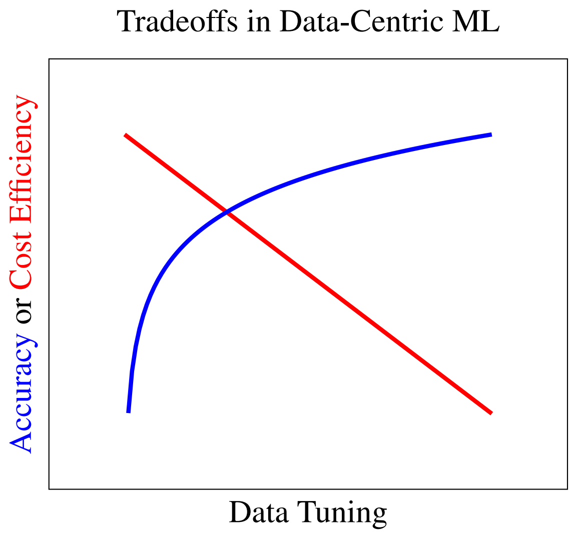


Figure 1.3: Data-centric computing brings a new dimension to creating Pareto optimal configurations—the effort required to tune and configure data-related systems. As the training or testing dataset is being tuned more, the model or tests on the model may become more accurate. However, the trade-off is that additional tuning may incur the cost of practitioner or system effort. Navigating these trade-offs is a new challenge for up-and-coming systems.

important, and large models are *not* a silver bullet. The reader should thus keep in mind that in the time it took to perform the research in this thesis, the narrative has shifted from one which was completely orthogonal to the thesis to one which is more favorable toward its message.

1.3.2 The New Metric: Data Tuning

From a systems point of view, data-centric machine learning adds a new metric: *the effort used to tune the dataset*. In concrete terms, data is no longer fixed and thus the only constraint in determining the optimal configuration is that it performs well on the provided task. With this new degree of freedom, practitioners and system designers are interested in the set of Pareto optimal configurations, which outperform other configurations on some metric *while varying the dataset*. Systems that efficiently traverse this new set of Pareto optimal configurations will be positioned to outperform what was state-of-the-art when data was fixed. The intuition for these trade-offs is shown in Figure 1.3. The curves reflect what one may encounter in practice: empirically, accuracy often follows power-laws of dataset size [145], and cost is a linear function of dataset size if the optimization procedure is kept similar (e.g., the number of epochs is fixed). The primary challenge faced in this thesis is building data systems and methods that reduce the effort required to train and test machine learning models.

1.4 Thesis Overview

The main argument of this thesis is that:

Thesis Statement: *Machine learning systems can improve programmer productivity and system efficiency by making datasets a first-class optimization target. Automated data tuning can increase traditional model training efficiency while abstractions for inference may help control for uncertainties in large pretrained model inference.*

While ML systems have traditionally focused on ensuring that it is easy and efficient to program the model fabric, they have placed little emphasis on understanding training or validation data. Modern frameworks tend to treat training and validation data as black-box placeholders that are materialized during the execution of the program. Meanwhile, efforts in data cleaning (traditionally a database problem), do not focus on performance and have strict notions of correctness. This thesis claims that there are potentially significant gains in understanding and exploiting structure that is present within the training and validation set values—the statistical distribution of values in the training and validation sets are important. Understanding the data allows the system and the user to make more informed decisions about configurations in the space of accuracy and cost.

1.4.1 Contributions

Our novel contributions are as follows and correspond to the outline of the thesis:

- **An Industrial Case-Study of Data Bottlenecks.** We demonstrate *evidence of performance issues* relating to data pipelines over real-world industrial jobs (§2). We find that, in practice, 1–10% of ML jobs are bottlenecked waiting for data at any point in time. We find that a significant fraction of bottlenecked jobs are not utilizing resources allocated to the data pipeline, indicating that they are potentially misconfigured or I/O bound. The work was published in [159].
- **A Method for Subsampling Augmentations.** We provide a methodology for *automatically subsampling augmentations* (§3). While it is standard practice to augment all training data, it is not necessarily efficient as augmentations may themselves involve extensive computation or manual verification. We find that 10% of the augmentation set can achieve 99.86% of the full augmentation performance by revisiting a classical method in the context of neural networks. The work was published in [157].
- **A Progressive I/O Image Format.** We provide a methodology for *automatically tuning image fidelity* along with a *progressive on-disk image format* (§4). We observe that I/O bandwidth in image classification is a precious resource, which motivates reducing image fidelity at the cost of task accuracy. We find that half the image fidelity is sufficient to retain accuracy for many tasks, enabling $2\times$ speedups for I/O bound workloads. We additionally develop static and dynamic methods for tuning the quality automatically as a function of the model, dataset, and task. The work was published in [158].
- **An Automatic Pipeline Tuning Framework.** We provide a general *system and framework for measuring and optimizing input pipeline throughput* (§5). We develop an interpretable modeling framework for CPU, disk I/O, and in-memory caching, mapping a set of input-pipeline statistics to expected input-pipeline performance. We implement this cost model along with the statistics collection in a tool, `Plumber`, which traces and tunes pipelines

with one line of code. We find that `Plumber` can achieve $47\times$ speedups compared to misconfigured pipelines and outperforms state-of-the-art tunes by up to 50%. The work was published in [159].

- **A Regular Expression Query Interface for LLMs.** We provide a *system for searching and validating large language models* (§6). We built a system, `ReLM`, which allows running standard regular expression queries on large language models. `ReLM` achieves a $15\times$ speedup or $2.5\times$ higher data efficiency over traditional approaches in memorization and toxicity finding. Additionally, `ReLM` facilitates novel tests for bias and easily recovers prompt tuning behavior necessary for state-of-the-art zero-shot performance. This work was published in [160].

Finally, we reflect on some of the lessons learned and future directions in §7.

1.4.2 Code

The code repositories to go along with this work are found at:

1. `https://github.com/mkuchnik/Efficient_Augmentation`
2. `https://github.com/mkuchnik/PCR_Release`
3. `https://github.com/mkuchnik/PlumberApp`
4. `https://github.com/mkuchnik/relm`

Chapter 2

Background and Fleetwide Analysis

This chapter outlines key background material relevant to the thesis, with a primary focus on training data pipelines, or simply *input pipelines*. While the presentation focuses on `TensorFlow`'s `tf.data`, the concepts discussed are general and can exist in other implementations. For instance, `PyTorch` has started supporting a functional dataloading library of its own [24], thus demonstrating that the functional abstraction discussed in this section are generally applicable to multiple frameworks. The programming of input pipelines is discussed throughout §2.1. We then shift to an empirical study of input pipelines in §2.2, highlighting the impact data pipelines have in industrial settings. Well-configured input pipelines are key to efficient datacenter performance, because a misconfiguration can create disastrous bottlenecks for its workload. Similarly, a machine should be sized to fit its input pipeline workload to avoid bottlenecks. This chapter looks into the divide between these two types of bottlenecks. In doing so, it motivates tools that characterize and tune input pipeline performance as well as methods that reduce CPU and I/O resource usage.

2.1 What are Input Pipelines?

All ML training begins with input data, which is curated by input pipeline frameworks. In this section, we outline the abstractions provided by input pipeline frameworks, noting design decisions that have an effect on understanding performance (§2.1.1). We next jump into common tools for understanding bottlenecks (§2.1.2).

2.1.1 Input Pipeline Architecture

Input pipelines specify: a data source, transformation functions, iteration orders, and grouping strategies. For example, image classification pipelines read, decode, shuffle, and batch their `(image, label)` tuples, called *training examples*, into fixed-size arrays [71, 156]. Unlike batch processing frameworks (e.g., Spark [23], Beam [2], Flume [4]), which may be used to *create* the data used for training, the main goal of input pipeline frameworks is to dynamically *alter* the training set online. Three major reasons for online processing are: 1) data is stored compressed in order to conserve storage space, 2) data is randomly altered online using data augmentations, and 3) practitioners may experiment with features throughout modeling.

```
1 model = model_function() # Initialize model
2 ds = dataset_from_files().repeat()
3 ds = ds.map(parse).map(crop).map(transpose)
4 ds = ds.shuffle(1024).batch(128).prefetch(10)
5 for image, label in ds:  # Iterator Next()
6     model.step(image, label) # Grad Update
```

Figure 2.1: `Python` pseudo-code for ImageNet-style training. Line 2 is file reading, line 3 is user-defined image processing, and line 4 samples, batches, and prefetches data. Lines 5–6 are the critical path of training, instantiating an `Iterator`.

Input pipelines are programmed *imperatively* or *declaratively*, each with different APIs. Imperative frameworks, like PyTorch's and MxNet's [58, 212] `DataLoader`, allow users to specify their pipelines in plain `Python` by overriding the `DataLoader`. Declarative libraries, like DALI [115] and `TensorFlow`'s `tf.data` [211, 272], compose functional, library-implemented primitives, which are executed by the library's runtime. While both styles are equally expressive in terms of pipeline construction, frameworks leave a large part of implementation to the user. In contrast, the libraries decouple specification from implementation, requiring the user to merely declare the pipeline structure, offloading optimizations to the runtime. We focus our discussion on `tf.data`, because it allows for various backends to service similar pipeline definitions, enabling tracing and tuning *behind the API*, and can be used with all major training ML frameworks.

Input Pipeline Abstractions. In `tf.data`, `Datasets` are the basic building blocks for declaring input pipelines. In Figure 2.1, each function call chains `Datasets`. Instantiating a `Dataset` (line 5) yields a tree composed of one or more `Iterators`, which produces a sequence of training examples through an iterator interface that maintains the current position within a `Dataset`. Figure 2.2 illustrates how `Datasets` are unrolled into an `Iterator` tree. Some `Datasets` (see `Map`) implement multi-threaded parallelism within the corresponding `Iterator`, while others (see `TFRecord`) can only be parallelized by reading from multiple sources in parallel (e.g., using `Interleave`). An `Iterator` implements the following three standard Iterator model [111, 187] methods:

- **Open** defines `Iterator` parameters and references to child `Iterators` and initializes internal state.
- **Next** yields an example from the `Iterator` or a signal to end the stream. Source nodes read from storage or memory to yield examples. Internal nodes call `Next` on their children to gather examples before applying transformations to them. In Figure 2.1, the data source outputs file contents, which then undergo image processing, shuffling, and batching.
- **Close** releases resources and terminates the `Iterator`.

User-Defined Functions. User-defined functions (UDFs) comprise the bulk of data pipeline execution time and are used to implement custom data transformations. Figure 2.1 shows an example of a computer-vision pipeline utilizing UDFs, which perform image decoding, image preprocessing, and tensor transposing for efficient execution on accelerators. Users are able to write UDFs in a restricted form of `Python`, which is compiled into an efficient and parallel implementation.

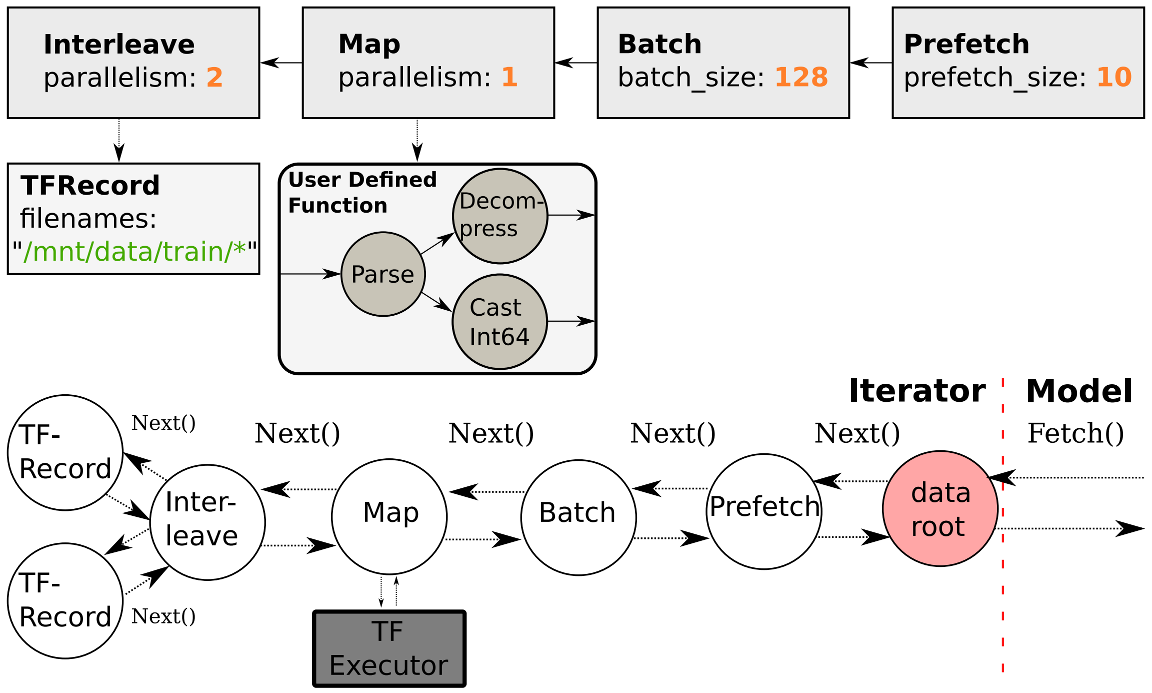


Figure 2.2: **Top (`Dataset` View):** A `tf.data` pipeline is composed of `Dataset` objects characterized by attributes, e.g., by their level of parallelism. **Bottom (`Iterator` View):** The root `Dataset` is instantiated into an `Iterator` tree at runtime, which feeds the model. `Iterators` pull data from their children in a recursive manner.

2.1.2 Understanding Input Bottlenecks

An input bottleneck occurs when the input pipeline is not able to generate batches of training examples as fast as the training computation can consume them. If the time spent waiting for the input pipeline exceeds tens of microseconds on average, the input pipeline is not keeping up with model training, causing a *data stall* [206] The current practice of pipeline tuning, which optimizes the throughput (rate) of the pipeline, is explained below.

Profilers. Event-based profilers, such as the `TensorFlow` Profiler [271], can emit metadata at particular software events to aid in determining control-flow. As there are *thousands of concurrent events per second* for a pipeline, it is difficult to quantitatively determine which events actually *caused* a throughput slowdown. To automate and generalize past heuristics deployed in guides [273], the `TensorFlow` Profiler added a bottleneck discovery feature [274]. This tool works by finding the `Iterator` with highest impact on the *critical path* of a `Dataset`. However, it can only rank `Datasets` by slowness, and it can't predict their effect on performance. Furthermore, critical-paths are not well-defined for concurrent and randomized event graphs (e.g., execution times of individual operations overlap and are data-dependent), forcing heuristics to be used.

Tuners. `tf.data` applies dynamic optimization of pipeline parameters when users specify `AUTOTUNE` for supported parameters, such as the degree of parallelism and size of prefetch buffers [211]. The autotuning algorithm works by representing `Iterators` in a pipeline as an M/M/1/k queue [169, 260], and a formulation for the queue's latency is analytically determined. Statistics about the execution of `Iterators` are recorded with a lightweight harness. Combining the analytical model with the runtime statistics enables tuning the latency of the pipeline with respect to performance parameters. Specifically, the processing time of each element is normalized by the parallelism and the ratio of input to output elements. This statistic is then combined with "input latency" statistics of the children nodes in a node-type dependent way to get an "output latency". Output latency tuning is done via hill-climbing or gradient descent and ends when the tuning plateaus or reaches a resource budget. While `AUTOTUNE` works in practice, it is hard to understand and extend for two reasons. First, open-systems, like M/M/1/k queues, have a throughput purely dependent on input arrival rates, which are not applicable for closed-systems. Second, because resource utilization is not modeled, the output latency function can be driven to zero if parallelism is allowed to increase unbounded, forcing heuristic constraints to be used.

2.2 Fleetwide Analysis

We analyzed over two million ML jobs that ran in Google datacenters to determine whether input bottlenecks are common and characterized their most typical root cause. The jobs we analyzed used `tf.data` and ran over a one-month period from July to August of 2020. The workload included production and research jobs from a variety of ML domains, including image recognition, natural language processing, and reinforcement learning. To measure the frequency of input bottlenecks in practice, we measured the average time spent fetching input data per training step across jobs.

2.2.1 Are Input Bottlenecks Common?

We detect input pipeline bottlenecks by measuring the average latency across all `Iterator Next` calls, which is the average time the job spends blocked waiting for input data in each training step.

Observation 1: *For a significant fraction of ML jobs, the input data pipeline produces data at a slower rate than the model is able to consume it.*

Figure 2.3 shows that for 62% of jobs, the average `Next` latency per training step exceeds 1ms and for 16% of jobs, the average wait time exceeds 100ms. Since the input pipeline affects both end-to-end training performance and hardware accelerator utilization, it is a critical part of ML training to optimize. We note that fetch latency applies to *each iteration* of the training process—over ten thousand times in a typical session. At any point in time, between 1–10% of the fleet is waiting on input data, which is significantly costly at the scale the fleet operates at.

2.2.2 Why Do Input Bottlenecks Occur?

We classify input pipeline bottlenecks into two categories: hardware and software bottlenecks. *Hardware bottlenecks* occur when hardware resources used for data processing saturate. The hardware resources typically used for input processing are CPU cores, host memory, and local or remote storage. Many workloads do not use local storage for I/O, but pull their data from external data sources (e.g., distributed filesystems) [211]. Thus, external I/O resources may bottleneck

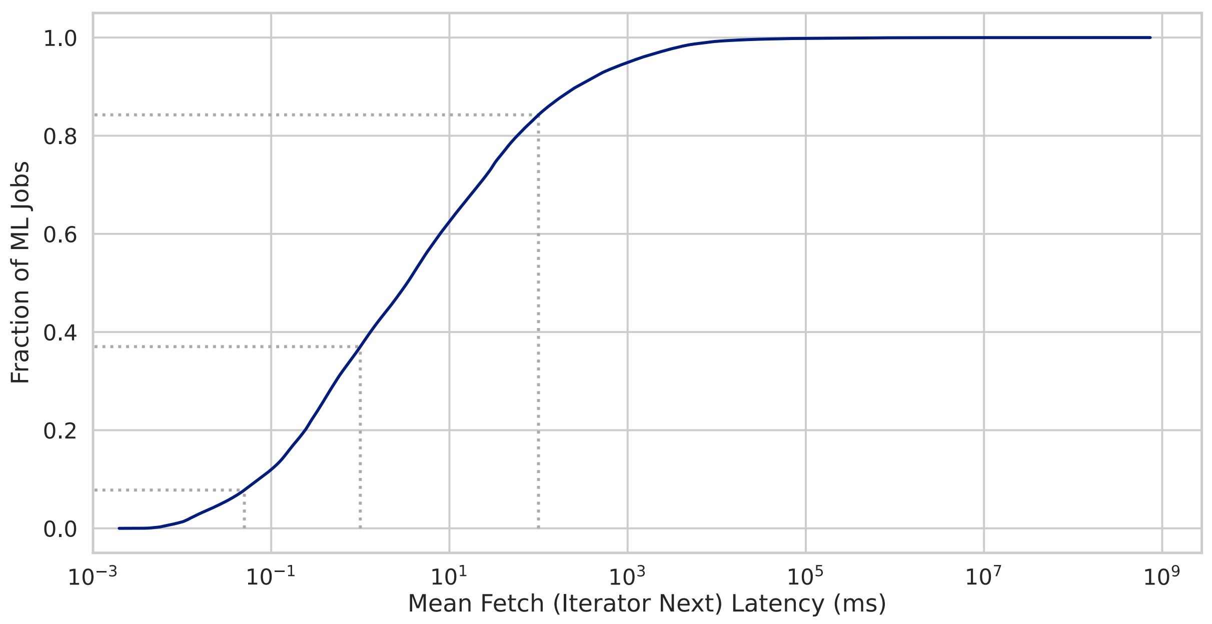


Figure 2.3: Time spent fetching training examples using `Next`, in ms. On average, for 92% of jobs `Next` latency exceeds 50μs, for 62% of jobs it exceeds 1ms, and for 16% of jobs it exceeds 100ms. Fetch latencies for well-configured pipelines are in the low tens of *microseconds*.

training jobs. Hardware resource saturation can be remedied by adjusting the resource allocation, e.g., switching to a node with more cores, memory, or I/O bandwidth.

Software bottlenecks occur because the software is not driving the hardware efficiently, e.g., by using too little or too much parallelism, or incorrectly sizing prefetch buffers to overlap communication and computation. When a user is confronted with a software bottleneck, they must find the root cause and fix it; otherwise, they risk underutilizing hardware performance. We note I/O bottlenecks can also be caused by software configuration, due to inefficient access patterns and low read parallelism.

Observation 2: *Host hardware is rarely fully utilized for jobs with high input pipeline latencies. Thus, input bottlenecks are likely rooted in software or I/O inefficiencies, rather than hardware saturation.*

To understand the breakdown between these categories of input bottlenecks, we measure the host CPU and memory bandwidth resource utilization for the jobs captured in our analysis. In Figure 2.4, we show a breakdown of different jobs' average `Next` call latencies organized according to the CPU and memory bandwidth utilization of the pipeline host. We exclude jobs with latency below 50μs because for those jobs the input pipeline is not a bottleneck, as it takes tens of microseconds to read input data that is readily available from a prefetch buffer (including thread wakeup and function invocation time). Our data indicates that jobs with latency between 50μs and 100ms (small, dark dots) utilize *more* of the host's resources than those with latency higher than 100ms (large, blue dots). For context, a TPUv3-8 [140, 203] takes roughly 120ms to process a minibatch for ResNet-50 [123]. Jobs where the average fetch latency exceeds 100ms are, then, *significantly* input-bound, and as shown in the Figure 2.4 their resource usage for both memory bandwidth and CPU is concentrated below 20%.

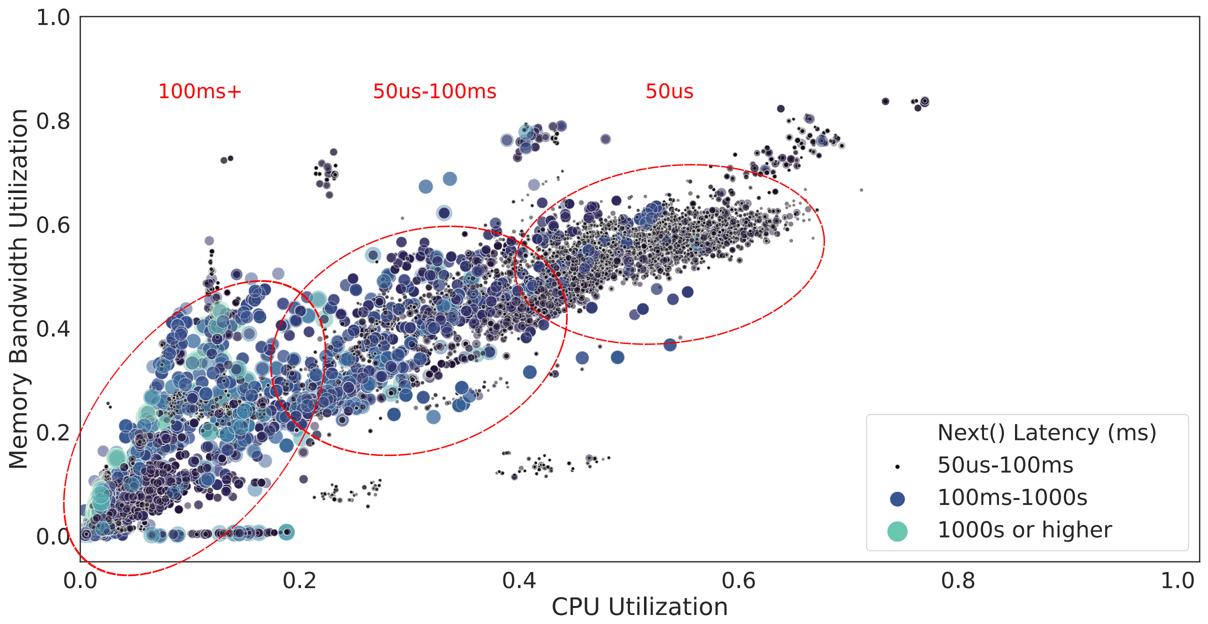


Figure 2.4: CPU utilization of training jobs compared to their memory bandwidth utilization. Larger points are for jobs with longer pipeline latency. We annotate three major clusters. The average CPU and memory-bandwidth usage is 11% and 18%, respectively, for jobs with pipeline latency of 100ms or more. The majority of jobs do not saturate host resources, suggesting bottlenecks in software.

2.3 Related Work in Input Pipeline Bottlenecks

The analysis in this section is just a snapshot of how impactful input pipelines can be in practice. Recent studies have analyzed ML workloads and found that input pipelines can become bottlenecks [206, 211]. This line of work is further supported by empirical evidence of bottlenecks in recommendation models [322]. Workloads for end-to-end workflow managers, such as TFX [32], see that many models are in fact shallow models and that these pipelines can be extremely complex [308]. Thus, on top of the analysis in this section, there is growing evidence that input pipelines should be accepted as a key aspect of machine learning performance.

2.4 Discussion

Training ML models requires reading and transforming large quantities of unstructured data. Our analysis of real ML training jobs at Google shows that data pipelines do not adequately utilize their allocated hardware resources, leading to potential bottlenecks, which waste precious accelerator resources. The analysis additionally indicates that many jobs are misconfigured or suffer from excessive I/O, motivating work on I/O reduction (§4) and pipeline tuning (§5). Overall, this chapter shows that the training input pipeline has a key role in datacenter performance in practice and should be optimized like any other part of the machine learning system stack.

Chapter 3

Selective Data Preparation and Augmentation

This chapter focuses on data augmentation and how to reduce the cost associated with the augmentations by subsampling them. Data augmentation is commonly used to encode invariances in learning methods and is a key transformation in input pipelines. However, this process is often performed in an inefficient manner, as artificial examples are created by applying a number of transformations to all points in the training set. The resulting explosion of the dataset size can be an issue in terms of storage and training costs, as well as in selecting and tuning the optimal set of transformations to apply. We demonstrate that it is possible to significantly reduce the number of data points included in data augmentation while realizing the same accuracy and invariance benefits of augmenting the entire dataset. We propose a novel set of subsampling policies, based on model influence and loss, that can achieve a 90% reduction in augmentation set size while maintaining the accuracy gains of standard data augmentation. In turn, systems can choose to drop augmentations in order to: 1) reduce the amount of online processing or 2) allow the augmented set to be cached in memory.

3.1 Introduction

Data augmentation is a process in which the training set is expanded by applying class-preserving transformations, such as rotations or crops for images, to the original data points. This process has become an instrumental tool in achieving state-of-the-art accuracy in modern machine learning pipelines. Indeed, for problems in image recognition, data augmentation is a key component in achieving nearly all state-of-the-art results [57, 79, 112, 253]. Data augmentation is also a popular technique because of its simplicity, particularly in deep learning applications, where applying a set of known invariances to the data is often more straightforward than trying to encode this knowledge directly in the model architecture.

However, data augmentation can be an expensive process, as applying a number of transformations to the entire dataset may increase the overall size of the dataset by orders of magnitude. For example, if applying just 3 sets of augmentations (e.g., translate, rotate, crop), each with 4 possible configurations, the dataset can easily grow by a factor of 12 (if applied independently), all the way to $64\times$ (if applied in sequence). While this may have some benefits in terms of overfitting, augmenting the entire training set can also significantly increase data storage costs and training time, which can scale linearly or superlinearly with respect to the training set size. Further, selecting the optimal set of transformations to apply to a given data point is often a non-trivial task. Applying transformations not only takes processing time, but also frequently requires some amount of domain expertise. Augmentations are often applied heuristically in practice, and small perturbations are expected (but not proven) to preserve classes. If more complex augmentations are applied to a dataset, they may have to be verified on a per-sample basis.

We aim to make data augmentation more efficient and user-friendly by identifying subsamples of the full dataset that are good candidates for augmentation. In developing policies for subsampling the data, we draw inspiration from the Virtual Support Vector (VSV) method, which has been used for this purpose in the context of support vector machines (SVMs) [44, 70]. The VSV method attempts to create a more robust decision surface by augmenting only the samples that are close to the margin—i.e., the support vectors. The motivation is intuitive: if a point does not affect the margin, then any small perturbation of that point in data space will likely yield a point that is again too far from the margin to affect it. The method proceeds by applying class-preserving data augmentations (e.g., small perturbations) to all support vectors in the training set. The SVM is then retrained on the support vector dataset concatenated with the augmented dataset, and the end result is a decision surface that has been encoded with transformation invariance while augmenting many fewer samples than found in the full training set.

Although proven to be an effective approach for SVMs, methods utilizing support vectors may not generalize well to other classifiers. We aim to develop policies that can effectively reduce the augmentation set size while applying to a much broader class of models. A key step in developing these policies is to determine some metric by which to rank the importance of data points for augmentation. We build policies based on two key metrics. First, we make a natural generalization of the VSV method by measuring the loss induced by a training point. Second, we explore using the *influence* of a point as an indicator of augmentation potential. Influence functions, originating from robust statistics, utilize more information than loss (i.e., residuals) alone, as they take into account both leverage and residual information.

The contributions of this chapter are as follows. First, we demonstrate that it is typically unnecessary to augment the entire dataset to achieve high accuracy—for example, we can maintain 99.86% or more of the full augmentation accuracy while only augmenting 10% of the dataset in the case of translation augmentations, and we observe similar behavior for other augmentations. Second, we propose several policies to select the subset of points to augment. Our results indicate that policies based off of training loss or model influence are an effective strategy over simple baselines, such as random sampling. Finally, we propose several modifications to these approaches, such as sample reweighting and online learning, that can further improve performance. Our proposed policies are simple and straightforward to implement, requiring only a few lines of code. We perform experiments throughout on common benchmark datasets, such as MNIST [171], CIFAR10 [155], and NORB [172].

3.2 Related Work

In the domain of image classification, most state-of-the-art pipelines use some form of data augmentation [57, 79, 112, 253]. This typically consists of applying crops, flips, or small affine transformations to all the data points in the training set, with parameters drawn randomly from hand-tuned ranges. Beyond image classification, various studies have applied data augmentation techniques to modalities such as audio [282] and text [188]. The selection of these augmentation strategies can have large performance impacts, and thus can require extensive selection and tuning [240].

Motivated by the ubiquity of data augmentation and the difficulty in selecting augmentations, there has been a significant amount of work in selecting and tuning the best *transformations* to use when performing augmentation. For example, Fawzi et al. [90] use adaptive data augmentation to choose transformations that maximize loss for the classifier; Ratner et al. [240] propose learning a sequence model of composed transformations; and Cubuk et al. [61] suggest a reinforcement learning approach. In contrast to these works, our aim is instead to select which *data points* to augment while holding transformations fixed. Our subsampling policies are therefore complementary to many of the described approaches, and in fact, could be quite beneficial for approaches such as reinforcement learning that can quickly become infeasible for large datasets and transformation spaces. Finally, we note that several recent works have proposed augmentation strategies based on adversarial training approaches, such as robust optimization frameworks or generative adversarial networks (GANs) [19, 107, 287]. These approaches generate artificial points from some target distribution, rather than by directly transforming the original training points. We view these works as orthogonal and complementary approaches to the proposed work, which is designed in concert with more traditional data augmentation strategies.

The area of work most closely related to our own is that of the Virtual Support Vector (VSV) method [44, 70]. This method was proposed in the support vector machine literature as a way to reduce the set of points for augmentation by limiting transformations to only support vectors. In the context of SVMs, the motivation is straightforward, as points that are far from the margin are unlikely to affect future models if they are transformed via small perturbations. However, to the best of our knowledge, there has been no work extending these ideas to methods beyond SVMs, where the notion of support vectors is not directly applicable.

Inspired by the VSV work, we similarly seek ways to downsample the set of candidate points for augmentation, though through metrics beyond support vectors. We begin by generalizing the notion of a support vector by simply measuring the loss at each training point[1]. We also explore model influence, which has been rigorously studied in the field of robust statistics as a way to determine which data points are most impactful on the model. Model influence has been studied extensively in the regression literature [59, 126, 232, 288], and more recently, in non-differentiable (SVMs) and non-convex (deep networks) settings [153]. We provide additional details on these metrics in Section 3.4.

Finally, we note that this work is closely related to work in subsampling for general dataset reduction (i.e., not in the context of data augmentation). For example, works using gradients

[1]We also investigate a more direct generalization of the VSV method—sampling points according to their distance from the margin— although this method generally underperforms the other metrics.

[325], leverage [81, 82, 190], and influence functions [196, 277, 292] have shown better results than uniform sampling of data samples in the original dataset. Our scenario differs from the subsampling scenarios in these works as we ultimately anticipate increasing the size of the dataset through augmentation, rather than decreasing it as is the case with subsampling. Subsampling methods are motivated by being unable to train models on entire datasets due to the datasets being too large. Our motivation is instead that the full *augmented dataset* may be too large, but the original training set is sufficiently small to be handled without special consideration. We therefore assume it is possible to obtain information (e.g., influence, loss, etc.) by fitting a model to the original data. Further, the interpretation of our scenario differs, as the subsampling is performed with the ultimate aim being to retain the accuracy of some yet-to-be-determined fully augmented dataset, as opposed to the original dataset.

3.3 Motivation: On the Effectiveness of Subsampling

In this work, we seek to make data augmentation more efficient by providing effective policies for subsampling the original training dataset. To motivate the effect of subsampling prior to augmentation, we begin with a simple example. In Table 3.1, we report the effect that performing translation augmentations has on the final test accuracy for several datasets (MNIST, CIFAR10, NORB). In the second column, we provide the final test accuracy assuming *none* of the training data points are augmented, and in the last column, the final test accuracy after augmenting *all* of the training data points (i.e., our desired test accuracy). Note that the test dataset in these examples has also been augmented with translation to better highlight the effect of augmentation; we provide full experimental details in Section 3.5. In columns 3–8, we report test accuracies from augmenting 5, 10, and 25 percent of the data, where these subsamples are either derived using simple random sampling or via our proposed policies (to be discussed in Section 3.4).

An immediate take-away from these results is that, even in the case of simple random sampling, it is often unnecessary to augment the entire dataset to achieve decent accuracy gains. For example, augmenting just 25% of the dataset selected at random can yield more than half of the total accuracy gain from full augmentation. However, it is also evident that subsampling can be done more effectively with the appropriate policy. Indeed, as compared to random sampling, when augmenting just 10% of the data, these optimal policies can achieve almost identical results to full augmentation (within $.1\%$ for CIFAR10 and higher accuracy than full augmentation for MNIST and NORB). These results aim to serve as a starting point for the remaining chapter. We describe our proposed policies in detail in Section 3.4, and we provide full experiments and experimental details in Section 3.5.

3.4 Augmentation Set Selection Policies

In this section, we provide details on our augmentation policies, including their general structure (described below), the metrics they utilize (Section 3.4.1), and improvements such as reweighting or online learning (Section 3.4.2).

Setup. The aim in this work is to find some subset $\mathcal{S} := \{(\boldsymbol{x}_i, y_i), \ldots, (\boldsymbol{x}_j, y_j)\}$ of the

Dataset	No Aug.	Baseline Random Policy			Best Policy			Full Aug.
	0%	5%	10%	25%	5%	10%	25%	100%
MNIST	93.2%	99.0%	99.3%	99.5%	99.7%	99.8%	99.7%	99.6%
CIFAR10	96.3%	96.6%	96.8%	97.0%	97.0%	97.2%	97.3%	97.3%
NORB	87.3%	88.0%	88.3%	88.4%	89.9%	89.8%	89.7%	89.7%

Table 3.1: Best observed policy vs. expected baseline with translate augmentations for various percentages of the training set being augmented. The best policies are capable of reaching near full augmentation performance with a small augmentation budget.

full training set $\mathcal{D} := \{(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_n, y_n)\}$[2], such that augmenting only the subset $\mathcal{S}$ results in similar performance to augmenting the entire dataset $\mathcal{D}$. More precisely, the goal is to minimize the size of $\mathcal{S}$, $|\mathcal{S}|$, subject to the constraint that perf$(\mathcal{S}_{aug}) \approx$ perf$(\mathcal{D}_{aug})$, where $\mathcal{S}_{aug}$ and $\mathcal{D}_{aug}$ represent the dataset after appending augmented examples generated from the original examples in $\mathcal{S}$ or $\mathcal{D}$, respectively. We note that while the performance measure perf$(\cdot)$ may be broadly construed, we specifically focus on measuring performance based on test accuracy in our experiments.

General Policies. Our proposed policies consist of two parts: (i) an *augmentation score* which maps each training point $(\boldsymbol{x}_i, y_i)$ to a value $s \in \mathbb{R}$, and (ii) a *policy* by which to sample points based on these augmentation scores. In Section 3.4.1, we describe two metrics, loss and model influence, by which augmentation scores are generated. In terms of policies for subset selection based on these scores, we first explore two simple policies—deterministic and random. In particular, given a set of augmentation scores $\{s_1, \ldots, s_n\}$ for the n training points, we select a subset $\mathcal{S} \subseteq \mathcal{D}$ either by ordering the points based on their scores and taking the top k values (in a deterministic fashion), or by converting each augmentation score s_i to a probability $\pi_i(\boldsymbol{z}) \in [0, 1]$, where $\boldsymbol{z}_i := (\boldsymbol{x}_i, y_i)$, and then sampling according to this distribution without replacement. As the augmentation scores (and resulting policies) may be affected by updates to the model after each augmentation, we additionally explore in Section 3.4.2 the effect of iteratively updating or re-weighting scores to adjust for shifts in the underlying model. A non-exhaustive overview of the various augmentation policies is provided in Table 3.2.

3.4.1 Metrics: Loss and Influence

We propose two metrics to determine our augmentation scores: training loss and model influence.

Training loss. One method to obtain augmentation scores is the loss at a point in the training set. This can be viewed as a more direct generalization of the Virtual Support Vector (VSV) method, as support vectors are points with non-zero loss. However, studying loss directly will allow us: (i) to extend to methods beyond SVMs, and (ii) to expand the augmented set to data points beyond just the fixed set of support vectors.

Influence. We also explore policies based on Leave-One-Out (LOO) influence, which measures the influence that a training data point has against its own loss when it is removed from the

[2]Note that we assume classification tasks, thus y is a scalar.

Policy Type	Selection Function	Update Scores	Downweight Points
Baseline	$P(\boldsymbol{z}_i) = \frac{1}{n}$	X	X
Random Prop.	$P(\boldsymbol{z}_i) = \frac{s_i}{\sum_j s_j}$	X	X
Deterministic Prop.	$\text{Rank}(\boldsymbol{z}_i) = \texttt{SELECT}_S^{-1}(s_i)$	X	X
Random Prop. Update	$P(\boldsymbol{z}_i) = \frac{s_i}{\sum_j s_j}$	✓	X
Rand. Prop. Downweight	$P(\boldsymbol{z}_i) = \frac{s_i}{\sum_j s_j}$	X	✓

Table 3.2: Overview of the augmentation policies and their parameters, where s_i is the augmentation score given to point $\boldsymbol{z}_i := (\boldsymbol{x}_i, y_i)$. The $\texttt{SELECT}_S^{-1}$ function corresponds to the inverse of an order statistic function. As a baseline, we compare to sampling data points at random, ignoring the augmentation scores. Note that the notation here is simplified to allow sampling with replacement, though in practice we perform sampling without replacement.

training set. We follow the notation used in Koh and Liang [153]. Let $\hat{\boldsymbol{\theta}}$ be the minimizer of the loss, which is assumed to be twice differentiable and strictly convex in $\boldsymbol{\theta}$. Let $\boldsymbol{H}_{\hat{\boldsymbol{\theta}}}$ be the Hessian of the loss with respect to $\boldsymbol{\theta}$ evaluated at the minimizer. We define the influence of upweighting a point, $\boldsymbol{z}$, on the loss at a test point, $\boldsymbol{z}_{\text{test}}$, as $\mathcal{I}_{\text{up,loss}}(\boldsymbol{z}, \boldsymbol{z}_{\text{test}}) := -\nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{z}_{\text{test}}, \hat{\boldsymbol{\theta}})^\top \boldsymbol{H}_{\hat{\boldsymbol{\theta}}}^{-1} \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{z}, \hat{\boldsymbol{\theta}})$. It follows that if the test point is $\boldsymbol{z}$, then the LOO influence can be calculated as:

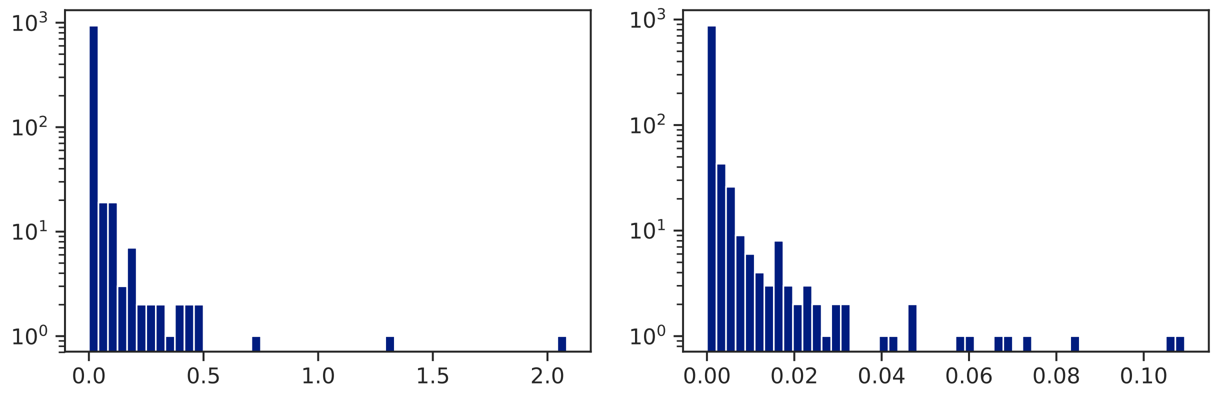
$$\mathcal{I}_{\text{LOO}}(\boldsymbol{z}) := \mathcal{I}_{\text{up,loss}}(\boldsymbol{z}, \boldsymbol{z}) = -\nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{z}, \hat{\boldsymbol{\theta}})^\top \boldsymbol{H}_{\hat{\boldsymbol{\theta}}}^{-1} \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{z}, \hat{\boldsymbol{\theta}})\,. \tag{3.1}$$

For our augmentation scores, we care only about the magnitude of the LOO influence, so it can be assumed that the sign is dropped.

To understand the potential of using training loss and model influence for scoring, we provide a histogram of model influence across the CIFAR10 and NORB datasets in Figure 3.1. In Figure 3.1, we see that while most of the mass is centered around 0 (which we utilize to avoid points), there is sufficient variability to allow for ranking points by preference. Further, as seen in Figure 3.2, these values are correlated before and after augmentation, indicating that these metrics are a reliable measure of the future impact of a data point after augmentation. We observe Spearman's rank correlations [264] between 0.5 and 0.97 with p-values less than 0.001 (and usually orders of magnitude less).

3.4.2 Refinements: Sample Reweighting and Score Updating

Reweighting. To motivate reweighting individual samples, consider an augmentation which is the identity map: $f_T : \boldsymbol{z}_i \to \{\boldsymbol{z}_i\}$. Since we add augmentations back to the training set, our augmentation policy will duplicate selected samples, resulting in a net effect which reweights samples with twice the original weight. Using transformations that result in larger augmentation sets will result in larger weights. One approach is post-processing; Fithian and Hastie [93], for example, show that the case of class imbalanced sampling can be corrected with a shift of the logistic regressor's bias. To normalize for the effect of this implicit reweighting, we divide the weights of the original samples and its augmented samples by the size of that set, $|f_T(\boldsymbol{z}_i)|$. Under



(a) CIFAR10 (b) NORB

Figure 3.1: Distribution of influence function values on initial training set for translate augmentations. Most values are not influential and can therefore be augmented with low priority. We find similar results when measuring training loss.

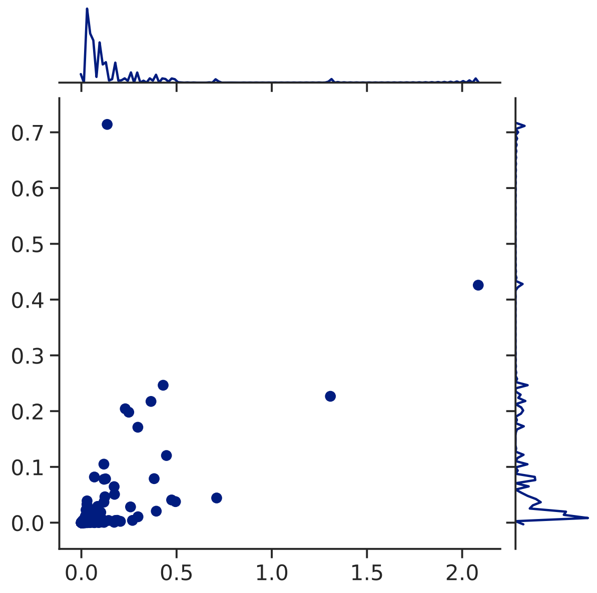
this scheme, we guarantee that we conserve the weight originally assigned to a point (and conserve the ratios of labels). More sophisticated policies, such as reweighting samples by a measure of how trustworthy they are (e.g., perhaps some bounds can be derived on the label-preserving properties of an augmentation), remain to be investigated as future work.

We find that in many cases, the performance of reweighting is similar in expectation to the base case. However, in some cases, reweighting can have a negative impact, as we discuss in Section 3.5.2. We expect this policy to be more useful in the case of class imbalance, where the duplication of minority class samples may significantly alter the distribution over classes.

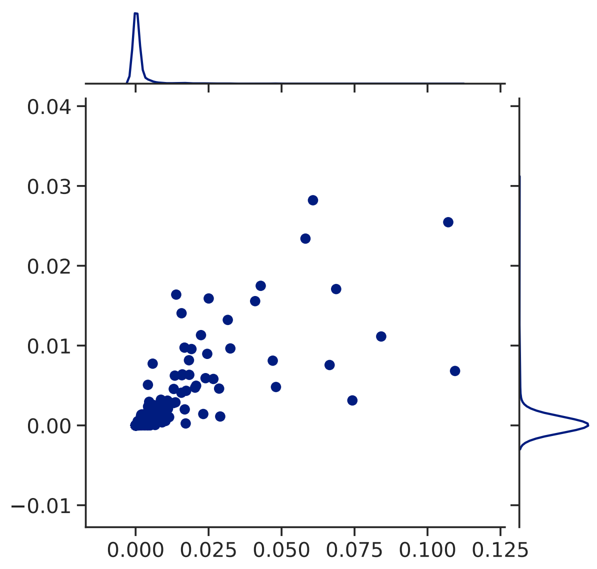
Updating scores. Once we decide to augment a data point, we can either continue to use the same influence information which we derived for the un-augmented dataset, or we can choose to update it. The reason for doing this is to account for the drifting model behavior as points are added to the training set and the model is retrained. However, if having a single estimate of influence for the whole lifetime of the model is sufficient, then avoiding repeated influence calculations will reduce the amount of computation required while also enabling an increased level of parallelism (e.g., minibatching, distributed computations). We find that this modification results in similar behavior to that of reweightings, where expected performance of a policy remains similar. Overall, we have not observed a significant enough effect to suggest that this technique is justified given the extra cost it requires. The benefit of this is that it implies that many applications may need to only compute selection metadata one time throughout the augmentation process.

3.5 Experiments

In this section, we provide detailed results on the performance of our proposed policies for data subsampling. For all experiments, we use a convolutional neural network (CNN) to create bottleneck features, which we then use as input into a linear logistic regression model. This is equivalent to freezing the weights of the CNN, or using a set of basis functions, $\phi_i(\cdot)$, to



(a) CIFAR10



(b) NORB

Figure 3.2: Influence distribution on initial training set (x-axis) vs. final training set (y-axis) for translate augmentations. Points that are uninfluential typically remain uninfluential.

transform the inputs [39], and allows us to quickly calculate training loss and model influence. We explore the results of our augmentation policies on three datasets: binary classification variants of MNIST, CIFAR10, and NORB. For MNIST features, we use a LeNet architecture [171] with ReLu activations, and for CIFAR10 and NORB, we use ResNet50v2 [124]. While for CIFAR10 and NORB we generate the bottleneck features once due to cost, for MNIST, we additionally study the effect of re-generating these features as new points are selected and augmented (i.e., training both the features and model from scratch throughout the experiments).

In terms of augmentations, we consider three examples: translation, rotation, and crop. To control for sources of variance in model performance, all augmentations under consideration are applied exhaustively in a deterministic fashion to any selected samples, and the resulting augmented points are then added back to the training set. Formally, given a data point, $\boldsymbol{z} := (\boldsymbol{x}, y) \in \mathbb{X} \times \mathbb{Y}$, our augmentation is a map from a data point to a finite set of data points: $f_T : \boldsymbol{z} \to \{\boldsymbol{z}_1, \ldots, \boldsymbol{z}_n : \boldsymbol{z}_i \in \mathbb{X} \times \mathbb{Y}\}$. We controlled for augmentation-induced regularization by performing a simple cross validation sweep for the regularization parameter λ each time the model was re-trained, and we found regularization to have negligible impact in the trends we observed. For all datasets and augmentations, we make the effect of augmentation more apparent by adding augmented test points to the test set. For example, in the case of translation, we test the performance of applying translation augmentations to the original training set, and then determine the accuracy using an augmented variant of the test data that has been appended with translated test examples. All augmentations are performed using Imgaug [144], and our code is written in `Python` using Keras CNN implementations.

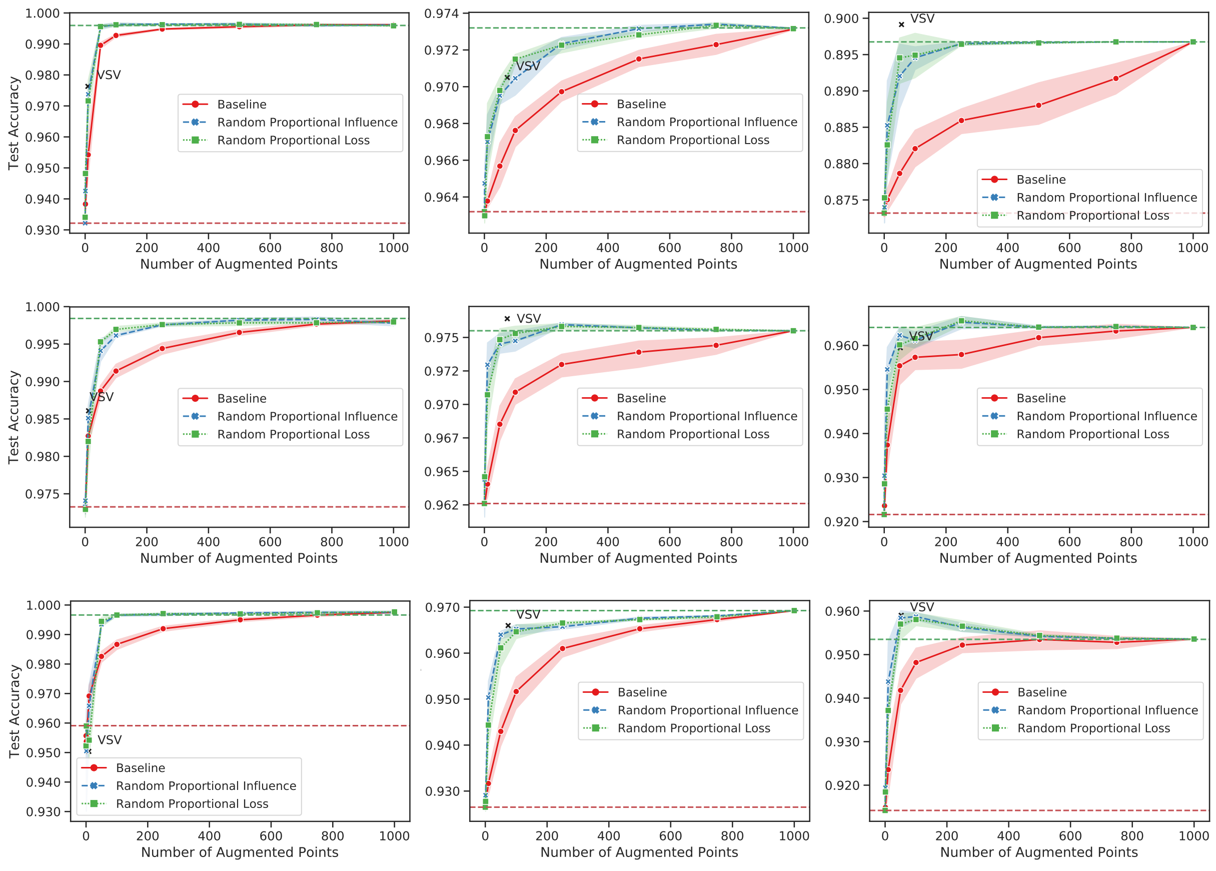

(g) MNIST-crop (h) CIFAR10-crop (i) NORB-crop

Figure 3.3: The performance of random policies using influence and loss vs. the baseline (simple random sampling). Random sampling based on loss/influence consistently outperforms the baseline.

3.5.1 General Policies: Influence and Loss

In Figure 3.3, we explore a first set of policies in which we randomly sample points for augmentation proportional either to their loss (green) or influence value (blue). To calculate the loss and influence, we incur a one-time cost of training the model on the original dataset. As a baseline (red), we compare these methods to a simple strategy in which data points for augmentation are drawn entirely at random (irrespective of loss or influence). The red-dotted horizontal line indicates the test accuracy with no augmentation, and the green-dotted line indicates the test accuracy after augmenting the entire training set. Note that all policies have the same accuracy when the number of points is 0 or k, where k is the number of points in the original training set, which correspond to the un-augmented training set and the fully augmented training set, respectively[3]. We observe similar behavior in terms of the deterministic policy.

Across the datasets and transformation types, we notice several trends. First, the policies based

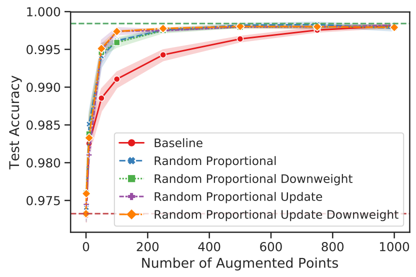
[3]In practice, the non-convexity of CNNs results in accuracies which may vary slightly.

on loss and influence consistently outperform the random baseline. This is particularly true for the rotation augmentation for all three datasets, where the random-influence and random-loss policies achieve the full augmentation accuracy with only 5–10$\%$ of the data, compared to 90–100$\%$ of the data for random sampling. Second, we note that the policies based on influence vs. loss behave very similarly. While influence has slightly better performance (particularly on the NORB dataset), the policies are, for the most part, equivalent. A benefit of this is that the loss calculation is slightly simpler than influence to calculate, as it does not require calculating the inverse Hessian component, $\boldsymbol{H}_{\hat{\theta}}^{-1}$, as described in 3.1. Third, we note that it is possible to achieve *higher* accuracy than full augmentation using only a reduced set of points for augmentation, as observed in several of the plots (most notably on NORB). We believe that this higher performance may be due to a stronger bias towards harder examples in the dataset as compared to full augmentation.

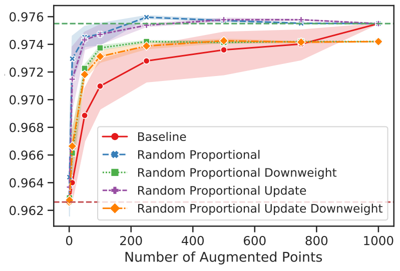
Finally, we explore the effect of using support vectors for augmentation, which was proposed in the Virtual Support Vector literature [44, 70]. In particular, we find VSV points by tuning a linear SVM on the bottleneck features of the original training set, and then using these points as the set of augmentation points for the logistic regression model with bottleneck features. We use search over $C \in \{0.01, 0.1, 1, 10, 100\}$ via cross-validation, and the best resulting model is used to obtain support vectors. Interestingly, we note that, though this transfer approach was not originally proposed in the VSV literature, it results in strong performance on a few of our tests (e.g., NORB-translate, NORB-crop, CIFAR10-rotate). However, the approach is not as reliable as the proposed policies in terms of finding the optimal subset of points for transformation (performing significantly below optimal, e.g., for MNIST and CIFAR10-translate), and the major limitation is that the augmentation set size is fixed to the number of support vectors rather than being able to vary depending on a desired data budget.

3.5.2 Refinements: Sample Reweighting and Score Updating

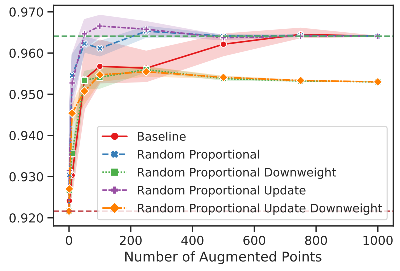
We additionally investigate the effect of two refinements on the initial policies: (i) reweighting the samples as they are added back to the training set and (ii) updating the scores as the augmentation proceeds, as described in Section 3.4.2. The latter policy assumes that the method is run in an online fashion, in contrast to the policies described thus far. This adds extra expense to the total run time, because the model must be continually updated as new points are augmented. In Figure 3.4, we observe the effect of these modifications for all datasets using the rotation augmentation with model influence as the score. Interestingly, while reweighting points seems to have a positive (if negligible) effect for MNIST, we see that it can actually hurt performance in CIFAR10 and NORB. This may indicate that the amplifying effect of augmentation may in fact be beneficial when training the model, and that reweighting may increase the role of regularization in a detrimental manner. In terms of the score updating, we see that, although updating the score can have a slight positive impact (e.g., for NORB-rotate), the performance appears to roughly match that of the original policy. Given the extra expense required in model updating, we therefore conclude that the simpler policies are preferable.



(a) MNIST-rotate



(b) CIFAR10-rotate



(c) NORB-rotate

Figure 3.4: The performance of policies when point downweighting is used or augmentation scores are updated.

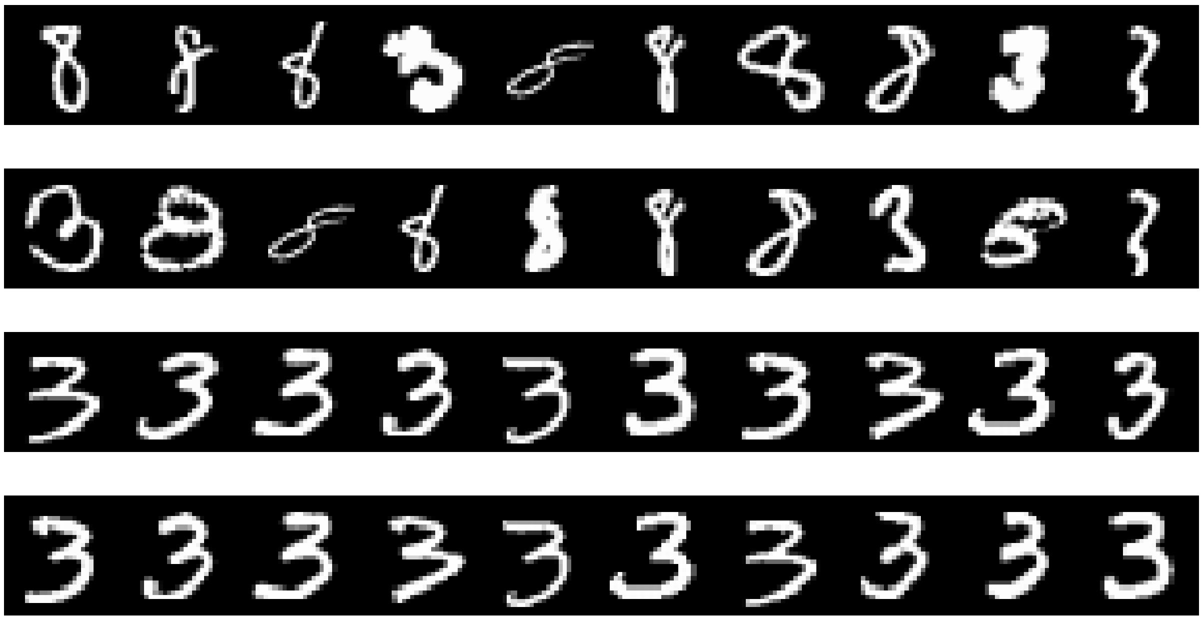


Figure 3.5: Points with highest influence / loss (top) and lowest influence / loss (bottom).

3.5.3 Understanding Policies

Finally, to give insight into the behavior of the proposed polices, we examine the 10 points with highest influence/loss vs. least influence/loss for MNIST. We observe similar results for the other datasets (CIFAR10, NORB). These examples help visualize the benefits of downsampling, as it is clear that the bottom set of points are all quite similar. The top points, in contrast, appear more diverse—both in terms of class label as well as features (thin lines, thick lines, slanted, straight, etc.). We postulate that promoting this diversity and removing redundancy is key in learning invariances through augmentation more efficiently.

3.6 Discussion

In this chapter, we demonstrate that not all training points are equally useful for augmentation, and we propose simple policies that can select the most viable subset of points. Our policies, based on notions of training loss and model influence, are widely applicable to general machine learning models. Obtaining access to an augmentation score vector can be obtained in only one training cycle on the original data (e.g., a fixed cost), yet the potential improvements in augmented training can scale superlinearly with respect to the original dataset size. With many fewer data points to augment, the augmentations themselves can be applied in a more efficient manner in terms of compute and expert oversight. At an extreme, they can be specialized on a per-example basis.

A natural area of future work is to explore subset selection policies that take the entire subset into account, rather than the greedy policies described. For example, even if two samples may independently have large leave-one-out influence, it may be the case that these points influence each other and leave-one-out influence may be an overestimate (e.g., consider the case of two identical samples). Including second-order information or encouraging subset diversity (e.g., via a determinantal point process (DPP) [161][4]) may therefore help to improve performance even further.

More recently, works such as *dataset echoing* have been proposed to satisfy high load on input pipelines [53]. Rather than exhausting the full set $\mathcal{D}$, echoing repeats a subset of $\mathcal{D}$ for a pre-configured number of times. In effect, the traversal of $\mathcal{D}$ is fixed to contain sufficient locality to cache and repeat the augmentations. Dataset echoing has been further expanded to support caching partially augmented samples [174]. Dataset echoing is similar to this work in that a subset of (a usually infinite set of) samples is repeated. If this subset contains $\mathcal{S}$ concatenated with the original training set, the policy is identical to ours. However, unlike this work, dataset echoing does not usually take into account how influential a data point is, and therefore, there is room to further expand on these works, taking into account both the points relevance to a task as well as allowing a more general framework for repeating the points. In any case, these types of pipeline approximations can be implemented in a pipeline tuning framework (§5) with either materialized caches or sample repetitions, both of which can be modeled.

[4]Our experiments using DPP indicate that DPP is more computationally taxing with little benefit for the setup we use.

Chapter 4

Progressive Data Loading and Image Fidelity

In this chapter, we introduce a new data loading format compatible with all major training frameworks. The major distinguishing factor for this format is that it is *dynamic*—the same files can be read at varying levels of fidelity without any data reorganization. In turn, the system can increase or decrease image fidelity to cope with system cost constraints while still matching the fidelity necessary for the model being trained. The cost model and techniques developed in this chapter are general and can be integrated with a system, such as the one introduced in §5.

4.1 Introduction

Deep learning training consists of three key components: the data loading pipeline (storage), the training computation (compute), and, in the case of parallel or distributed training, the parameter synchronization (network). A plethora of work has investigated scaling deep learning from a compute- or network-bound perspective [11, 15, 63, 64, 69, 139, 179, 181, 297, 301, 321, 324]. However, little attention has been paid to scaling the storage layer, where training data is sourced.

Current hardware trends point to an increasing divide between compute and the rest of the hardware stack, including network or storage bandwidth [167, 176, 179] and main memory [168, 257, 307]. Indeed, in the last decade, the amount of compute available to deep learning (DL) has increased exponentially [227]. However, I/O bandwidth has been slower to evolve, potentially resulting in I/O becoming a dominating factor in the overall runtime of deep learning tasks [167, 176, 202, 294]. Recent evidence highlights that, while accelerators are the workhorse of any ML fleet, 30% of resources are spent on the data pipeline [211] in industrial workloads and up to 65% of epoch time is spent in data pipelines in research workloads [206]. While I/O is only a part of the data pipelines, it has the possibility to create bottlenecks and thus lower end-to-end ML training efficiency.

The resource requirements for I/O can be prohibitive, either due to cost, scaling limits of filesystems, or quality of service requirements. Figure 4.1 shows that this can be problematic even at small scales. We mark the time it takes to fetch one training batch using 1GiB/s of data bandwidth with a dashed red line, since cloud instances are typically limited to 1–4 GiB/s of

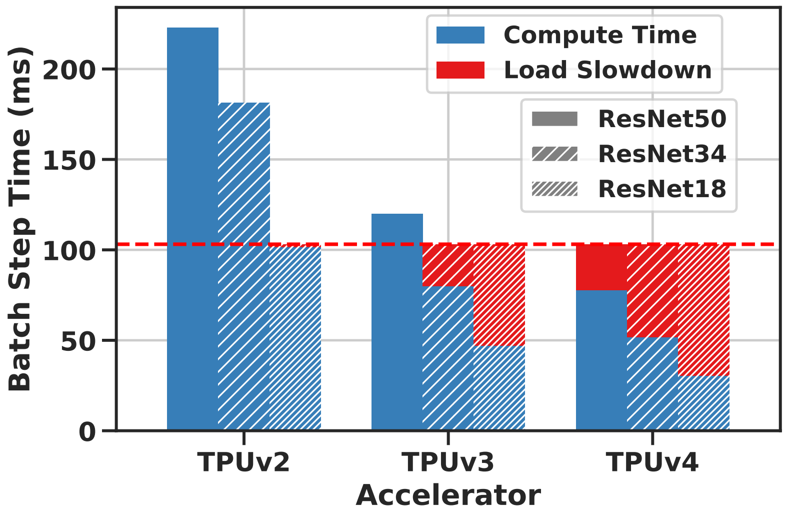


Figure 4.1: Three generations of single-node TPU hardware performance on ResNet/ImageNet (batch size of 1024 images). A 1GiB/s (observed in §5.2.4) limit of data bandwidth is shown (red line) with the corresponding expected mean slowdown. Faster hardware combined with smaller models places more stress on the input pipeline.

network [103] and 1GiB/s of disk bandwidth [102]. We have trained ResNet [123] models of varying complexity (ResNet18 being the least complex) using the ImageNet [71] dataset. We find that TPUv2 [140], Google's second version of their custom AI accelerator, completes computation on a given training batch within the time it takes to fetch the next batch (according to the dashed red line). However, the least complex of the models, ResNet18, is expected to toe that line. TPUv2 is 6 years old, and the third version of Google's accelerator manages to pack enough compute to speed up batch computations so that two out of the three ResNet models spend more time fetching the next training batch than computing on the current one. This is projected to become a problem for even more complex models, according to publicly available per-core performance numbers for TPUv4 [203], which we include in the figure.

These trends highlight that I/O, if left unaltered, stands to dominate training costs. Worse, if the underlying data used in training were to get larger, the problem could become much worse. Contrary to conventional wisdom, datasets like ImageNet consist of *small* images with an average image resolution that is 7× smaller than industry workloads [242], and thus the combination of training *rates* and *data sizes* are likely to increase.

To cope with the divide between compute and I/O, architects have turned to hardware/software co-design in an attempt to meet scaling and efficiency goals [140, 165]. Two common, complementary approaches to optimize the storage layer include *caching* [162, 206] and *reducing data volume* [147, 185]. From the caching point of view, I/O pressure can be relieved by keeping a subset of the workload in memory, and optimizing access patterns to hit in the cache and thus avoid I/O. However, for large datasets, one must choose between prohibitively large cache sizes or weaker forms of sampling [197]. From the point of view of data reduction, practitioners can resize images to reduce their size. However, choosing the resize parameters is task-specific, and is subject to error [280], which diminishes task performance. In this work, we show deep neural

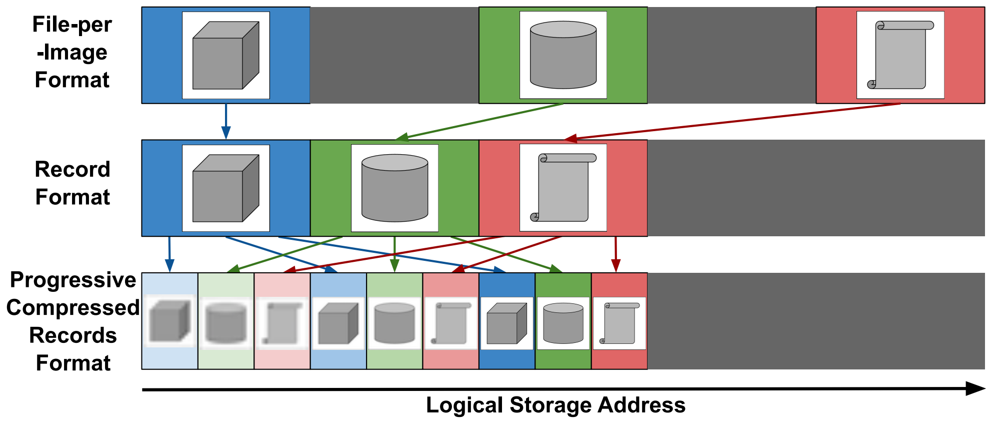


Figure 4.2: The design space of ML image file formats. File-per-Image (**Row 1**) formats randomly read files in the logical storage address space. Record layouts (**Row 2**) batch a subset of files into a single, large sequential read, promoting locality in address accesses. PCRs (§4.3, **Row 3**) group by image fidelity (3 shown) to maintain the sequential behavior of record layouts while enabling dynamic compression without the need for duplicating data. Reading a full record recovers the full data fidelity for all images. Metadata (not shown) is small and can be kept in memory.

network training is amenable to a range of JPEG compression; however, unlike resizing, this fact can be exploited as a *mechanism for dynamic data reduction*. Notably, we show that different training tasks—a product of the dataset, model(s), and objective, and it is non-trivial to determine these levels *a priori*, which motivates a need for dynamic compression.

In this work, we propose *Progressive Compressed Records* (PCRs), a novel data format that reduces the bandwidth cost associated with DL training. Our approach leverages a compression technique that decomposes each data item into a series of *deltas*, each progressively increasing data fidelity. PCRs use deltas to *dynamically* access entire datasets at a fidelity suitable for each task's needs, while avoiding inflating the dataset's size. This allows training tasks to control the trade-off between training data size and fidelity. The careful layout of PCRs ensures that data access is efficient at the storage level. Switching between data fidelities is lightweight, enabling adaptation to changing runtime conditions.

4.2 Background

Advances in scalable training methods, software, and compute (e.g., accelerators) suggest that the time spent on training computation is decreasing relative to time spent accessing data [162, 167, 176, 179, 206, 294]. Data bandwidth is therefore an increasingly important bottleneck to consider for machine learning pipelines. Two complementary concepts make up the process of storing/loading data: the *data layout*, which helps to utilize the bandwidth of the underlying storage system, and the *data representation*, which can increase bandwidth by reducing the amount of data transferred. In this work, we develop a novel, flexible, and efficient storage format, PCRs, by combining a data representation (progressive compression) with an efficient data layout. Our work serves to lower three fundamental storage costs: storage capacity, storage operations

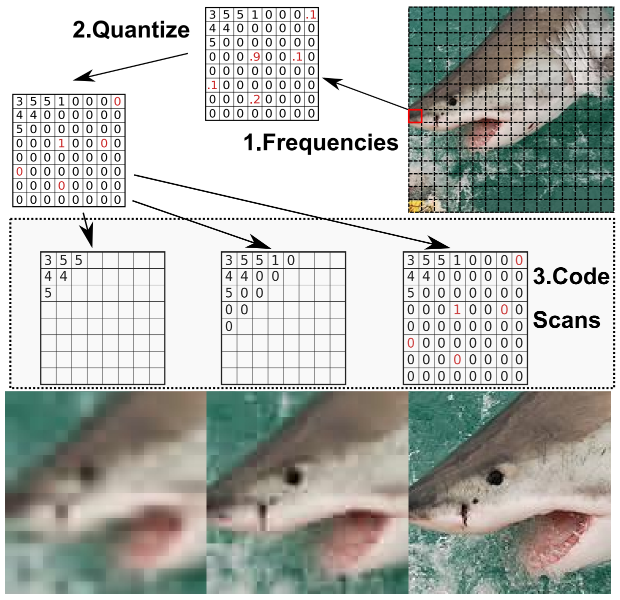
(IOPS), and storage/network bandwidth.

Record Layouts. Learning from data requires sampling points from a training set. In the context of image data, perhaps the simplest way to access data is with a *File-per-Image* layout, such as PyTorch's ImageFolder, which can cause small, random accesses that are detrimental to storage bandwidth and latency, while also stressing filesystem metadata [194, 234, 300]. *Record layouts*, such as TensorFlow's TFRecords [276], MXNet's ImageRecord [241], or even TAR files [234], attempt to alleviate this problem by batching data together into records. Record layouts increase performance (i.e., read rate) by exploiting locality (Figure 4.2). Our experiments indicate a single epoch can take $25\times$ longer with File-per-Image formats compared to reading Record formats—limiting their practicality without caching. To achieve randomness, each Record is read into memory, where it may be shuffled with other Records and broken into minibatches by the data loader. While Record layouts improve over File-per-Image layouts, they are designed to store data at a specific fidelity level, thus requiring multiple copies of each dataset at different fidelities in order to realize efficient training across tasks. In this work, our aim is to combine the efficiency of Record layouts with dynamic compression schemes (described below) to offer quick, easy access to data at multiple fidelity levels.

Image Compression. Compressed formats are commonly used to represent training data. JPEG [290] is one of the most popular formats for image compression and is used ubiquitously in machine learning [71, 88, 180, 252]. Most compression formats (including JPEG) require the compression level to be set at encoding time, which often results in choosing this parameter in an application-agnostic manner. However, as we show in Section 4.5, it is difficult to set the compression level for deep learning training without over- or under-compressing, as the appropriate level may vary significantly across training tasks. Current approaches resort to storing multiple copies of the dataset at different compression levels, particularly for applications using multiple data fidelities within a single training task [147]. This is infeasible for larger datasets. For example, we find duplicating a 2GiB dataset at 9 resolutions can *amplify the dataset size by* $\mathbf{1.5{-}40}\times$ *and require hours of extra processing time*. Terabyte-sized datasets rely on distributed frameworks to reduce dataset creation from weeks to days [21].

In Figure 4.3, we provide a simplified illustration of the JPEG algorithm. First, an image is split into blocks of size 8×8, which are then converted into the frequency domain. The low frequencies (top left of the matrix) store the bulk of the perceptually relevant content in the image. Quantization, which discards information from the block and results in compression, is used to diminish the high frequency values, compressing the data. *Sequential formats* serialize the image's blocks from left to right, top to bottom. Decoding the data is simply a matter of inverting this process.

Progressive Image Compression. Progressive compression is an alternative to standard image compression, which—combined with an additional rearrangement of data (Section 4.3)—forms the basis of the idea behind PCRs. *Progressive* formats allow data to be read at varying degrees of compression without duplication. As an example, over slow internet connections, these formats allow images to be decoded *dynamically* as they are transmitted over the network. With the



(a) Scan 1 (b) Scan 3 (c) Scan 10

Figure 4.3: **Top:** JPEG carves an image into blocks, which are then converted into frequencies, quantized, and serialized. Progressive compression writes out key coefficients from each block before re-visiting the block. **Bottom:** Higher scans (a→c) have greater fidelity from more frequencies.

sequential case, data is ordered by blocks, and thus partially reading the data results in “holes” in the image for unread blocks [290]. Dynamic compression schemes interleave information (*deltas*) from each block, allowing all blocks (and thus the entire image) to be approximated without reading the entire byte stream. As progressive formats are simply a different traversal of the set of quantization matrices, they contain the same information as sequential JPEG [142] and are actually often smaller in practice. As we depict in Figure 4.3, while non-progressive formats serialize the image matrix in one pass, progressive formats serialize the matrix in disjoint groups of deltas which are called *scans*. Scans are ordered by importance (e.g., the first few scans improve quality more than subsequent scans). Thus, any references to images generated from scan n will implicitly assume that the image decoder had access to prior scans. Progressive formats exist not only for images, but also for modalities such as audio [205] and video [256].

4.3 Progressive Compressed Records

We present *Progressive Compressed Records* (PCRs), a novel storage format that reduces data bandwidth for ML training. We specifically explore PCRs for training deep neural networks with image data, but note that the ideas behind PCRs could be readily extended to other modalities (e.g., audio [205] or video [256]), and compression strategies (e.g., cropping [263], interlaced PNG, or neural compression [278]). PCRs define a data layout that ensures bandwidth is fully utilized, and a data representation that permits accessing data at multiple levels of fidelity with

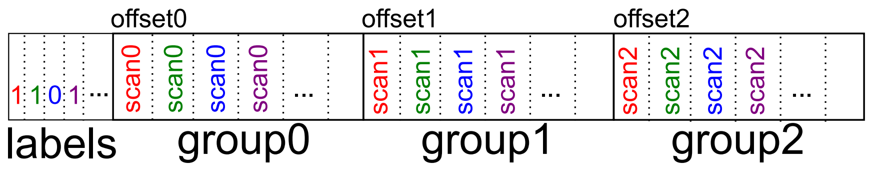

offset0
offset1
offset2
1 1 0 1 ...
scan0 scan0 scan0 scan0 ...
scan1 scan1 scan1 scan1 ...
scan2 scan2 scan2 scan2 ...
labels
group0
group1
group2

Figure 4.4: PCRs encode label metadata followed by all scan groups. Accessing the dataset at lower quality requires reading up to a given scan group. Reading all scan groups returns the full quality data, and decodes to identical bytes as the conventional JPEG format.

minimal overhead.

PCRs are optimized to allow the entire training dataset to be read at a given fidelity. To achieve this, data is rearranged into *scan groups*, i.e., collections of deltas of the same fidelity that are stored together in the address space. To dynamically increase the fidelity of data read and decoded, a task then merely needs to read subsequent scan groups until the desired fidelity level is reached. PCRs differ from other formats (e.g., TFRecord, RecordIO) because PCRs allow these lower fidelity versions of each record to be accessed efficiently (without space/throughput tradeoffs). This efficiency is achieved by using progressive compression and changing the order that data is stored and accessed. Space overhead for PCR conversion is negligible; PCRs are usually 5% smaller than TFRecords. This is because record format size is dominated by the image payload, which is *simply rearranged* with progressive compression. File size differences stem from the efficiency of entropy coding in JPEG, which typically has higher compression ratios over progressive layouts [266]. Since PCRs allow a lower fidelity version of the entire dataset to be accessed efficiently, they can drop the effective size and utilized bandwidth of a record by a factor of 2–10$\times$.

Figure 4.4 depicts the PCR format as it is organized on the storage medium. PCRs logically consist of two parts: *metadata* and *data*. Metadata consists of sample metadata (e.g., labels, bounding boxes) as well as PCR metadata (e.g., mapping of files and scans to storage addresses). Metadata is small in size (e.g., an image label can be represented by a 32 bit `int`, while an image is 100kiB or more) and can be kept in a database, mapped in memory, or pre-pended to PCRs (for per-sample metadata e.g., labels). The data, which is orders of magnitude larger, consists of the images themselves, organized in terms of increasing levels of fidelity. Each fidelity level for an image is a *scan* and each grouping of images of the same fidelity is a *scan group*. For example, the scan 1 representation of the shark in Figure 4.3 can be retrieved by reading its data from scan group 1. Likewise, the scan 3 representation will be available once the records up to scan group 3 are read, and the reconstructed representation will be of higher fidelity than that of scan 1. As scan groups consist of scans of the same fidelity, every image contained in a record of the same group offset is available at the same fidelity. Users of PCRs can read data at a given fidelity by simply reading the on-disk byte stream from the start of the PCR to the end of the corresponding scan group. This way of dynamically selecting data fidelity allows for bandwidth savings without re-encoding the data.

4.4 Design

The key insight behind PCRs is that, for storage or network I/O bound workloads, training tasks can be sped up by reducing data fidelity (and, thus, the amount of data read) to match the available I/O bandwidth. Figure 4.5 shows the training throughput obtained by using PCRs vs. the traditional TFRecord format. As a fair point of comparison, we use our `tf.data` [211, 272] implementation, and thus only the dataset reader operation has changed. As we describe in Section 4.4.1, PCRs can reduce the bytes read per image, and thus proportionally increase the throughput of the end-to-end training process (up to the compute limits of the accelerator). However, a speedup is only possible if the CPU overhead introduced by PCRs can be absorbed by the machine (Section 4.4.2). The final part of the design of PCRs involves choosing the image quality level automatically, which we describe in Section 4.4.3.

4.4.1 I/O Speedup Analysis

End-to-End Slowdown. Amdahl's Law [18] states that if p fraction of a program is waiting for data (see red/blue bars in Figure 4.1), a $\frac{1}{1-p}$ speedup can be obtained by removing the wait. Recent work determines possible speedups empirically by finding the gap between the data preparation rate and the I/O rate [206]. However, because PCRs are dynamic, it is important to know what PCR configurations can actually lead to a speedup (i.e., what scan group to select). Although we observe over $500\times$ max/min range on ImageNet, mean size-per-sample, $\mathbb{E}_{x\sim\mathcal{D}}[s(x)]$, is all that is required for an accurate performance model. We tabulate this information in Table 4.1 and motivate the model below.

Table 4.1: Image size reduction for various scans and the size of an average image, which can be combined to predict I/O speedups. Scan 10 is approximately the same size as baseline JPEG, and scan 5 is roughly half.

Dataset	Scan 1	Scan 2	Scan 5	Scan 10	$\mathbb{E}_{x\sim\mathcal{D}}[s(x)]$
ImageNet	$16\times$	$7\times$	$2\times$	$1\times$	110kB
HAM10000	$30\times$	$15\times$	$3\times$	$1\times$	250kB
Cars	$14\times$	$6\times$	$2\times$	$1\times$	110kB
CelebAHQ	$7\times$	$4\times$	$3\times$	$1\times$	80kB

Input Pipeline Throughput. Using closed-system Little's Law [119, 182] and basic assumptions on the characteristics of a storage system (i.e., the cost of large reads is proportional to bytes read), the image throughput, X (e.g., images per second), of an image pipeline is explained by the equation: $X = \dfrac{W}{\mathbb{E}_{x\sim\mathcal{D}}[s(x)]}$, where W is the bandwidth and $\mathbb{E}_{x\sim\mathcal{D}}[s(x)]$ is the mean image size (average size of an image sample). The number of bytes in a record (a large read) is, by linearity of expectation, the number of images, n, times the average image size. Thus, the amortized cost per image (dividing by n) is the average image size, and time taken is proportional to W. If a

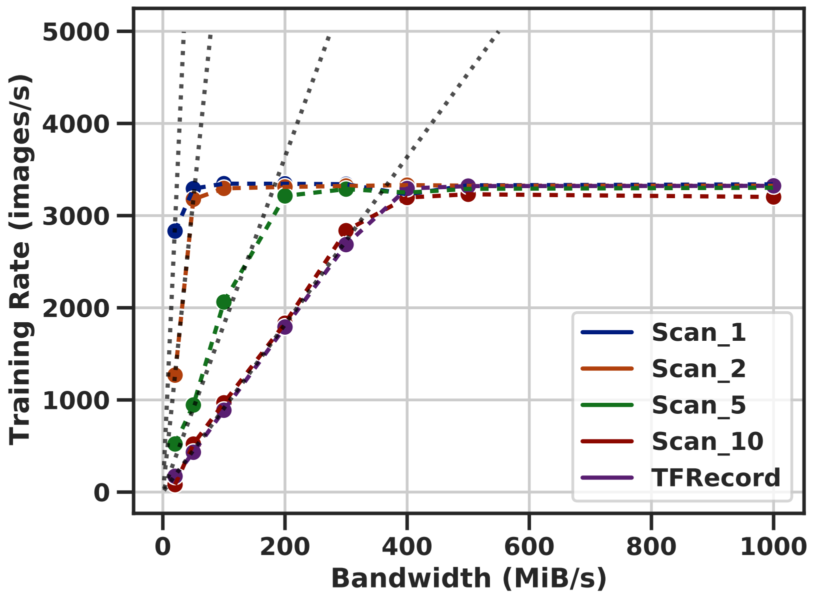


Figure 4.5: The training rate of a 10-node `TitanX` GPU cluster with a ResNet-18 workload using PCRs (the scans) and TFRecord. The throughput of the training process is dominated by I/O bandwidth until the compute limit of the GPU is reached. PCRs at scan 10 are approximately the same size as TFRecord, and thus have similar performance. Predicted rates are shown.

model/accelerator trains at 500 images/second (a function of the resized and cropped input-matrix dimensions [123, 156, 262]), we can conclude that, using ImageNet images, it will use up to $110\text{kB} * 500\text{s}^{-1} = 55\text{MB/s}$ (Table 4.1), as demonstrated in Figure 4.5.

Dataset-Level Bounds. To remove dependence on the accelerator's speed, the equation can be applied on both scans and the original data: Theorem 4.4.1 presents the asymptotic bounds for the impact of data reduction on training speedups. It is derived by noticing that, when a system with fixed W is bound by the throughput of the I/O subsystem, X, one can calculate the speedup ratio $\hat{X}\ /\ X$, where $\hat{X}$ is using a reduced image size. In sum, reducing the mean data read results in proportionally higher I/O throughput, which results in proportional speedup on I/O bound workloads. For example, using Table 4.1, which displays the ratios, one can anticipate that a $2\times$ speedup would be seen on ImageNet with scan 5.

Theorem 4.4.1 *If a training pipeline is data bound, then the maximum achievable system speedup, S_{max}, for switching from dataset $\mathcal{D}$ to $\mathcal{D}'$ is the ratio of mean sample sizes, $s(x)$:*

$$S_{max}\left(\mathcal{D}, \mathcal{D}'\right) = \frac{\mathbb{E}_{x\sim\mathcal{D}}[s(x)]}{\mathbb{E}_{x'\sim\mathcal{D}'}[s(x')]}.$$

4.4.2 Data Preparation Decoding Overhead

Changing the dataset encoding inevitably changes data-preparation work, which consumes CPU resources and must be managed. The cost of progressive compression is dependent on the image size, scan configuration, and decoder implementation [312]. To analyze the cost of this progressive decoding, we test the rate of decoding $30k$ images using one thread—these rates can then be multiplied by the number of cores on the machine in a parallel training scenario. The figures are shown in Table 4.2, indicating that decoding with a subset of scans can be comparable to traditional decoding. On the other hand, using all the scans is over $2\times$ slower than traditional decoding. On a many-core machine (e.g., a 32+ core setup like the one used in Section 4.5.1), this overhead can be absorbed by idle cores—in Figure 4.5, we do not see any slowdown by using PCRs relative to baseline JPEG TFRecords because each of the 10 machines can decode 3k images in the worst case. However, for less core-heavy machines, we note three optimization paths to lower decoding overhead. First, excessive and unused scans can be removed. Second, using spectral selection can lower decoding overhead. Third, hardware acceleration can be used.

Table 4.2: The single-core decoding rate (images/s) of various JPEG encodings across the datasets. Progressive decompression can be over $2\times$ more expensive than baseline decoding.

Dataset	Scan 1	Scan 2	Scan 5	Scan 10	Baseline
ImageNet	433	412	340	146	419
HAM10000	465	438	275	96	240
Cars	266	240	225	127	268
CelebAHQ	239	213	195	129	286

4.4.3 Autotuning Image Fidelity

Lossy compression of input data creates concerns for the output of model training, and thus creates questions for how to select a tolerable scan group. To analyze the effect of lower image fidelity, we observe that deep learning training is based on stochastic gradient descent, which involves taking a "step" in the *direction* (a vector) that improves the model. If two datasets, $\mathcal{D}$ and $\mathcal{D}'$, yield the same direction, then they will yield the same model. Therefore, we may intuitively find an alternative dataset $\mathcal{D}'$, which is close to the original dataset $\mathcal{D}$ in terms of how the model views the gradient direction of the loss function, $\mathcal{L}$. More formally, we want the *angle* between gradient vectors to be small.

To accomplish this, we freeze the model mid-training and empirically measure the gradient direction, $\nabla_\theta \mathcal{L}$, over the full fidelity dataset, $\mathcal{D}$, which contains batches of images, $\mathbf{X}$, and labels, $\mathbf{y}$. As the parameters are frozen, we can also measure the gradient on the lower fidelity dataset, $\mathcal{D}'$, which has alternative images, $\mathbf{X}'$. The angle between the lower fidelity dataset and the original dataset yields the similarity score, which ranges between -1 and 1. Maximizing similarity would yield an *identical* model.

$$\text{score}\left(\mathcal{D}, \mathcal{D}'\right) = \text{sim}\left(\nabla_\theta \mathcal{L}(\mathbf{X}, \mathbf{y}), \nabla_\theta \mathcal{L}(\mathbf{X}', \mathbf{y})\right)$$

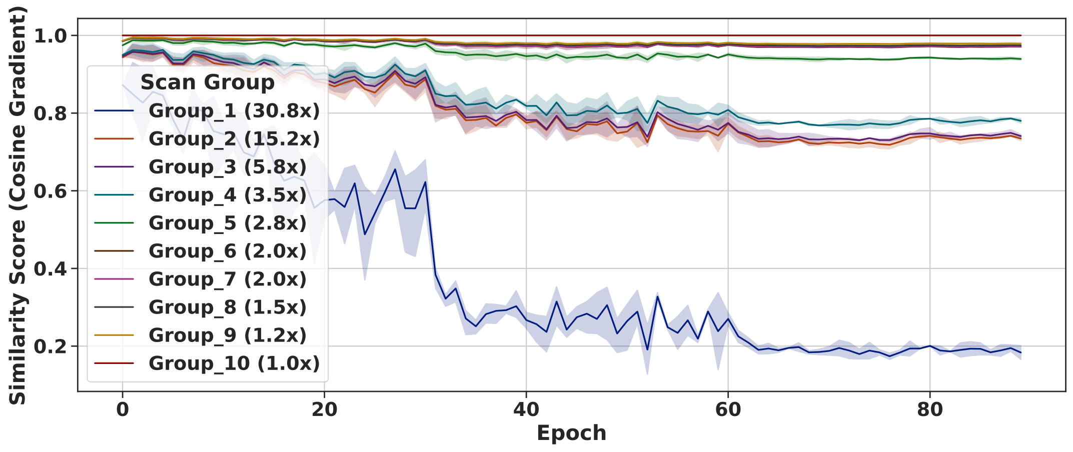


Figure 4.6: The similarity of gradients across epochs for ResNet/HAM10000 (max of 1.0). Legend shows bandwidth savings. Gradient similarity is exact for scan group 10 and decreases for other scans as the model converges. Higher quality scans lead to gradients within 0.1 of the baseline's gradient, and thus should result in similar final models.

where similarity is the cosine similarity:

$$\text{sim}\left(\mathbf{A}, \mathbf{B}\right) = \cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{||\mathbf{A}||||\mathbf{B}||}$$

We evaluate this procedure with HAM10000/ResNet, using 2560 images to estimate the gradient and 3 trials to get 95% confidence intervals. As shown in Figure 4.6, decreasing image fidelity decreases the similarity with respect to the true gradient. Scan 10 is bit-identical to the baseline dataset, and thus we observe maximum similarity in that case. Meanwhile, scan 1 has the lowest similarity, and the difference increases as the model converges. Given the gradients are well-behaved with respect to fidelity, it is natural to parameterize scan tuning to match some level of gradient similarity.

Using gradient similarity for autotuning quality requires choosing a minimum gradient similarity threshold for scans throughout training, which is the main drawback of this approach. As is shown in Figure 4.6, the similarity for high quality data is bounded within a threshold of 0.8—therefore, we find this threshold a good default. The computational cost of evaluation is on the order of tens of gradient steps, and is proportional to the number of scans, epochs, and minibatches used. Our implementation tunes once every 20 epochs and does not tune for the first 5 epochs because models are unstable during this period [110, 123]. As training progresses, low scans become too dissimilar (in terms of gradients) from the higher scans, and therefore are avoided. This, in turn, allows the faster scans to apply a burst of speed to the training process before fine-tuning at a higher fidelity (e.g., when learning rates drop). Compared to static schedules [295], there is only one hyperparameter (the threshold), which is independent of other schedules (e.g., learning rates), and the parameter does not require validation data. We leave tuning using QoS/congestion information [95] to future work.

4.5 Evaluation

We evaluate the flexibility and efficiency of PCRs using a suite of large-scale image datasets. We begin by describing our experimental setup (§4.5.1) and present an end-to-end evaluation of PCRs (§4.5.2), demonstrating their ability to reduce training time. We show that dynamic compression is crucial because the appropriate level of compression varies across models and training tasks (§4.5.3). We explore metrics that can be used to explain the effectiveness of compression on a training task (§4.5.4) and introduce autotuning heuristics for dynamic training (§4.5.5).

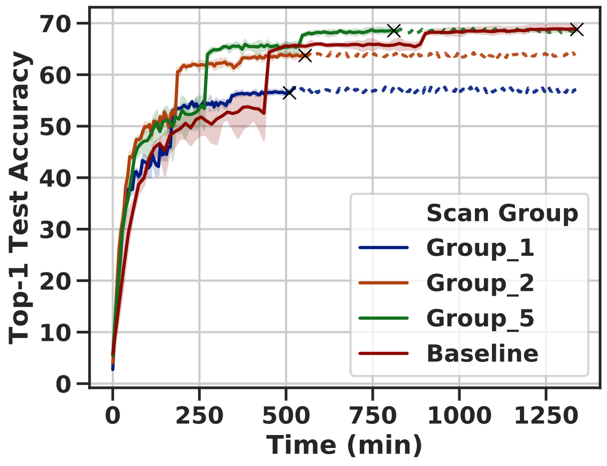
Table 4.3: PCR dataset size and record count information. Datasets vary in terms of number of images, their JPEG quality, and the image sizes. Some datasets, such as HAM10000, have image sizes larger than average. Record sizes concentrate around the dataset size divided by the record count.

Dataset	Records	Images	Size	Quality	Classes
ImageNet	1251	1281167	129GiB	91.7%	1000
HAM10000	125	8012	2GiB	100%	7
Cars	63	8144	887MiB	83.8%	196
CelebAHQ	93	24000	2GiB	75%	2

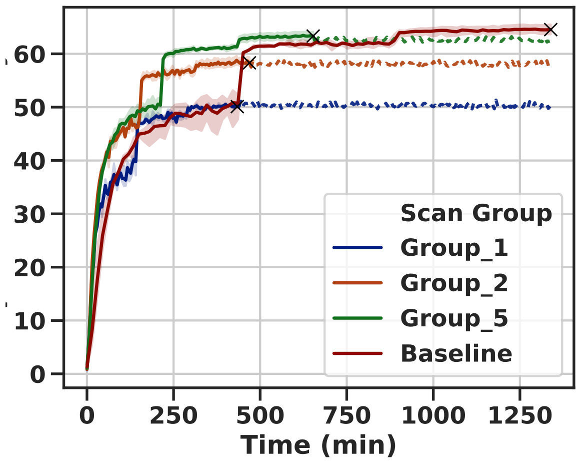
4.5.1 Experimental Setup

Our evaluation uses the ImageNet ILSVRC [71, 252], CelebAHQ [147], HAM10000 [281], and Stanford Cars [154] datasets. For CelebAHQ, we classify if the celebrity is smiling or not. A summary of each dataset is given in Table 4.3. We aimed to select datasets that vary in terms of the image resolution, number of examples, number of classes, and image/scene type. *All of the datasets are in fact already compressed before progressive compression is applied, making the presented speedups conservative estimates of the potential benefit of PCRs.* Specifically, the datasets natively use a JPEG quality level varying from 75% (CelebAHQ) to 100% (HAM10000) (§4.3). Experiments use resizing, crop, and horizontal-flip augmentations, as is standard for ImageNet training [269, 280] The sizes of each dataset's scan groups (used in Table 4.1) are shown in Figure 4.8; sizes decrease for lower scan groups.

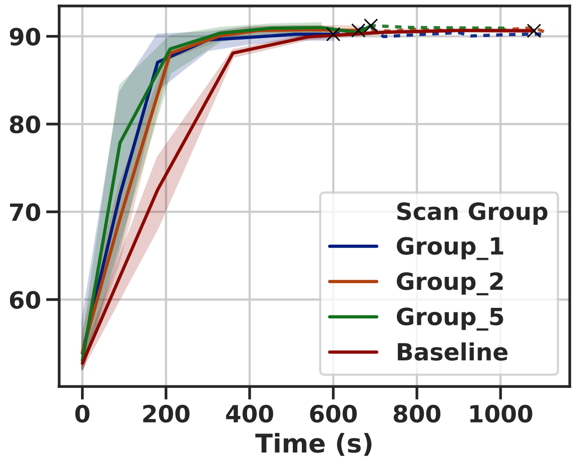
Training Regime. We evaluate two loader implementations of PCRs, comparing PCR scans against themselves and then comparing PCRs against strong baselines (TFRecords). For both setups, we use PyTorch [222] for all model training; the two loaders are using DALI [214] Loader, which was used for initial prototyping, and `tf.data` [211, 272], which we have since made native operator extensions to for maximum performance. In our experiments we use pretrained ImageNet weights for HAM10000 and Cars due to the limited amount of training data. We use standard ImageNet training, starting the learning rate at 0.1 with gradual warmup [110], and dropping it on epochs 30 and 60 by $10\times$. After augmentations, all inputs are of the same size; thus, a model's update rates are the same across datasets. The pretrained experiments (HAM10000



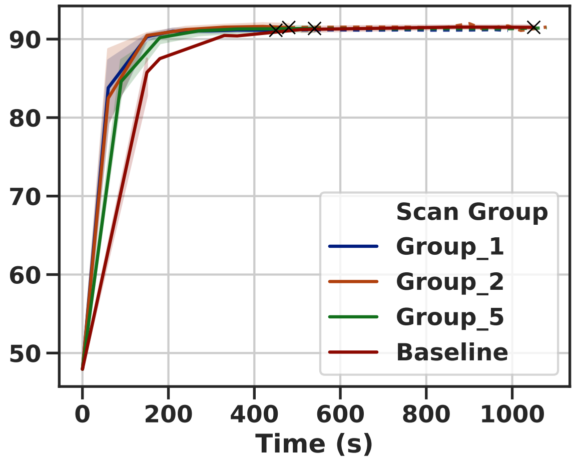
(a) ImageNet ResNet



(b) ImageNet ShuffleNet



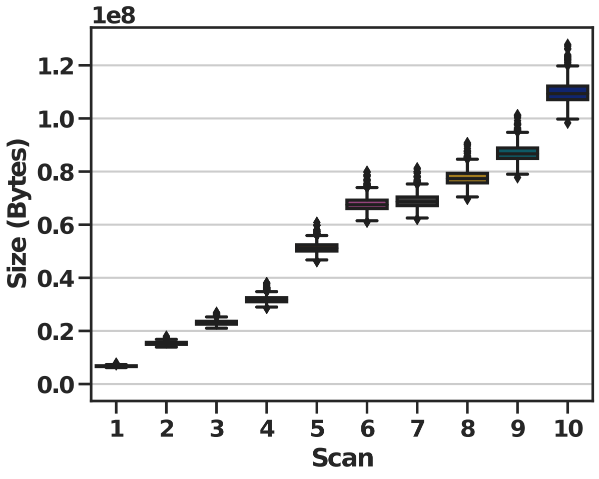
(c) CelebAHQ ResNet



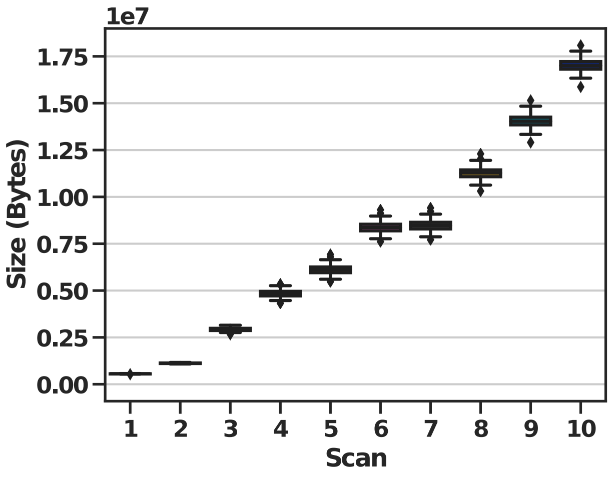
(d) CelebAHQ ShuffleNet

Figure 4.7: Top-1 test performance (with 95% confident intervals) using ResNet and ShuffleNet on ImageNet (a,b) and CelebAHQ (c,d). Lower scan groups (corresponding to further compressed data) can provide faster overall training, often without sacrificing accuracy. However, the appropriate level of compression depends on the model, infrastructure, and data—necessitating the ability to easily access data at multiple fidelities, as with PCRs. We explore these factors in Sections 4.5.3, 4.5.4, and 4.5.5.

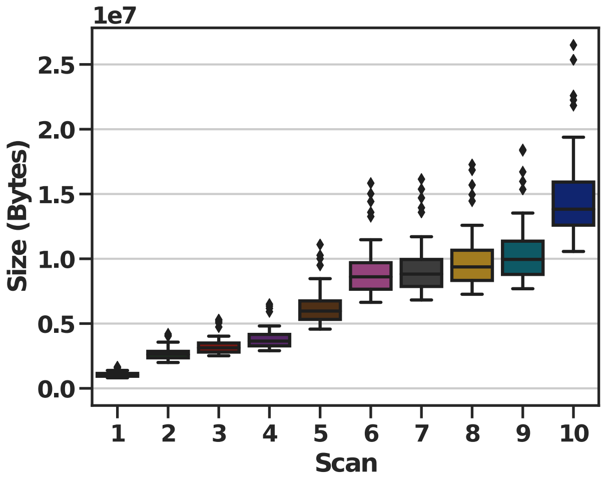
and Cars) start at a learning rate of 0.01 to avoid changing the initialization too aggressively. We use mixed-precision training [22, 199] for the DALI runs. We use ResNet-18 [123] and ShuffleNetv2 [189] architectures for our experiments with a batch size of 128 per worker. We run each experiment at least 3 times to obtain confidence intervals. We sample test accuracy every 15 epochs for non-ImageNet datasets. Our evaluation focuses on the differences obtained by reading various amounts of scan groups. For the DALI runs, we consider reading all the data (up to scan group 10) to be the baseline, as the baseline formats will perform similarly under I/O bounds—we later provide a direct comparison with baseline TFRecords when using `tf.data`. Our results are



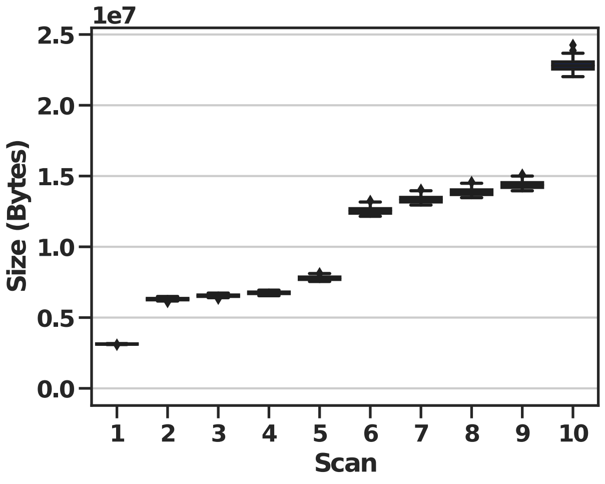
(a) ImageNet



(b) HAM10000

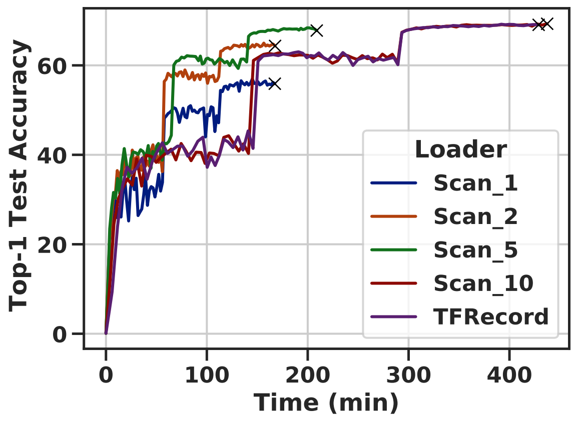


(c) Stanford Cars

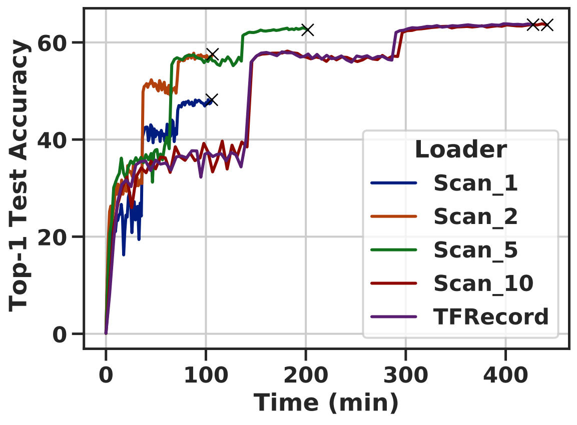


(d) CelebAHQ

Figure 4.8: The size in bytes of various levels of scans read. Scan group 0 (not shown) contains only labels and is typically $\sim$100 bytes. Each scan adds roughly a constant amount of data (i.e., linear scaling), although certain scans add considerably more than others (i.e., sizes sometimes cluster) due to techniques like chroma subsampling. Using all 10 scans can require over an order of magnitude more bandwidth than 1–2 scans. Interquartile ranges are shown.



(a) ImageNet ResNet (heavy)

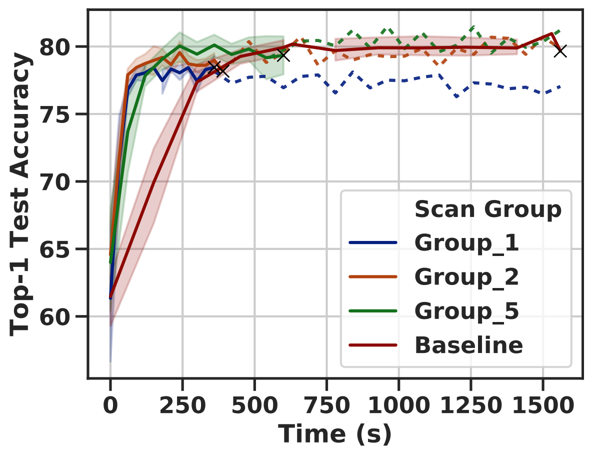


(b) ImageNet ShuffleNet (heavy)

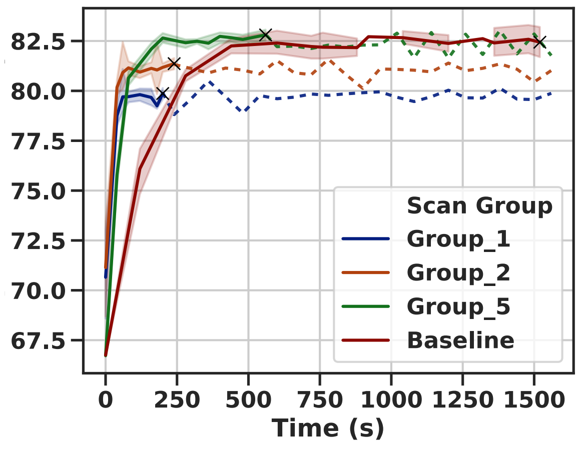
Figure 4.9: Top-1 test performance on ImageNet with ResNet and ShuffleNet, using double the compute (20 workers with same configuration). Doubling the compute forces bottlenecks to appear by approaching hardware limits of aggregate disk throughput. We run this experiment once and terminate at epoch 90, showing a $2\times$ speedup for scan 5.

conservative as we are already utilizing pre-compressed data and we include evaluation times in our results. For the purpose of evaluation, all scan groups within a dataset were run for the same number of epochs (90 for ImageNet, 150 for HAM10k, 250 for Cars, and 90 for CelebAHQ). We also provide annotated (dashed) lines for subsequent epochs.

System Setup. Our experiments were run on a 16 node Ceph [299] cluster with NVIDIA TitanX GPUs and 4TB 7200RPM Seagate ST4000NM0023 HDD. We use six Ceph nodes: five dedicated Object Storage Device (OSD) nodes, and one Metadata Server (MDS). The remaining 10 nodes are machine learning training workers. This 2:1 ratio between compute and storage nodes results in 400+ MiB/s of peak storage bandwidth; we have also tested a heavily I/O bound 10:1 ratio and found the trends comparable. Ceph is a common production-grade open-source filesystem, but our results would generalize to any setup with a mismatch between compute power and data bandwidth (either storage or network). In addition to microbenchmarks, we evaluated the generalization of PCRs to SSD setups in the cloud using `n1-instance-16` with a `P100` GPU, and found that we can get $2\times$ speedups over `TFRecords` in this alternative setting. Since state of the art compute is **$150\times$ faster** than our own setup on a more expensive model (ResNet-50) [315], we focus on models which are fast to train (while still being modern; AlexNet [156] is potentially faster) while limiting read parallelism. The DALI setup uses O_DIRECT to ignore caching effects and highlight bandwidth usage. To reflect what PCRs may look like in realistic, heavy-load situations, we provide 20 node experiments in Figure 4.9 with the same storage system and double the workers, which allows speedups to be seen with full read parallelism per node (over 700MiB/s of peak bandwidth). This setup uses our `tf.data` loader implementation to fairly compare against TFRecords and File-per-Image formats, showing its effectiveness. We use the same setup in Figure 4.14.

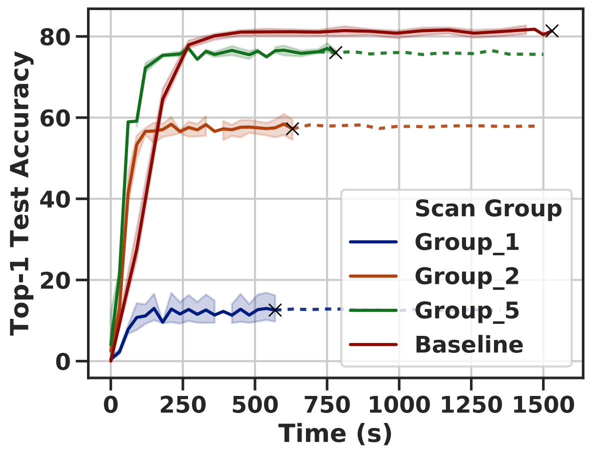


(a) HAM10000 ResNet

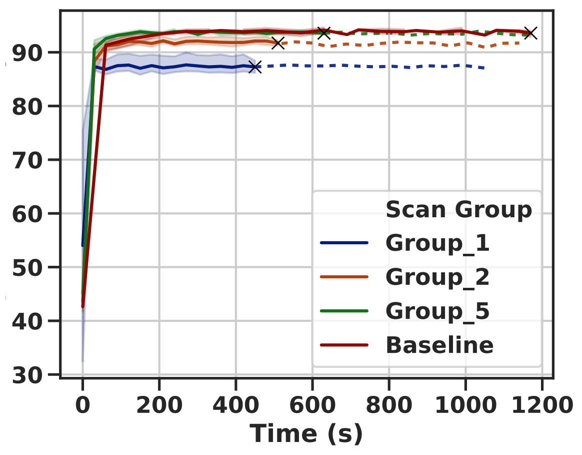


(b) HAM10000 ShuffleNet

Figure 4.10: Test accuracy on HAM10000. While ResNet is robust to additional compression, ShuffleNet requires higher fidelity data (at least scan group 5) for higher accuracy. Time is relative to first epoch. 95% confidence intervals are shown.



(a) Original Multiclass



(b) Binary Is-Corvette

Figure 4.11: Test accuracy with ResNet-18 on the Stanford Cars dataset and a binary variant. The gap between scan groups closes as the task is simplified. Time in x-axis is relative to first epoch. 95% confidence intervals are shown.

4.5.2 Reducing Time to Accuracy via Compression

Observation 3: *Training time can be reduced by up to $2\times$ using data compression. PCRs capitalize on this by dynamically reducing training data size, all without adding space overhead.*

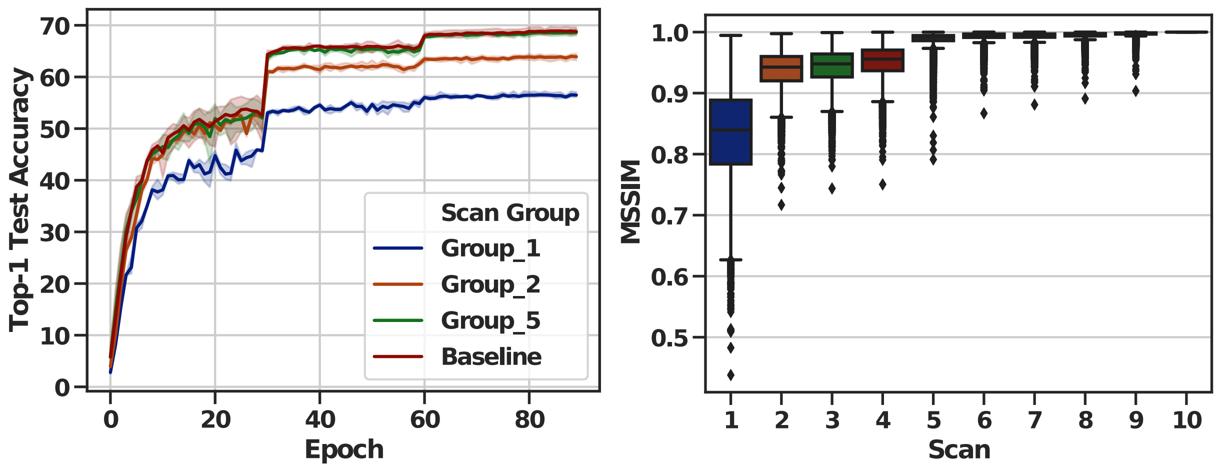
We begin our empirical study by exploring the effect of data compression on training time and training loss/test accuracy. We provide time-to-accuracy results for ResNet-18 and ShuffleNetv2 training on ImageNet and CelebAHQ (Figure 4.7), HAM (Figure 4.10), and Cars (Figure 4.11). Across these experiments, we find that PCRs can provide a $2\times$ boost on average in time-to-accuracy compared to the baseline, by dynamically providing data at a higher level of compression. We make several observations about these results. First, we note that we tend to see larger speedups for smaller, faster models (e.g. ShuffleNet), than for bigger models (e.g., ResNet). Indeed, the current speedups may in fact become significantly larger with faster compute [e.g., 164, 167, 315]. Such a trend is visible in the heavy ImageNet experiments featured in Figure 4.9—both TFRecords and scan 10 are about the same size, and therefore finish simultaneously, but scan 1 and 2 finish nearly an hour faster for ShuffleNet. For this same setup, we observe Files-per-Image take over 2 hours per epoch due to a lack of sequential reads—$25\times$ slower than TFRecords, which scan 5 improves by $2\times$; therefore, we conclude that progressive compression and record formats are both necessary for performance.

Second, while time-to-accuracy is reduced as we move to lower scan groups, there is a *statistical efficiency cost*. Namely, models trained on scans 1 and 2 may not always converge to an acceptable solution, as shown for ImageNet (Figure 4.7). Certain tasks like CelebAHQ, however, can tolerate this fidelity loss, either because they consist of less compressed images or because the training task is less dependent on high-frequency image features. These results suggest that, while compression saves bandwidth and offers a potential speedup, the ideal amount of compression depends on two factors: (i) the speed of the model and the underlying compute infrastructure, and (ii) the structure of the task and the images in the dataset. We explore these factors in more detail below.

4.5.3 Task Tolerance to Data Fidelity

Observation 4: *Different models can tolerate different fidelities.*

Given a fixed dataset, we show that there is variation in the data fidelity/compression level that different models can tolerate for training. This motivates an important use-case of PCRs, as the format allows data to be stored *once* but then accessed at multiple compression levels while models are tuned or various models are applied to the problem at hand. In Figure 4.10, both ResNet and ShuffleNet are trained with the HAM10000 dataset. While ResNet consistently tolerates low fidelity images, ShuffleNet training tends to degrade with low fidelity data. ShuffleNet reaches its best accuracy at scan 5, but our other results suggest that lowering fidelity results in lower accuracy for the same epoch in nearly all cases (Figure 4.12). This suggests that different models will experience different speedups for similar accuracy levels, depending on their sensitivity to fine-grained features unavailable in low fidelity data.



(a) Epoch Accuracy (b) MSSIM

Figure 4.12: Left: Top-1 test performance vs. epoch on ResNet-18/ImageNet; other models/-datasets are similar. Using lower quality scans can only degrade performance; it does not act as a beneficial data augmentation. Right: Corresponding image quality degradation according to MSSIM.

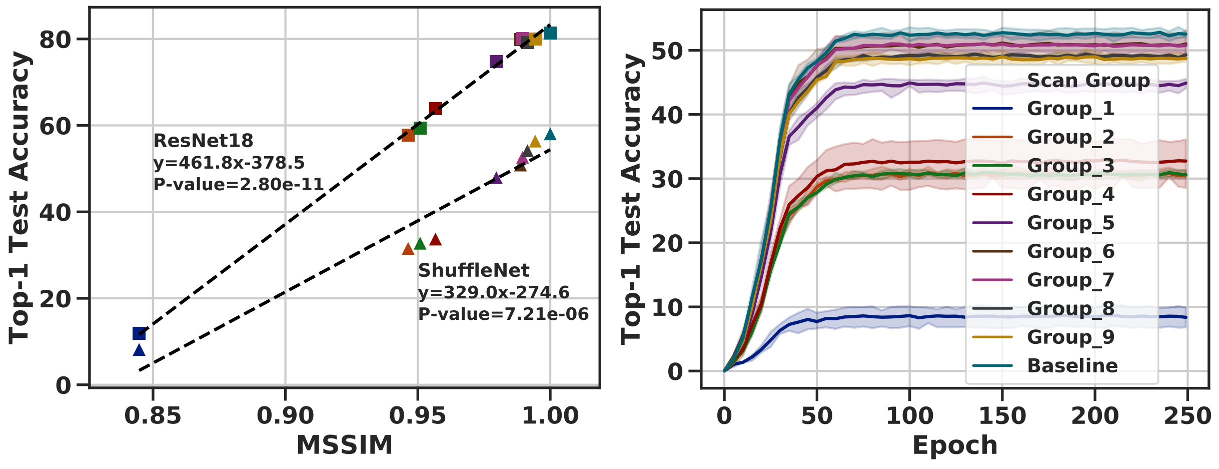
Observation 5: *Different tasks, e.g., multi-class classification vs. binary classification, can tolerate different levels of data fidelity. The same PCR dataset can service these different tasks.*

The difficulty of a task, or training objective of interest, also affects the amount of compression that can be tolerated. Harder tasks, e.g., multi-class classification with a large number of classes, require higher fidelity data. We validate this empirically in Figure 4.11 (and additional evidence is provided in the supplement). This experiment reduces the number of classes for the classification task, demonstrating that lower scan groups can be used for easier tasks. The full range of classes investigated includes: *Baseline* (i.e., Car Make, Model, Year), *Make-Only* (i.e., car Make only), and *Is-Corvette*, a binary classification task of Corvette detection. Compared to the original task, the coarser tasks reduce the gap between scan groups, decreasing the gap from baseline to the binary case. These results suggest that the optimal image encoding can be dependent on the exact labeling or task complexity. Thus, while static approaches may need one encoding per task, a fixed PCR encoding can support multiple tasks at optimal fidelity by simply changing the scan group depending on how the labels (metadata) are remapped.

4.5.4 Compression Level Estimation

Observation 6: *MSSIM image similarity is a reliable estimator of the accuracy loss between scan groups, and can be used to determine appropriate levels of compression for training with PCRs.*

To better explain the effectiveness of compression, we compare how various scans approximate the reference image through MSSIM [296], a standard measure of image similarity. We find



(a) MSSIM Regression (b) Clustered Convergence

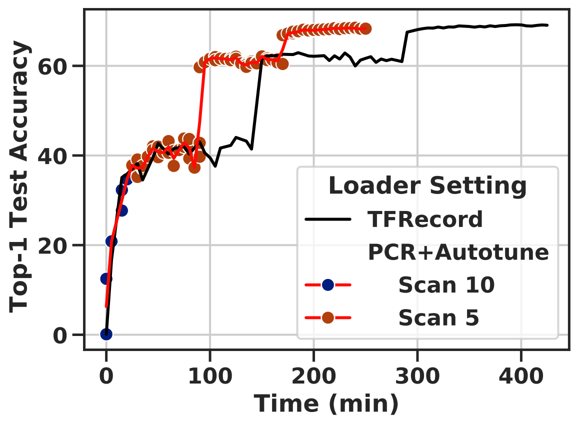
Figure 4.13: MSSIM vs. accuracy for the Cars dataset with ResNet18 and Shufflenet. We obtain similar results for other datasets. **Left:** There is a linear relationship between MSSIM and the final test accuracy. **Right:** Scan groups (ShuffleNet) cluster by MSSIM and accuracy.

a correlation between MSSIM and final test accuracy, especially when comparing scan groups *within* a task. Our preliminary tests show that scan groups with similar MSSIM achieve similar accuracy (Figure 4.13), which is why only scan groups 1, 2, 5, and the baseline are shown. Due to the way progressive JPEG is coded by default, groups tend to cluster, e.g., scans 2, 3, and 4 are usually similar, while 5 introduces a difference. Such "banding" or clustering is seen in the accuracy trends; the major jumps correlate with the appearance of Y (luminance) AC coefficients in the JPEG encoding. Scan groups of 5 or higher have an MSSIM of $95\%+$, which is likely why they consistently perform well. MSSIM can therefore be used as a diagnostic for choosing scans, although we acknowledge that changes in perception are hard to predict for large deviations (MSSIM $< 95\%$). For some datasets, linear regression on MSSIM recovers final test accuracy even with different models (Figure 4.13) or augmentations. Test accuracy per epoch degrades with worse image fidelity across our experiments (Figure 4.12), further highlighting that time-to-accuracy speedups are caused primarily by bandwidth reduction (rather than e.g., a form of regularization induced by lower scans).

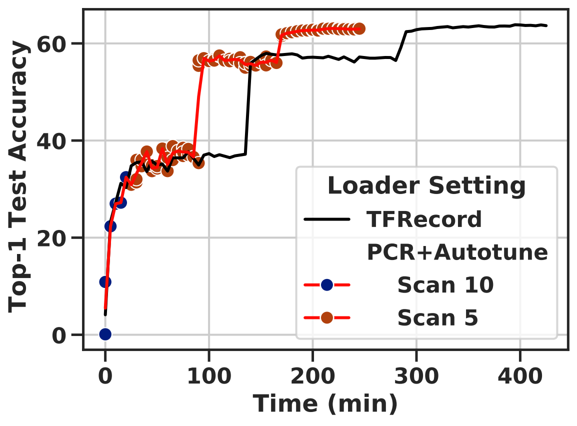
4.5.5 Autotuning Compression Level

Observation 7: *It is possible to automatically determine an appropriate level of compression at runtime by dynamically accessing various data qualities via PCRs.*

In cases where training resolution is not structured into learning [147] or image fidelity heuristics prove too costly to tune (§4.5.4), automatic tuning of the scan hyperparameter may be desirable. One way of doing this is by tuning with a measurement of the *bias* of the gradient given a lower fidelity image (§4.4.3), intuitively measuring how the model "sees" the image similarity. As we showed in Section 4.2, a similarity threshold of 0.8 or higher is sufficient to avoid bad



(a) ResNet



(b) ShuffleNet

Figure 4.14: Adaptive tuning on ImageNet for 90 epochs compared to TFRecord training, which is comparable to Scan 10 training. For adaptive tuning, epochs are marked with scatter points. Training is fastest after epoch 5, when autotuning search is enabled. Changing the threshold from 0.8 (shown) to 0.9 results in the last few epochs switching to scan 10.

scans throughout training—Figure 4.6 clusters low-quality scans below that point. We apply this threshold and the rest of the procedure described in Section 4.4.3 to the ImageNet dataset and observe that such autotuning repeatedly matches accuracy while almost being as fast as a pure scan 5 approach. The main slowdown is due to starting at scan 10 for the first 5 epochs of training, blending the latencies of scan 10 with those of scan 5. We note that, unlike MSSIM, which is statically concentrated above 95% for good quality scans, the gradient similarity changes over training. For example, ResNet18 has a similarity of 0.88 by epoch 85, whereas it had a similarity of 0.95 at epoch 5. We observe that using a higher threshold, approaching 0.9, forces scan 10 to be used for the last few epochs of training when gradient similarity is lower, retaining similar accuracy at slightly longer training times.

4.6 Related Work

Numerous works have explored methods for decreasing training time with large datasets [110, 136, 164, 167, 311, 315, 317], motivating a need for improved I/O internals [14, 32, 51, 56, 233], formats [234], caching [165], and data pipeline frameworks [211]. We discuss caching, forms of compression, and frequency-domain DL literature below.

Dataset Caching. Caching places data in faster storage tiers to offload the bandwidth burden from slower devices. ML applications lack locality due to uniform sampling [162], requiring either prohibitively large cache sizes or weaker forms of sampling [197]. However, when done correctly, caching can obtain significant speedups [206]. PCRs are designed for datasets which do not fit in caches, and, by virtue of accessing less data, can increase cache hit rates.

Dataset Cardinality Reduction. "Big data" spawned interest in dataset reduction techniques that aim to reduce the *number* of training samples while maintaining model accuracy [26, 65, 91, 146, 178, 191, 305]. Similarly, dataset echoing [53, 320] re-uses subsamples to speedup the data pipeline. PCRs differ in that they reduce I/O burden by modifying data representation and layout.

Dataset Sample Compression. Techniques such as compression [12, 185] or resizing [147] reduce data size by lowering fidelity, reducing I/O pressure. Prior work has shown that resizing as a form of data reduction is particularly effective for DL tasks, as resized data can speed up training, transfer to high fidelity test points, and in some cases, even *increase* accuracy when combined with certain data augmentations [52, 147, 280]. However, resizing parameters are chosen statically and heuristically (therefore suboptimally [280]), and thus may not meet the needs of all applications without duplicated work. PCRs differ in that they provide a *dynamic* mechanism for adjusting I/O load, and thus can adapt to both the system and task at runtime.

Similar to our work, MLWeaving [295] has shown that *transposed layouts* (i.e., column major) can accelerate machine learning training. However, this work differs in that we focus on I/O in the context of deep learning models, whereas MLWeaving focuses on memory bandwidth for general linear models. Additionally, the compression method differs. For image data, the three canonical dimensions of compression are 1) quantization, 2) frequency selection, and 3) spatial selection [290]; MLWeaving uses the first while PCRs use the first and second (via JPEG). More recently, transposed layouts have been used to reduce I/O in industrial settings for recommendation systems [322] by progressively ordering columns by access frequency. These techniques are complementary to ours as one can perform training over images and tabular data jointly, exploiting both domain-specific and statistical access patterns to reorder the reads.

Neural compression [278], which learns custom compression formats using neural networks, is an interesting direction for future work and is compatible with PCRs. However, while neural compression can outperform JPEG in terms of quality [278], it does so at significant cost. Using state-of-the-art neural compression [27, 28, 198], we find decoding to be between $900\times$ and $5000\times$ the cost of baseline JPEG, and thus incompatible with real-time performance.

General Compression. Bandwidth reduction extends to databases, memory hierarchies, and the web [10, 13, 223, 224, 327]. Progressive compression has been used in the context of dynamically saving bandwidth for mobile phone downloads [312]. HippogriffDB [176] uses GPUs to compress data in the context of databases, which lowers I/O bandwidth to get a speedup. Other work has investigated how image degradation affects inference [77, 225, 284, 323]. In contrast, our work is focused on compression for I/O savings in deep learning. Reinforcement learning has been used to choose JPEG parameters for cloud inference [175]; other work has hand-designed JPEG encodings for training [185]. These works are similar to ours in that they tune compression for the model, though they differ in that they are static. Other work investigates compressing models [20, 50, 73, 116, 117, 118, 132, 309] or network traffic [15, 179, 181, 297, 301, 321]; these are orthogonal to our work.

Frequency Domain Deep Learning. Prior work modifies models to directly train over compressed representations [97, 114, 279, 283] or with frequency-domain operators [85]; our work does not modify the model. Other work investigates generalization performance from the view of low [31, 310] and high [293] frequencies, which provides insight into our work. JPEG mostly filters low frequency components, and thus prior work has attempted to use JPEG as a defense mechanism against adversarial attacks [66, 86, 186]. Motivated by adversarial attacks exploiting spurious, high-frequency features [105, 133] other work investigates if frequency filters can impact model robustness [78, 314]; our work primarily focuses on retaining test accuracy under non-adversarial conditions.

4.7 Discussion

We introduce a novel storage format, *Progressive Compressed Records* (PCRs), to reduce the bandwidth cost of training over large datasets. PCRs use progressive compression to split training data into multiple fidelity levels, while avoiding duplicating space. The format is easy to implement and can be applied to a broad range of tasks dynamically. PCRs provide applications with the ability to trade off data fidelity with storage and network demands, allowing the same model to be trained with $2\times$ less bandwidth while retaining model accuracy. We introduce methodology for choosing the particular data fidelity necessary for a task, as well as a tuning heuristic that can be applied automatically. Using PCRs, our approaches can dynamically switch between multiple data fidelities while training without loss of accuracy. Future directions include alternative compression methods, data modalities, and hardware acceleration. In particular, hardware acceleration for image processing combined with lightweight compression algorithms seem to be the most promising path toward incorporating PCRs ubiquitously. In the next chapter (§5), we'll discuss how to generalize the I/O performance model to the full input pipeline.

Chapter 5

Automated Input Pipeline Configuration

To address the challenges that users face in configuring input data pipelines in ML jobs (§2), we introduce `Plumber`, an extensible tool that automatically detects and removes input data bottlenecks. From a modeling point of view, `Plumber` is a generalization of I/O techniques developed for PCRs (§4), combined with novel CPU-profiling and cache estimation techniques. The breadth and generality of the modeling allows a novel solver to be proposed and evaluated, eliminating a significant fraction of configuration for the user.

5.1 Plumber

Using a top-down approach, we start with the architecture of `Plumber` (§5.1.1) and then explain tuning methodology (§5.1.2). We later formulate the resource allocation problem across `Datasets` as a LP (§5.1.3), which relies on per-`Dataset` resource rates derived by `Plumber` (§5.1.4). `Plumber` is released as an open-source artifact and consists of a 3k line patch on top of `tf.data`'s `C++` `AUTOTUNE` infrastructure and 8k lines of `Python` interface.

5.1.1 Software Architecture

`Plumber` reasons about performance in a *layered* fashion. The goal of the layers is to abstract basic `Dataset`-level statistics into costs, which can be compared, optimized over, and extended. We demonstrate this architecture with a simplified, misconfigured ImageNet pipeline in Figure 5.1. This pipeline requires reading from `TFRecords` in parallel, decoding the examples of each record, and randomly augmenting the examples with crops and flips. Each of the pipeline components has a tunable, except for the `TFRecords`, which is parallelized by `Interleave`.

Tracing. By enabling a runtime flag, tracing collects `Dataset`-level statistics, such as counters for elements processed, CPU time spent, and the number of bytes per element. `Plumber` periodically dumps these statistics into a file along with the entire serialized pipeline program. Joining the `Datasets` with their program counterpart enables building an in-memory model of the pipeline dataflow. The bottom of Figure 5.1 shows how a `TFRecord` `Dataset` is traced, which reads a single `TFRecord` file. `Plumber`'s tracing instruments all `read()` calls into the filesystem within `tf.data`, allowing it to see all the reads into the 144MB file. Each record is

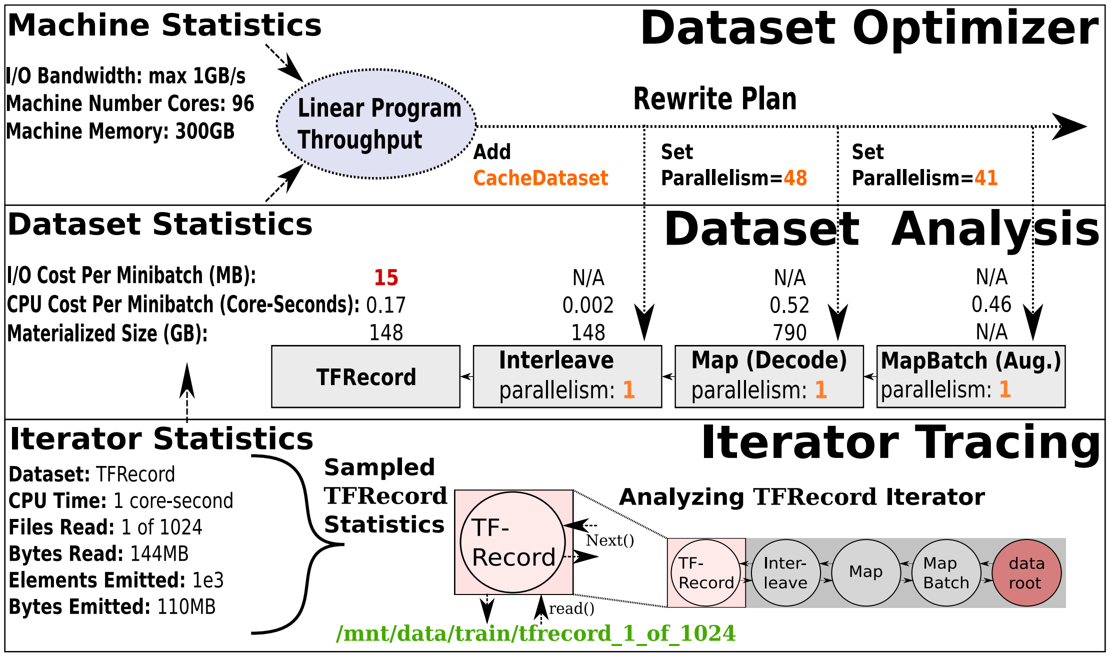


Figure 5.1: An ImageNet pipeline with Plumber's various states of processing. `Plumber` starts with `Dataset`-level tracing, which is then followed by analysis for CPU, disk, and memory costs, which are subsequently modeled and optimized. Reading a `TFRecord`, for example, is converted from bytes read to an I/O cost per minibatch, which can then be linked with available I/O resources to find bottlenecks.

unpacked into roughly 1200 elements (110kB images), which `Plumber` counts. By inspecting the serialized program, `Plumber` knows that there are 1024 total `TFRecord` files, and thus can estimate that the dataset contains 1024×1200 elements—ImageNet has 1.2 million. To get CPU usage, `Plumber` wraps a thread-level CPU timer around `Next` calls, which only counts *active* (not blocked) CPU-cycles. All `Next` calls are instrumented such that: 1) CPU timers stop when `Datasets` call into their children and start when control is returned and 2) statistics (e.g., counts and sizes) about each yielded element are attributed to the producer. The statistics necessary for optimization total less than 144 bytes per `Dataset`.

Analysis. To find bottlenecks, `Plumber` must analyze the traced data using analytical modeling, which puts each `Dataset` in units of cost that can be compared. `Plumber` treats the pipeline as a closed system, with each component operating asynchronously yet sharing the same resource budget. For example, we can see that `TFRecord` reads at a rate of 15MB per minibatch, but decoding consumes 1/2 core per minibatch. Determining which `Dataset` is the bottleneck depends on the resource allocation of CPU and I/O—for example, 30 minibatches per second can only be hit with 450MB/s of I/O and 15 CPU cores.

Optimizer. `Plumber` supports reasoning about CPU and disk parallelism, caching, and prefetch injection. Achieving optimal CPU and I/O parallelism requires allocating sufficient parallelism to keep the pipeline balanced in terms of throughput capacity, which we formalize in the LP. Caching reduces the amount of work done by avoiding re-computing data dependencies, and the optimal cache minimizes total work by placing it as high in the pipeline as possible. Prefetching is a subsequent pass which injects prefetching proportional to the idleness in the pipeline under a benchmark workload. As shown in the example, by placing caching at MapDecode, we can avoid all CPU and I/O operations except for the subsequent Crop and Batch at the expected cost of 790GB of memory. In this example, the optimizer knows that the machine only has 300GB of memory and thus it must settle with caching at the 148GB Interleave. By caching Interleave, Plumber can redistribute remaining CPU resources to other stages with the LP.

Extensions. Extending Plumber requires minimal effort by cleanly interacting with existing optimizations. For example, to add the ability to cache materialized results to disk in addition to memory, one can reuse all caching logic up to the cache decision itself, which would dispatch to in-memory caching preferably and disk caching if space and disk bandwidth allow it. Significant extensions (e.g., networking or GPU data transfer I/O) require adding the corresponding tracing and rates at the lower levels, which can then be incorporated as LP expressions and constraints. Two areas of future work are in extending Plumber's optimizer to reason about correctness optimizations, such as reusing data [53], and extending Plumber to perform optimal resource provisioning for matching a target throughput (e.g., to minimize cost).

5.1.2 Tuning Methodology

The intuition behind Plumber's performance debugging methodology is that input pipeline performance is limited by the Dataset with the lowest throughput and can be alleviated by adjusting the resource allocation (out of the available hardware resources) for this Dataset. However, simply maximizing the number of threads used to compute elements for a parallelizable Dataset is not always productive, as threads compete for CPU and memory resources. The end-to-end throughput of the input pipeline can also be limited by I/O bandwidth. Caching Datasets in memory alleviates CPU or I/O bottlenecks at the cost of memory capacity. As all Plumber traces are also valid programs (that can be rewritten), Plumber simply requires a way for a user to mark the program for tracing, and thus one entry point is sufficient for Plumber to accomplish all tuning.

Tuning Interface. Plumber introduces *tuning annotations* to add on top of existing data loading code. A data loading function is one which returns a tf.data Dataset, which has an associated signature. Tagging the code with an @optimize annotation gives Plumber permission to intercept one or more Datasets and return an optimized variant matching the Dataset signature. The annotation provides Plumber with an entry point into the loader allowing Plumber to trace it under a benchmark workload, and rewrite it before passing it back to the application. An example of the API can be seen in Figure 5.2.

We note that there are cases where two query algorithms implement the same logical goal, with different performance characteristics, and thus a plain annotation is not sufficient to expose all optimizations. For example, JPEG files can be decoded then randomly cropped, or the two can be fused. The tradeoff is the former is amenable to caching (after decode), while the latter is faster to decode—therefore, the optimal implementation depends on runtime conditions (namely, memory capacity). To get around Plumber's lack of a logical query system, Plumber allows users to specify multiple pipelines, each with the same signature. In practice, this is on the order of two pipelines (keeping the choice fast), and Plumber will automatically trace both and apply any optimizations it can before picking the fastest pipeline. The optimization shown in Figure 5.2 is a difficult optimization to perform for an online tuner, like AUTOTUNE, because, even with

```python
@plumber.optimizer(pick_best={"cache": [True, False]})
def loader_fn(data_dir, cache):
    ds = my_dataset(data_dir)
    if cache:
        return ds.map(decode).map(crop)
    else:
        return ds.map(fused_decode_crop)
```

Figure 5.2: `Python` pseudo-code for optimizing a pipeline via annotations. `Plumber` will trace both caching paths and pick the one best suited for runtime conditions (e.g., memory allows caching). We note that the cached-path will typically have user-defined caching, though `Plumber` discards such performance-optimizations as *suggestions* and inserts them itself, if possible, for each of the different code-paths—avoiding memory errors and duplicate caches.

both pipelines to inspect, the effect of caching does not kick in until a *whole epoch into the training*. `Plumber`, knowing that cache cold-start is a factor, can simulate the steady-state effects of caching by truncating the cached data with advanced rewriting, yet still return the normal dataset. We use these annotations for our end-to-end ResNet experiments, which feature the fused decode and crop example in the figure.

Modeling. `Plumber` maximizes throughput by modeling the input pipeline as an asynchronous closed-system in the context of operational analysis [72], which explicitly defines bottlenecks. The operational framework has few statistical assumptions, unlike Markovian queues, and parameterizes each component of a system with a cost relative to the resource usage, usually expressed in units of time. `Plumber` further measures the resources and traces the analytical network to automatically "operationalize" and tune an arbitrary input pipeline.

5.1.3 Allocating Hardware Resources

In order to derive an improved input pipeline configuration, we need to understand how each `Dataset`'s performance is affected by changing the fraction of hardware resources allocated to it, e.g., by changing the degree of parallelism of a `Dataset` or inserting prefetching or caching `Datasets` which consume extra memory. We are interested in characterizing the usage of three types of resources in input pipeline execution: CPU, disk bandwidth, and memory capacity. We first formulate a Linear Program (LP) to solve for the optimal CPU resource allocation before moving onto disk and, finally, memory capacity. The calculation of the inputs into this LP are discussed in the following subsection (§5.1.4).

CPU. For CPU optimizations, we optimize over a `Dataset` *tree* in the input pipeline to decide *what fraction* of the CPU-time each `Dataset` should receive such that throughput is maximized. Having two `Datasets` in series with rate R_1 and R_2 yields an aggregate rate of $X = \min(R_1, R_2)$. The bottleneck `Dataset` determines performance. For example, in Figure 2.2, if `Map` has rate R_1 and `Batch` has rate R_2, to increase X, we must assign the slower of the two `Datasets` more resources. However, the above is only true *if* we can parallelize the bottleneck node and *if* we have resources left.

We optimize $\text{Max}_\theta\,[X = \min_{i \in \mathbb{D}}[\theta_i * R_i]]$ subject to constraints: $\sum_{i \in \mathbb{D}} \theta_i \leq n_c$; $\theta_{i \in \mathbb{D}} \geq 0$; $\theta_{i \in \mathbb{S}} \leq 1$. In the above equations, $\mathbb{D}$ is the set of `Datasets` in consideration, $\mathbb{S} \subseteq \mathbb{D}$ a subset of sequential `Datasets`, n_c the total number of cores, X throughput, R_i the measured rate of minibatches per second per core, and θ the fractional number of cores allocated. The equation maximizes the input pipeline throughput X by maximizing the aggregate rate ($\theta_i * R_i$) of the slowest `Dataset` (the minimum). We constrain sequential `Datasets` to have at most one core, and we cannot exceed all cores on the machine.

Disk. As shown in Figure 2.2, data flows from disk (`TFRecordDataset`) into the CPU section of the pipeline. Therefore, if the data source is slower than the rest of the pipeline, the entire pipeline is disk-bound. For large reads (e.g., records), there are two factors that can cause a disk bound: 1) insufficient read parallelism and 2) max disk I/O bandwidth. The latter can be found via benchmarking tools [25], but `Plumber` goes a step further by benchmarking entire empirical parallelism vs. bandwidth curve for a data source (via rewriting). The source parallelism results can then be fit with a piecewise linear curve to be injected into the optimizer to determine a minimal parallelism to hit max bandwidth.

Memory. Caching aggressively is always desirable, because it allows I/O to be entirely reduced and some CPU processing to be partially reduced. The potential speedup grows as one goes up the pipeline, but so does the memory requirement to cache, which may exceed system memory. `Plumber` checks if operators are random functions (e.g., augmentations)—randomized functions have infinite cardinality and cannot be cached. However, if an operator's output is finite, `Plumber` estimates the materialized size.

To check for randomness, we are interested in the relation: if a function, f, accesses a random seed, s: $f \rightarrow s$. The transitive closure, $f \overset{+}{\rightarrow} s$, measures if *any* child functions of f touch a random seed. If $f \overset{+}{\rightarrow} s$ is true, then we cannot cache f or any operations following it. This simple relation can be computed via a graph traversal and holds nearly always, as random seeds are necessary in implementations that enable determinism for reproducibility. To solve memory caching for typical, linear-structure pipelines, `Plumber` uses a greedy (yet optimal) approach to select the `Dataset` closest to the root that fits in memory. `Plumber` can also solve for more generic topologies by adding Boolean decision variables for each cache candidate over the LP already presented.

5.1.4 Resource Accounted Rates

The LP construction (§5.1.3) assumes the existence of calculated rates (for CPU and disk I/O) as well as the materialized size (for caches), and uses those values as inputs into the algorithm. This section explains how those values are calculated by using resource accounted rates. Resource accounted rates encompass the cost of an operation or decision to cache in the pipeline. For CPU and I/O, the cost is the ratio of CPU core-time or I/O bytes per minibatch, which are *throughput bounds*. Memory capacity measures the cost in terms of bytes required for materialization at a particular `Dataset`, and is a *throughput optimization*. The middle of Figure 5.1 illustrates all three of these costs. The full algorithm for resource accounted rates can be found in the appendix; however, we give a brief description here.

Common Units. The root of the pipeline gives a common set of units for CPU and I/O: minibatches. Children of the root do not necessarily output elements in terms of minibatches (e.g., prior to batching); thus, a conversion factor between an arbitrary `Dataset`'s elements and that of the root must be calculated—this is called the “visit ratio”, V_i, and represents the mean number of completions at `Dataset` i for each completion from the pipeline. To calculate V_i, we start with the pipeline’s root “visit ratio” $V_0 := 1$. Then, we apply the following recurrence: $V_i = (C_i/C_{i-1}) \times (C_{i-1}/C_0)$, where C_i is the average number of items of work completed at `Dataset` operation i. The former ratio is the `Dataset`’s local input-output ratio, and the latter is calculated in the recurrence. Intuitively, the visit ratio allows one to say that n elements are in a batch, and thus dependencies to batching must have a throughput n times faster to “keep up”.

Throughput Cost. The throughput at the root, X_0, is the number of minibatches completed, C_0, in a timeframe, T, and the child `Datasets` have $X_i = V_i X_0$. However, this equation does not explain the bottleneck cause, which requires reparameterizing the throughput in terms of CPU core time or I/O bytes. As $X_i = C_i/T$, we factor the equation into 1) the product of completions per resource (e.g., elements per core-seconds) and 2) resource per time (core-seconds per time). The former is the ratio of two traced variables (element completions and CPU-time or bytes used), and the latter is a knob for modeling adding or removing resources (e.g., CPU parallelism or extra bandwidth). In the LP (§5.1.3), R_i is the first factor normalized by V_i, and θ_i is the second factor; in a bottleneck, they determine X_0.

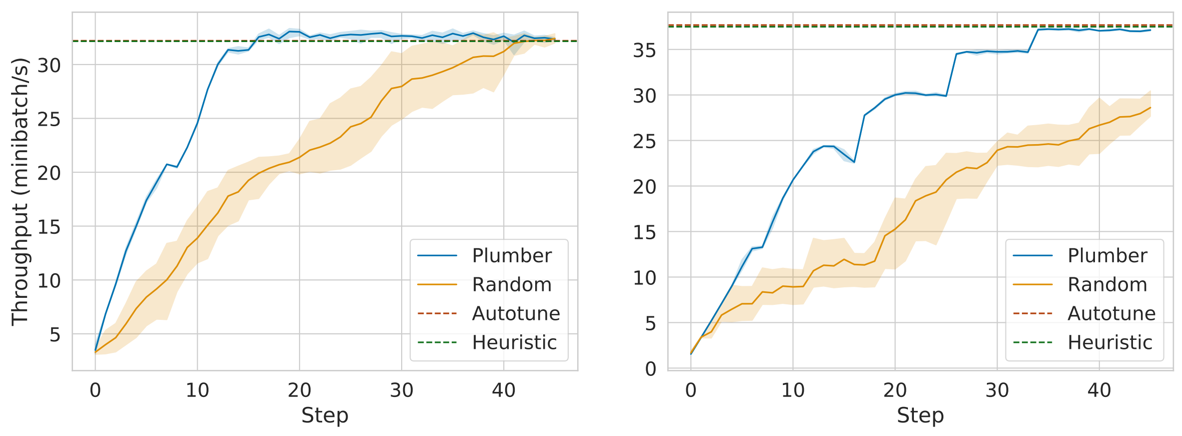
Materialization Cost. Estimating the size of a `Dataset`’s materialized artifacts is similar to the prior operational treatment but involves propagating the estimates *up* from data source to root. The materialized size of a data source is the product of 1) the number of elements (cardinality) and 2) the average size of each element. Both are necessary because a `Dataset`’s semantics may modify one or the other; for example, truncation only modifies the former and decompression only the latter. To start, the size of a data source is the number of files, n, times the average bytes per file, $\bar{b}$. Propagating the number of elements, n_i, involves multiplying n by an input-output completion ratio. The sum of output bytes and the number of completions for each `Dataset` is measured in tracing, and thus $\bar{b}_i$ is readily computed. n_i can grow unbounded (and thus uncacheable) if the data is infinitely repeated or augmented. `Datasets` that are children to a cache can be modeled as having no steady-state cost (e.g., after the first epoch).

5.2 Evaluation

We evaluate CPU bottleneck removal in §5.2.1, showing that `Plumber` can indeed find bottlenecks, and we further analyze how `Plumber`’s solutions differ from those of baselines. We additionally evaluate disk and caching in §5.2.2 and §5.2.3. End-to-end results are presented in §5.2.4. `Plumber`’s overhead on modern hardware is less than 21% (due to text workloads) and drops to below 5% on vision workloads. We compare against a naive configuration, which has minimal parallelism, to two strong baselines: `AUTOTUNE` [211] and `HEURISTIC`, which set the parallelism tunables to the number of cores on the machine.

Hardware. For microbenchmarks, we evaluate over two setups to ensure our results generalize. Setup A is a consumer-grade AMD 2700X CPU with 16 cores and 32GiB RAM. Setup B is an older enterprise-grade 32–core Intel E5–2698Bv3 Xeon 2GHz CPU with 64GiB RAM. Setup C is for end-to-end results and is a TPUv3-8 [140] with 96 Intel Xeon cores and 300GB of RAM.

Workloads. Our evaluation uses the MLPerfv0.6 subset of MLPerf training [192] benchmarks, which is representative of datacenter workloads and covers both images and text. We use the



(a) ResNet (Setup A) (b) ResNet (Setup B)

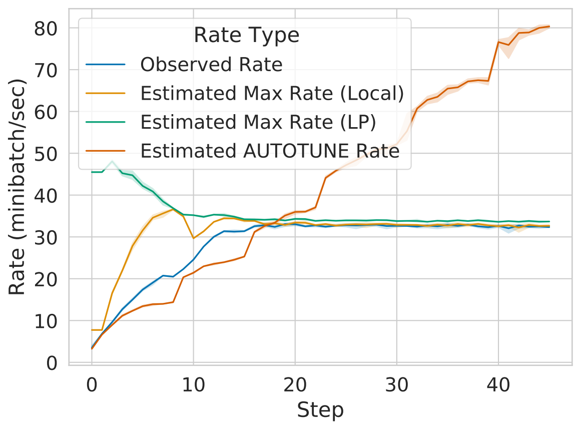
Figure 5.3: `Plumber` outperforms random walks by 2–3×, demonstrating that `Plumber`'s signal is markedly better than guessing. X-axis denotes the optimization step, starting at minimal parallelism. Y-axis denotes the pipeline rate in minibatches per second with 95% confidence intervals.

following tasks and datasets: **ResNet/ImageNet** [71, 123], **Mask RCNN/COCO** [180, 243], **MultiBoxSSD/COCO** [183], **Transformer/WMT** [258, 285], and **GNMT/WMT** [92, 306].

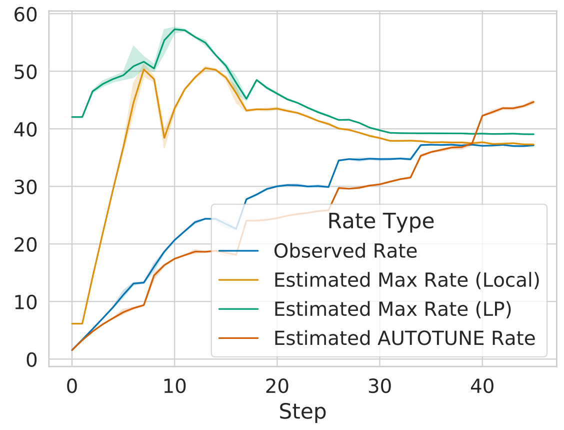
5.2.1 CPU Bottleneck Removal

To assess how accurately bottlenecks are found, we use `Plumber`'s analysis layer to rank nodes by bottleneck. The pipeline's parallelism parameters are initialized to the naive configuration (parallelism=1) with prefetching, and `Plumber` iteratively (using 1 minute of tracing) picks the node to optimize by ranking nodes by their parallelism-scaled rates. To compare against uninformed debugging, we plot a random walk, which randomly picks a node to parallelize for each "step" (x-axis in our plots). We run each experiment three times to get confidence intervals.

Sequential Tuning. Figure 5.3 shows the ResNet workload across both setups; other workloads look similar. `Plumber` is consistently better than the random walk, as expected. We observe that both `HEURISTIC` and `AUTOTUNE` are equivalent in terms of reaching peak performance—over-allocation does not usually result in performance degradation. On the ResNet workload, the bulk of the work is in the JPEG decoding `Dataset`, which services 2.5 minibatches/second/core on Setup A. Most of Setup A's steps are spent increasing the parallelism of this `Dataset`, a transpose operation being the second bottleneck. The bumps in Setup B correspond to increasing the parallelism of Transpose rather than JPEG decoding and occur roughly once every 8 steps. We observe that, across all pipelines, such transition regions are the only regions (in addition to fluctuations at convergence) where `Plumber` struggles to characterize the locally optimal decision (see MultiBoxSSD example Figure 5.7). While Setup B has 2× more cores than A, the per-core decoding rates for B are lower, resulting in only a 1.2× higher throughput.



(a) ResNet (Setup A)



(b) ResNet (Setup B)

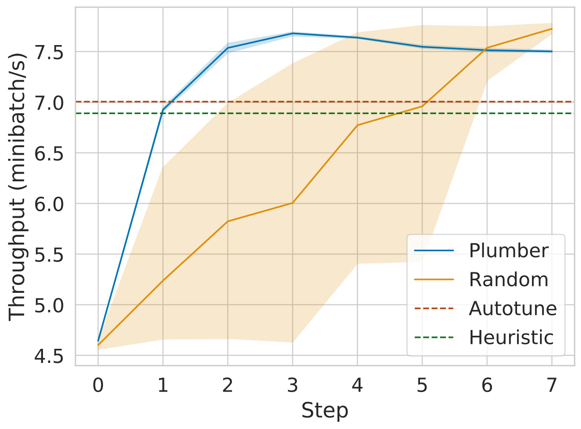
Figure 5.4: Before optimizations begin, `Plumber` is able to bound performance within $2\times$ with the LP, and the gap decreases over time. Setup B exhibits superlinear scaling around step 10 and also exhibits more pronounced bottlenecks, as it takes longer to converge. The `AUTOTUNE` model does not account for saturation and therefore has unbounded predicted throughput.

Observation 8: *`Plumber`'s bottleneck finder converges to the optimal throughput in 2–3$\times$ fewer steps than a random walk. Inspecting the individual `Dataset` rates offers pipeline and machine performance insights.*

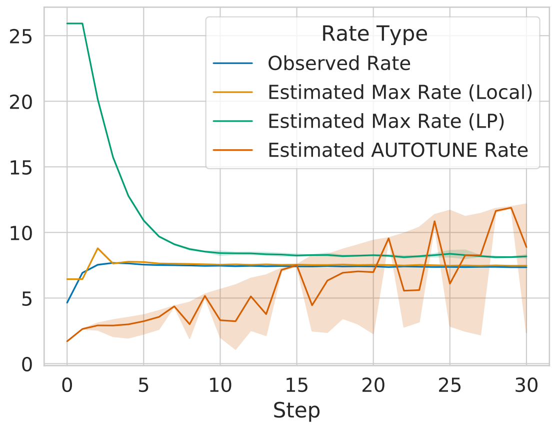
Linear Programming. Figure 5.4 demonstrates that `Plumber` can understand performance through the LP formulation on ResNet; other workloads are similar. As a baseline, we include a "local" method, which allocates all remaining resources to the current bottleneck node. This baseline is unable to see past one bottleneck and thus oscillates as the bottleneck node changes (the "bumps" in Figure 5.3). Meanwhile, the LP steadily declines—upon inspecting the solution, we find a strong correlation with the LP's decline and the value of R_i for the bottleneck node (JPEG decoding). While R_i typically decreases (e.g., due to scaling overhead), we find it *increases* briefly especially on Setup B, peaking at step 10 and resulting in a "bell shape" LP prediction (JPEG rate peaks at 1.8 and then drops to 1.4 by the end of training), explaining the peak in the LP. `AUTOTUNE`, being oblivious to saturation, either overestimates or underestimates throughput without bounding it.

Observation 9: *`Plumber`'s LP solution captures both resource utilization and bottlenecks, bounding throughput to within $2\times$ from when optimization starts for pipelines like ResNet and MultiBoxSSD. The bounds get tighter as optimization proceeds due to differences in empirical rates.*

Large UDF Parallelism Challenges. As shown in Figure 5.5, RCNN on Setup A is challenging and displays counter-intuitive behavior. The reason is that RCNN features a large UDF in its `MapDataset`, which is transparently parallelized by the `TensorFlow` runtime. Parallelizing the `Map` is dangerous because the parallelism compounds with UDF parallelism—1 parallelism uses nearly 3 cores! As parallelism increases for this `Dataset`, the corresponding per-core



(a) RCNN Convergence



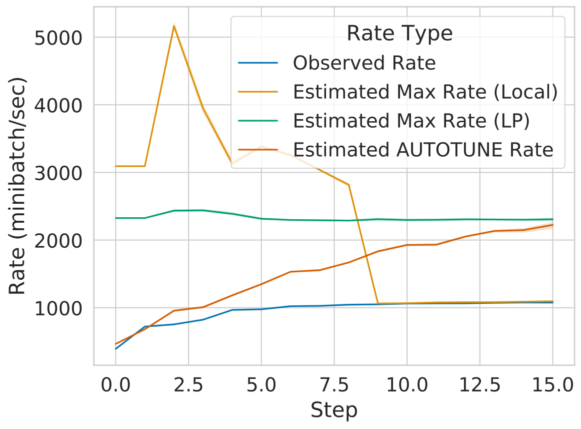
(b) RCNN Predictions

Figure 5.5: RCNN on Setup A, along with its predictions. RCNN exhibits heavy UDF parallelism, which causes thread over-allocation to quickly deteriorate performance. `AUTOTUNE` has high variance estimates of latency.

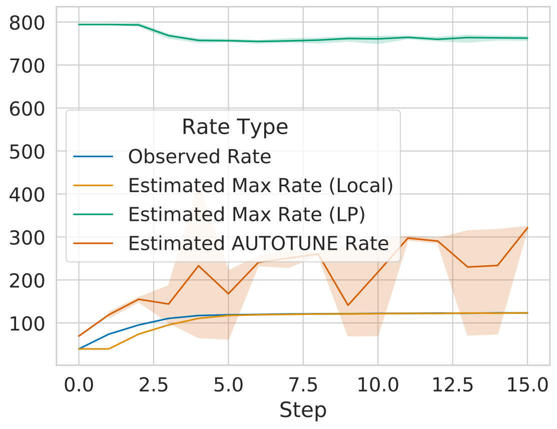
rate drops, causing the LP prediction to drop and baselines to overshoot peak performance on both setups. While the LP overestimates peak performance by $4\times$, it is still qualitatively better than `AUTOTUNE`, which oscillates in predictions. Upon inspection, `AUTOTUNE` allocates 16 parallelism to the bottleneck node, but 3 parallelism to a different `MapDataset` node. The bottleneck node operates at 0.5 minibatches/second/core, while the other node operates at 20 minibatches/second/core. Thus, the optimal policy, which `Plumber` follows, is to only allocate parallelism to the main bottleneck (thus bounded by $0.5 * 16 = 8$). In fact, due to UDF parallelism, only 4–5 parallelism is necessary. Counterintuitively, this policy is no longer optimal for our end-to-end results (§5.2.4), which have $6\times$ more cores.

Observation 10: *`AUTOTUNE` and `HEURISTIC` are vulnerable to over-allocation, which can cause performance degradation. Pipelines with heavy UDF parallelism may experience drops on the order of 10%. Dynamic parallelism makes end-to-end performance hard to predict.*

Characterizing Local Behavior. While `Plumber` may be able to converge to a good solution 2–3$\times$ faster than an uninformed user, it is worth knowing how *optimal* are `Plumber`'s predictions. While we cannot enumerate all possible convergence paths, we can test if there was a better `Dataset` selection to be made by `Plumber`. To test this, we sample three one-step deviations from `Plumber`'s recommended action. We highlight MultiBoxSSD in Figure 5.7, which exhibits many transitions between bottlenecks. For MultiBoxSSD, we observe similar bottleneck behavior as ResNet; `Plumber` prioritizes the image processing operation and alternates to the `TFRecord` parsing operation every 4 steps (nearly twice as often as ResNet cycles). For ResNet, we observe local optimality except at the bottleneck transitions (once every 8 steps). For the other datasets, we don't observe any significant mispredictions before convergence. Across the datasets, we observe that both transition regions and resource saturation cause measured rates to become *correlated*—the rates begin spiking/oscillating together, which makes ranking `Datasets` unpredictable On ResNet, for example, we observe oscillations begin at 90% of peak performance. Therefore, choosing the bottleneck node among similarly bottlenecked nodes is

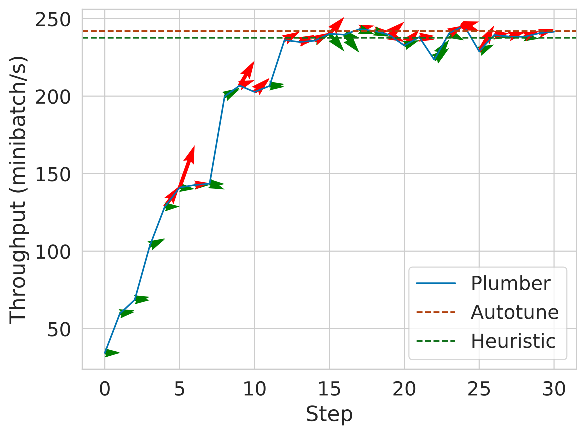


(a) Transformer Predictions

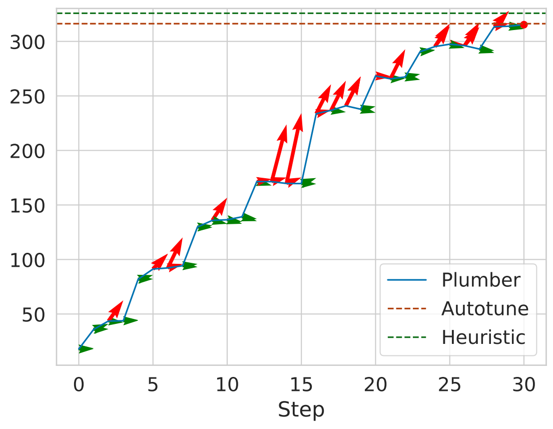


(b) GNMT Predictions

Figure 5.6: Transformer and GNMT predictions on Setup A. Transformer and GNMT exhibit small operations, which are a mismatch to the `Iterator` model. It is difficult to fully saturate a CPU with `Dataset` parallelism alone—caching or outer parallelism are required.



(a) MultiBoxSSD (Setup A)



(b) MultiBoxSSD (Setup B)

Figure 5.7: MultiBoxSSD with local perturbations. `Plumber` is able to find optimal `Dataset` choices when `Datasets` are different with respect to performance. MultiBoxSSD exhibits two bottlenecks alternating every 4 steps, resulting in confusion at the steps. There are diminishing returns to pursuing an existing bottleneck, which slightly lag behind the rates measured.

ambiguous. For workloads with larger dynamic ranges between slower nodes, performance is nearly always optimal.

Observation 11: *`Plumber`'s model tends to accurately predict the current bottleneck operator until transitions between bottlenecks. Bottleneck transitions and resource saturation are inherently difficult to model in terms of the current bottleneck.*

Text Processing (NLP) Challenges. We observe that both Transformer and GNMT are difficult to optimize in practice. As shown in Figure 5.6, both pipelines are predicted to be 2–8× faster than they actually end up being. Upon investigation, we observe that nearly all operations in NLP are very small e.g., grouping a few integers in a vector. The operations are so small that they are significant compared to the `Iterator` abstraction's overhead, causing idle "bubbles".

According to `Plumber`, GNMT is bottlenecked by `ShuffleAndRepeatDataset`; this `Dataset` is performing minimal work and thus the result is unexpected. Before getting to this point, `Plumber` allocated 9 parallelism to the first `MapDataset` and then gave up upon seeing the non-optimizable `Dataset`. Similarly, for Transformer, after alternating optimizing the 3 `MapDatasets` (all similarly fast), `Plumber` indicates Transformer is bottlenecked by `FilterDataset`, which is operating at about half of its max rate (explaining the $2\times$ difference). Using outer-parallelism for both pipelines, introducing inner-parallelism for `Batching` (GNMT), and resuming `Plumber`'s optimization results in closing the predicted performance gap to less than $2\times$ (Transformer) and $4\times$ (GNMT).

Observation 12: *Text pipelines can require significant tuning to overcome framework overheads. An in-memory cache materializing results is ideal, when possible.*

5.2.2 Disk Microbenchmarks

To simulate various bandwidths, we implement a token-bucket bandwidth limiter in the `TensorFlow` filesystem layer and use the most I/O intensive pipeline, ResNet. `Plumber` correctly concludes that ImageNet records are approximately 110KB on average and infers that 128 records are necessary for 1 minibatch. It thus infers that the I/O load per minibatch is $128 * 110$ KB, or 6.9 minibatches per 100MB/s of bandwidth, which it can join with known bandwidth (e.g., from token-bucket or `fio` [25]). Using this bound, `Plumber` predicts the pipeline's performance within 5% from 50MB/s to 300MB/s, when the compute bottleneck begins. We observe similar results for RCNN and MultiBoxSSD, though MultiBoxSSD is easier to bottleneck consistently due to its faster CPU speed. `Plumber` estimates that RCNN and MultiBoxSSD can push 145 minibatches/sec on 100MB/s, since they use the same dataset and same batch size; therefore, MultiBoxSSD is $25\times$ more I/O bound for a fixed CPU.

We then run this experiment on setup B with a real HDD (Seagate ST4000NM0023) and NVMe SSD (400 GB Intel P3600), which have 180MB/s and 2GB/s of read bandwidth, respectively. We run the load for one minute to prevent the page cache from kicking in (when the dataset reads repeat). For ResNet, `Plumber` predicts 11.06 minibatches/sec, and we are bound at 12.7 (15% error). On the NVMe SSD, `Plumber` predicts 135 minibatches/sec, and, indeed, we observe a compute bound. On RCNN, `Plumber` predicts disk bounds of 970 and 11850 minibatches/sec, respectively; for both, we observe the compute bound of 14 minibatches/sec.

On HDD with MultiBoxSSD, Plumber predicts 235 minibatches/sec, whereas we observe 215 (10% error). On NVMe SSD with MultiBoxSSD, Plumber predicts a disk bound of 2900 minibatches/sec; we observe the compute bound being hit before the drive is saturated. The text datasets are too small to test.

Observation 13: *Plumber is able to bound disk-bound workloads to within 15% of the observed throughput, notifying users of potential hardware misconfigurations.*

5.2.3 Memory (Cache) Microbenchmarks

We evaluate Plumber's predictions on memory optimization and summarize them below. Plumber predicts that 148GB are necessary to cache ImageNet for ResNet; 20GB to cache COCO for RCNN and MultiBoxSSD; and 1GB and 2GB for the Transformer and GNMT WMT datasets, respectively, which matches their known size. While this is expected for a full sweep of the training set (by simply tracking all file sizes), it also holds for sampling the dataset. Empirically, we find that observing a small subset of the data is sufficient: 1% of files provide a relative error of 1% for ImageNet and 2% for COCO, and 5 files gives less than 2% relative error for WMT datasets.

For materialized caches, we observe that Plumber predicts no changes until a node is hit, which changes the bytes/element. For ImageNet, we observe that, if Plumber is fed a fused decode and crop pipeline, it predicts caching is only possible at the source, because the crop is random. However, when the ImageNet pipeline is not fused, image decoding amplifies the dataset size by $6\times$, which Plumber observes as 793GB of a true 842GB, or 6% error with 60 seconds of profiling. For this pipeline, we observe that relative error decreases as a function of tracing time, saturating at 2 minutes and yielding relative error rates below 1%, offering a knob for refining estimates at the expense of tuning time. RCNN can only be cached at the disk-level, since the following UDF is randomized. For MultiBoxSSD, Plumber detects it takes 84GB of a true 97GB (14% error) to materialize the dataset in memory after image decoding with only 60 seconds of profiling, with 2 minutes yielding a 5% error, which continues to drop by $\sim$1% for each additional minute. Plumber additionally detects that the filter in the pipeline reduces the dataset by less than 1%.

Observation 14: *Plumber captures dataset sizes at the source exactly, and, for large datasets, it is able to subsample 1% of files to obtain 1% error. For materialized caching, Plumber propagates changes to dataset sizes (e.g., data decompression and filtering).*

5.2.4 End-to-End Pipeline Optimization

The end-to-end benefits of Plumber's optimizations, evaluated over 5 epochs of training, are shown in Figure 5.8. None of the pipelines have caches inserted manually, naive configurations have 1 parallelism and no prefetching, and HEURISTIC uses the prefetching hard-coded into the dataset. We observe overprovisioning (HEURISTIC) is competitive, if not faster, than AUTOTUNE across all trials. Plumber can go beyond both of these strong baselines because of caching, obtaining up to a $47\times$ speedup. Such a speedup, an absolute throughput of over

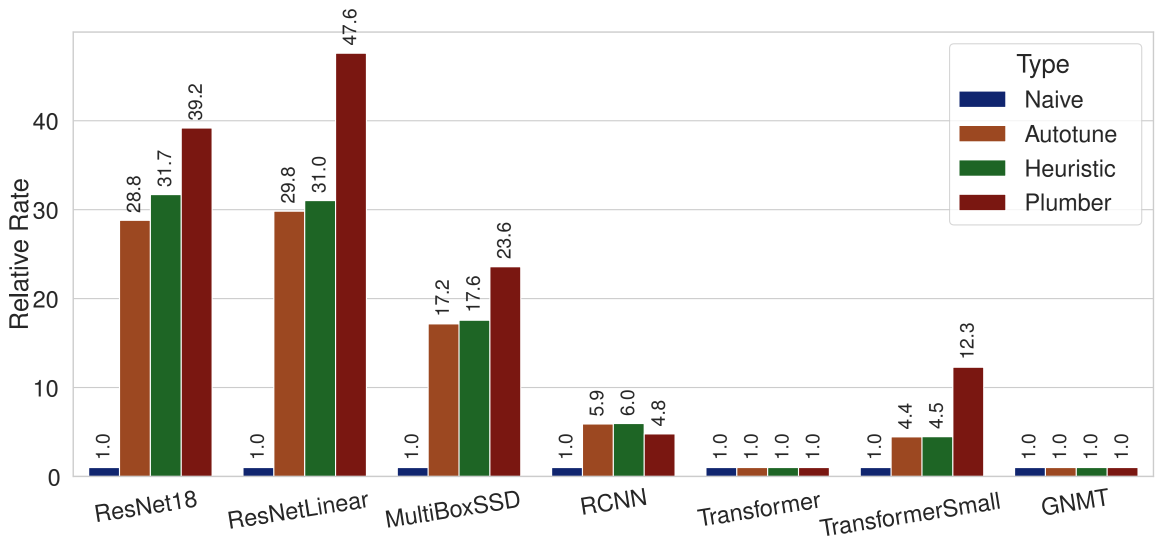


Figure 5.8: Relative speedups over the naive configuration for end-to-end model training on TPUv3-8. Apart from RCNN, `Plumber` surpasses strong baselines by adding caching, yielding speedups of up to $47\times$ compared to naive and 50% compared to tuners. While text pipelines are challenging to improve on MLPerf, using a smaller Transformer combined with a more complex pipeline can increase the tuners gap to $2.5\times$.

14k images/second, is only possible because caching bypasses the (`Plumber`-derived) 11k image/second data source bottleneck for the cloud storage. For ResNet-18, it is sufficient to cache at the data source, which is only 148GB; therefore, `Plumber` picks the CPU-optimized branch of the pipeline. However, when we use a linear model for ResNet, we use the smaller validation set, which allows `Plumber` to cache the $6\times$ bigger decoded images in memory, avoiding a CPU bottleneck. We also evaluate ResNet-50 and find that `Plumber` obtains a $24\times$ speedup over the naive configuration, though it cannot improve over other baselines, as the model's throughput limits are hit at 8k images/second.

Compared to `AUTOTUNE`, `Plumber` obtains a 36–59% speedup on three of the MLPerf benchmarks and ties on Transformer and GNMT (due to model throughput). These numbers are conservative—we observe that `AUTOTUNE` on ResNetLinear often sets the I/O parallelism to 1, which results in a 35% performance drop compared to what is shown in Figure 5.8 and creates a $2.4\times$ performance gap between `AUTOTUNE` and `Plumber`. To allow more competitive tuning from `AUTOTUNE` on that workload, we set the I/O parallelism to the default value of 10 used in MLPerf submissions, which improves its final performance. For RCNN, `Plumber` may perform worse than `AUTOTUNE` because `Plumber` is conservative in its parallelism allocation, while `AUTOTUNE` tends to allocate maximum parallelism to all `Datasets`. `Plumber` estimates that one `MapDataset` is two orders of magnitude more expensive than the other. In some cases, `Plumber` allocates 95 parallelism to the former, leaving only 1 parallelism for the remaining `MapDataset`, which results in the shown 23% performance loss. However, we also observe cases where `Plumber` matches the throughput of the baselines because it allocates at least 2 parallelism to the cheaper `Dataset`, which removes the bottleneck—suggesting that a mild form of "hedging" would be effective in preventing under-allocation for such skewed workloads. For MultiBoxSSD, `Plumber` is able to materialize the data after filtering is performed, which makes the cache smaller and increases throughput by removing load from the CPU. While we don't see a large benefit from `Plumber`'s optimizations on the MLPerf NLP pipelines, we do see improvements when moving to the related official `Flax` implementations of Transformer when it is configured to use a single-layer Transformer (TransformerSmall). We see a $2.5\times$ difference between strong baselines and `Plumber` on TransformerSmall because the `Flax` pipeline performs text-processing and packing on-the-fly, which, when combined with a smaller model, makes peak throughput only achievable with aggressive caching.

Observation 15: *`Plumber` frees the user from making individual caching and parallelization decisions, enabling 50% end-to-end improvements over `AUTOTUNE` and heuristics and $40\times$ improvements over naive configurations.*

5.3 Related Work

Pipeline Optimization. Dataset Echoing [53] repeats input pipeline operations to match compute rate. DS-Analyzer predicts how much file cache memory is necessary to match the compute steps [206]. Progressive Compressed Records [158] match compression levels to likewise minimize I/O. Each of these works is characterizing a piece of the input pipeline, all of which can be homogenously dealt with via `Plumber`—the first is a visit ratio and the latter two are memory caching/disk.

Systems have also been built to natively support the data pipeline workload. A cache library has been developed to carefully partition and coordinate caches in distributed training [206] Dataset echoing has been further expanded to support caching partially augmented samples [174]. Filesystem middleware has been developed to natively support data prefetching [83]. While `Plumber` primarily focuses on native `tf.data` primitives, it could also be used to hook into such systems to perform cross-system optimization.

Bottleneck Detection. Tools in the big-data domain (e.g., Spark [319]) have similar performance problems as those found in ML pipelines [219, 220], but the differences in domain encourage a different approach. Notably, Monotasks [220] enables bottleneck finding by designing Spark primitives to be easily measured on a per-resource level. In contrast, `Plumber` does not modify the framework and instead elects to carefully instrument selected resource usage, though the design is similarly simplified by having resource-specialized operations. Big data systems, such as DyradLINQ [318], have similarly benefited from dedicated debugging utilities [135] and dynamic query rewriting [148]. Roofline [45, 302] models bound compute kernels with CPU limits. `Plumber` generates similar plots using `Dataset` and resource limits.

Recent work has also characterized the design space of input pipelines with a profiling tool [134]; we do not focus on general pipeline decisions (e.g., the choice of image resolution), but rather focus on automatic tuning of performance knobs. These works all highlight a growing need for systems which better support the input pipeline workload.

5.4 Discussion

Today, it is common for input pipelines to be manually configured to better utilize their hardware and improve training performance. Motivated by evidence of data pipeline misconfiguration in practice (§2), this chapter introduces `Plumber`, an extensible tracing and optimizing tool. `Plumber` localizes bottlenecks to an individual operator by tracing both the resource usage and rate of each operator, enabling both bottleneck-finding as well as inferences on future performance. By adding caching in addition to tuning, `Plumber` can surpass the end-to-end performance of existing tuners by 50%. Future extensions to `Plumber` include: distributed and accelerator-infeed level profiling, optimal resource allocations for pipelines, and semantic-level re-writing of augmentations. In the final chapter (§7), we discuss generalizations of `Plumber`'s model as well as what system designers can learn from experience with `Plumber`.

Chapter 6

Validation Systems for Language Models

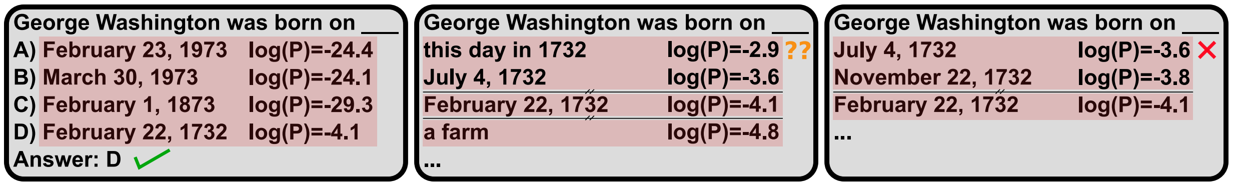
6.1 Validating Language Models with Regular Expressions

Large language models are being increasingly used as a way to better understand and generate natural human language. Language models need to be validated, because they express surprising biases in their learned behavior (e.g., learning private information or hate speech). Unlike the trends in vision and recommendation systems, which have been successful with moderately-sized models, these language models are only continuing to grow, making them train-once and reuse-often. Such *foundation models* [40] have new properties: 1) they must be deployed and transferred in a variety of contexts, 2) they can only be trained a few times due to cost, and 3) they will continue to showcase surprising behavior well-after they are originally evaluated. In turn, training is essentially a special case, and most time will be spent wrangling with the errors and biases of the resulting model.

This chapter describes a general-purpose mechanism through which language model validation tests can be built by exposing a general "search" interface to language models. This chapter therefore shifts the thesis from a focus on training data to that of validation data. The proposed mechanism provides an efficient and reusable abstraction capable of testing a wide range of models—leveraging the powerful yet well-understood theory of regular grammars. Additionally, the mechanism is implemented at decode-time, which may allow it to be used to steer models out of bad behavior at runtime.

6.2 Introduction

Large language models (LLMs), such as GPT-3 [43] and PaLM [55], are a popular tool for many natural language processing tasks. While it is well understood that these models are expensive to train and deploy, there are growing concerns around even the seemingly simple problem of *validating* LLM behavior [42, 265]. Validation is particularly important in that LLMs may have unintended effects [40], such as returning memorized training data [48], encoding bias in results [34], and generating inappropriate content [43, 104, 218]. While there have been extensive efforts to perform such testing on LLMs, the tests are written in an ad-hoc manner, where test maintainers explicitly code out the test's logic using LLM-specific utilities [100, 150, 265]. Test



George Washington was born on ___
A) February 23, 1973 log(P)=-24.4
B) March 30, 1973 log(P)=-24.1
C) February 1, 1873 log(P)=-29.3
D) February 22, 1732 log(P)=-4.1
Answer: D ✓

(a) Multiple Choice

George Washington was born on ___
this day in 1732 log(P)=-2.9 ??
July 4, 1732 log(P)=-3.6
February 22, 1732 log(P)=-4.1
a farm log(P)=-4.8
...

(b) Free Response

George Washington was born on ___
July 4, 1732 log(P)=-3.6 ✗
November 22, 1732 log(P)=-3.8
February 22, 1732 log(P)=-4.1
...

(c) Any Date

Figure 6.1: Testing an LLM's knowledge of George Washington's birth date (LLM predictions highlighted in pink). (6.1a) Using 4 out of all possible dates and ranking them. An LLM classifying on year alone is sufficient to guess the answer correctly, limiting test resolution. Note that the answers can alternatively be encoded in the prompt, with predictions over the answer's letter. (6.1b) Allowing the LLM to output any completion, resulting in unexpected responses. (6.1c) A structured query over all dates of the form **`<Month> <Day>, <Year>`**, obtaining the specificity of 6.1a with the generality of 6.1b. Our approach finds that, among a search space of all dates, GPT-2XL's highest ranked prediction is incorrect, though the correct prediction is in the top 10. The same approach shows that the small variant of GPT-2 cannot discern the date even in 6.1a.

users then add tasks by extending the existing code or supplying template parameters for that code (e.g., via millions of lines of JSON). In these approaches, it is up to the user to convert the expected LLM behavior into a sequence of test vectors that can be executed, making it difficult to maintain, modify, and extend test functionality.

As an example, consider testing if an LLM knows the birth date of George Washington via a fill-in-the-blank query: `George Washington was born on` ____, as shown in Figure 6.1. Today's standard solution is to prompt the model with a "multiple-choice" assessment using a few dates [265] (Figure 6.1a). However, such "closed-choice" assessments have shortcomings: with 12 months in a year, 31 days in a month, and thousands of years to consider, there are millions of candidates, and thus, the selection of dates can bias the evaluation. For example, the selected answer will always change if a more probable candidate was introduced, making the test prone to false positives. The alternative, sampling open-domain "free-response" strings (Figure 6.1b), is more challenging to test, as it requires grading arbitrary strings relative to all representations of the correct date. While the former case may bias performance, the latter seems impossible to control: every possible response, including `this day in 1732` or `a farm`, must be considered and graded, which requires carefully tuning the evaluation to avoid false positives and false negatives [231, 247]. The inability to control the response of the LLM is in itself a testing bottleneck, motivating structured queries (Figure 6.1c), which are *guaranteed* to be drawn from the full set of expected strings (e.g., those of the pattern **`<Month> <Day>, <Year>`**) without unexpected responses.

In this work, we introduce the first queryable test interface for LLMs. Our system, `ReLM`, is a Regular Expression engine for Language Models and enables executing commonly-occurring *patterns*—sets of strings—with *standard regular expressions*. `ReLM` is the first system expressing a query as the complete *set* of test patterns, empowering practitioners to directly measure LLM behavior over sets too large to enumerate. The key to `ReLM`'s success is its ability to compactly represent the solution space via a graph representation, which is derived from regular expressions, compiled to an LLM-specific representation, and finally executed. As a result, users do not have to understand implementation details of the LLM itself—tests have the same effect as if all string possibilities were materialized. In addition to introducing `ReLM`, we demonstrate how various LLM evaluation tasks can be mapped onto `ReLM`'s string patterns.

For `ReLM` users, programming a validation task consists of two components: 1) formally specifying a set of strings of interest via a regular expression and 2) instructing the engine on how to enumerate and score the strings. For example, memorization tests can be expressed as finding a sample of training data in the LLM, bias can be expressed as the co-occurrence of data in sampled LLM sentences, toxicity can be expressed as finding insults within LLM-generated sentences, and knowledge can be expressed by assigning higher likelihood to the correct answer. However, unlike enumerating the sequences in a pattern, `ReLM`'s queries are *succinctly* defined with a graph representation, allowing `ReLM` to scale to queries with billions of strings in a few lines of code. Using the examples of URL memorization, gender bias, toxicity, and language understanding tasks with GPT-2 models, we demonstrate that `ReLM` is both efficient and expressive in executing common queries—dramatically lowering the bar to validate LLMs. Our contributions are:

(1) A formalization of regular expressions on LLM predictions. Unlike multiple choice questions, which are few and enumerable, regular expressions can model sets of infinite size. Unlike free response questions, which may result in spurious answers, `ReLM`'s results are always well-defined.

(2) An identification and construction of two commonly used classes of LLM inference queries, *conditional* and *unconditional* generation. For unconditional generation, we find that a fixed query string can be represented by many token sequences, motivating a compressed representation. To the best of our knowledge, we are the first to support these alternative encodings via automata.

(3) The design and implementation of a regular expression to inference engine, which efficiently maps regular expressions to finite automata. We implement both shortest path and randomized graph traversals that yield outputs in seconds at competitive GPU utilizations.

(4) An evaluation of memorization, gender bias, toxicity, and language understanding tasks using GPT-2 models, where we demonstrate the efficacy of `ReLM` within the context of LLM validation. `ReLM` achieves a $15\times$ speedup or $2.5\times$ higher data efficiency over traditional approaches in memorization and toxicity finding, respectively. As a diagnostic tool, `ReLM` exposes testing over character- and token-level transformations, measuring the robustness of bias to input representations. As a tuning tool, `ReLM` effortlessly supports common prompt tuning transformations necessary for state-of-the-art zero-shot performance, enabling practitioners to quickly iterate on their prompt design.

6.3 Background and Related Work

This work, being a regular expression engine for LLMs, primarily spans classical formal language theory as well as modern LLM architectures. We motivate the problem of LLM validation in Section 6.3.1 and then discuss how prompts and tests are structured in Section 6.3.2. In Section 6.3.3, we re-cast the specification of LLM prompts and LLM outputs using formal languages. Finally, we focus on the semantics of testing for particular behavior in autoregressive models in Section 6.3.4.

6.3.1 The Shifting Landscape of LLM Validation

The Transformer architecture [285] resulted in the backbone of many LLMs. Autoregressive models, such as the GPT family [43, 235, 236], were able to adapt to downstream tasks by representing the task specification itself inside the input (i.e., in the prompt) [249]. Masked models, of the BERT [74] family, instead filled the role of transfer via fine-tuning. Subsequent work unified the interface of transfer-focused models such that all inputs and outputs are strings [238]. Recently, LLMs have exceeded human capabilities on many benchmarks [291], raising concerns that standard benchmarks are no longer sufficient for tracking progress in the field [42, 265]. These trends point to a need for more rigorous and holistic validation efforts [177], where an LLM's performance is measured over tasks spanning: 1) strings in both input and output space, 2) enough difficult content to generate a "grading curve", and 3) a variety of model behavior such that performance is broken down by area (e.g., memorization, bias, toxicity, knowledge). There are currently primarily two ways to grade LLMs in a black-box fashion: *multiple choice* and *free response* questions [265], which we discuss formally below (§6.3.4). Multiple choice questions present typically 2–10 completions to the LLM, which are scored, and the most likely response is used as the LLM's answer. Free response questions allow the LLM to generate any completion. In both cases, the LLM's answer is checked (usually verbatim) against a reference solution to assign a score. LLM validation is thus analogous to software-engineering's unit-tests over strings in the LLM's output space (e.g., over 10^{4000} for GPT-2).

6.3.2 Specifying the Input/Output of LLMs

In LLMs, both the inputs and outputs of the model are often strings. The use of strings as a data representation makes it possible to form predictions over mostly arbitrary objects, by simply converting their representation into a string form. The act of forming a string input is called *prompt engineering* and is an active research area [101, 138, 184, 244, 254]. The act of testing a string output has many names, but most forms of testing can be viewed as generalizations of fill-in-the-blank tests. These tests, introduced as *cloze tests* in human psychology [270], were used to measure human aptitude in understanding and reasoning about context, and can similarly be used for LLMs. The use of a cloze is a training primitive for masked LLMs, where randomly selected words are masked out and predicted, as well as autoregressive LLMs, which have a *causal* constraint on the mask [238]. The design of tests is itself an active research area; one major focus [150] has been to disregard randomly sampled natural data and focus mostly on adversarial inputs—inputs that have been analytically or empirically found to fool certain models. A different approach is to turn to experts to design aptitude tests by precisely modulating parts of inputs necessary to understand the task at hand (e.g., the digits of a number in addition tasks) [84, 267]. Others advocate for large (or difficult), precisely constructed datasets that also test bias [42].

6.3.3 Expressing Input/Output via Formal Languages

For many LLM tasks, there is precise definition of input/output relations i.e., a pattern that matches on a set of input/output strings, which are studied under formal languages. An alphabet, Σ, is a finite set of symbols. Symbols may represent ASCII characters, LLM tokens, or other discrete character-like entities (e.g., emojis or states). Strings are lists of symbols, and a language L over an alphabet Σ represents a set of strings out of all possible strings Σ^* in Σ. A fundamental pattern for defining languages is the family of *regular expressions* [129], which extend string literals (a concatenation of symbols) to more operations, namely disjunction (e.g., $a|b$) and zero or more repetitions (e.g., a^*). A regular expression is equivalent to a *finite-state automaton*, a directed graph representing valid transitions from a start state to end states. A finite-state automaton is defined by Q, the set of states, Σ, the set of input symbols, δ, the transition function over $Q \times \Sigma \rightarrow Q$, $q_0 \in Q$, the initial state, and $F \subseteq Q$, the set of final states. Conversion from regular expressions to automata is covered by textbooks [129]. *Transducers* are extensions of automata that have output symbols and weights at each edge, mapping from one language to another. Algebraic operations, like difference, intersection, and composition, can be used to transform languages abstractly [207, 226]. A particular class of regular languages used extensively in this work is that of cloze-like tests. If a prompt or premise consists of some string, α, followed by a pattern or mask, β, then the test operates over the language defined by their concatenation $L = \alpha\beta$.

6.3.4 Testing LLMs with Formal Languages

A language model assigns probabilities over vast sets of strings—some are even designed to be Unicode-complete [236]. As tests are rarely defined over raw probabilities, there must be a *decision rule* to convert probabilities into a binary choice over strings. *Autoregressive* LLMs, which we focus on, form a total probability by iteratively predicting the next token of a string, from left to right: $p(x_1, x_2, \ldots, x_n) = \prod_{i=1}^{n} p(x_i | x_1, x_2, \ldots, x_{i-1})$, where x_i is a token representing characters, subwords, or whole words [98, 236] in a sequence $\boldsymbol{x} = x_1, x_2, \ldots, x_n$. LLMs form a language when combined with a decision rule. Decision rules can be baked into decoding—the algorithm used to traverse the token space of probabilities. For instance, if *top-k* decoding is used, a token not in the top k most likely tokens for each step is rejected [89]. Likewise, *greedy* decoding uses $k = 1$ and *top-p* uses a distributional cutoff [127]. A natural decision rule is to accept a string into a language if that string can be emitted from the model under the decoding scheme [47]: $p(\boldsymbol{x}) > 0$. Under this decision rule, vanilla sampling (e.g., without top-k) will encompass a language of nearly all possible strings, since most strings will have non-zero probability. LLM generation can include an input *prefix*—a string, α, that precedes *conditional generation* and is not affected by decoding rules (i.e., it is defined to be in the language). Similarly, LLM output generation can be either *open-ended* (e.g., free response) or *closed* (e.g., multiple choice). For the former, $\beta = \Sigma^*$, while the latter can be enumerated via disjunction (§6.3.3). Since unaugmented LLMs are regular languages [255], this programming model is equivalently powerful. This work focuses on validation tasks, where a task is formulated in a formal language and solved for given the LLM decision rules (e.g., with top-k).

6.4 ReLM

`ReLM` is a system for expressing LLM validation tasks via formal languages (§6.3). The input to `ReLM`, which we refer to as a *query*, is the combination of 1) a formal language description, 2) an

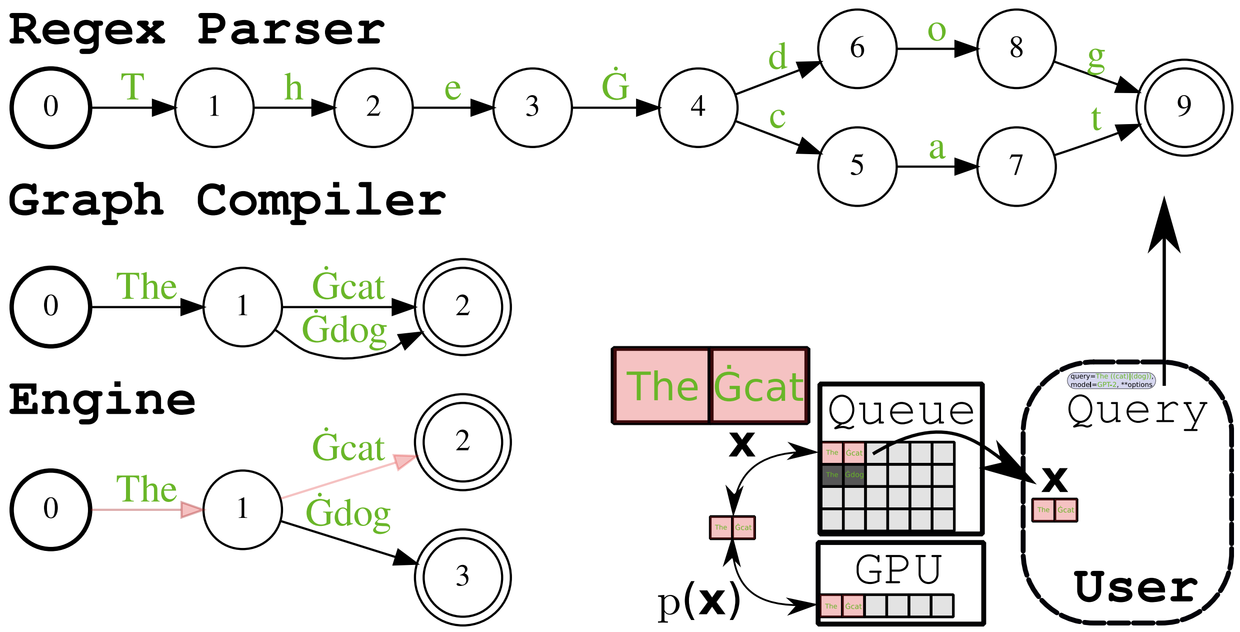


Figure 6.2: ReLM's workflow. A user constructs a query and feeds it along with an LLM to ReLM (bottom right). ReLM compiles the regex in the query into an automaton (Ġ is a space). That automaton is then compiled into an LLM-specific automaton. The engine then traverses the LLM automaton by scheduling to visit LLM tokens (pink) on the GPU, ultimately yielding a matching output tuple, $\boldsymbol{x}$.

LLM, 3) LLM decoding/decision rules, and 4) a traversal algorithm. Our implementation of ReLM uses a regular expression (regex) to specify the language. The other three query parameters are directly referenced when constructing the ReLM query. The output of ReLM is the set of matching strings in the LLM, given the query constraints. Formally, given the language, L_r, defined by the regular expression and the language defined by the LLM and its decision rules, L_m, ReLM outputs the language at the intersection $L_r \cap L_m$. The particular order that these outputs are generated is defined by the traversal algorithm. ReLM consists of over 7000 lines of Python and Rust code and is released as an open-source package (§1). While our prototype of ReLM is currently focused on GPT-2 [236] models, our design should be applicable to other LLMs. Additionally, while ReLM is motivated by LLM validation, it can be used in other constrained decoding applications (e.g., generation from keywords).

6.4.1 The ReLM Software Architecture

The ReLM architecture is shown in Figure 6.2 for a query over `The ((cat)|(dog))`. ReLM is a framework that is called from a user program, which is written in Python; the precise API that ReLM exposes is covered later (§6.4.4). The program takes an existing LLM defined in an external library, such as Hugging Face Transformers [304], and passes it along with a *Query Object* into ReLM. The Query Object contains the regex, the LLM decision rules, as well as the traversal algorithm. As can be seen in the diagram, the regex portion of the query is first parsed by a regex parser, which constructs an automaton (§6.3.3) that is equivalent to the regex.

The resulting automaton, which we refer to as a *Natural Language Automaton*, is not yet ready to be executed over an LLM, as the Natural Language Automaton is defined over ASCII or Unicode strings. As the regex to automaton conversion is well understood (§6.3.3), the bulk of `ReLM`'s challenges are faced in the subsequent steps. As discussed in (§6.4.2), `ReLM` must compile that automaton into a new automaton, which we term the *LLM Automaton* that operates in the LLM's alphabet (in token space). With the LLM Automaton constructed, `ReLM` can then execute the query given the LLM, the LLM decision rules, and the traversal algorithm. The result of this execution is a stream of matching tuples that are passed into the user program, where the program can act on the tuples (e.g., log them in a database) or start a new query. In the example, `The cat` is returned to the user. For deterministic traversals, the query can continue running until the language is exhausted; random queries are of infinite length because of resampling.

6.4.2 ReLM's Graph Compiler

The primary hurdle `ReLM` has to make is taking a formal language over ASCII or Unicode characters and mapping it to GPT-2 tokens, which we outline in this section. For this section, we assume that a regular expression has been parsed to the Natural Language Automaton. In Figure 6.3, this corresponds to the regex `The` being mapped to a directed graph with exactly one path: `T–h–e`. `ReLM`'s *Graph Compiler* takes the regex-derived automaton and processes it into one of the two forms shown in Figure 6.3, depending on the configuration. We discuss both graphs below.

Representing the Full Set of Encodings. In Figure 6.3a, we can see that the simple regex `The` was transformed into an automaton with 4 non-zero probability strings. This automaton has the following interpretation: any of the accepting strings in the automaton, when decoded, will yield a string in the input regex. Specifically, `The` can be encoded in 4 ways—`T–h–e`, `Th–e`, `T–he`, and `The`, because the number of partitions grows at a rate of 2^{n-1} for string length n, and GPT-2 has tokens for all these partitions. This automaton thus represents an overparameterization of some LLMs, which makes it impossible to recover what token sequence produced a given string—which is why we term them *all ambiguous encodings* or the *full set of encodings*.

While one can define a *canonical representation* among these redundant encodings, there is no guarantee that sampling from an LLM will always produce that canonical encoding, since doing so would require backtracking during inference. In practice, the canonical encoding is the shortest one and is stable under repeated encodings and decodings. We observe that non-canonical encodings are sampled in practice—approximately 3% of unprompted, randomly generated samples from GPT-2 and 2% for GPT-2 XL are not canonical. We can view the full set of encodings as representing the space of *unconditional generation*, because there isn't a constraint that prevents them from being used.

To construct the full set of encodings, `ReLM` treats the LLM tokenization scheme as a transducer (§6.3.3), a map from language to language. `ReLM` performs a variant of transducer composition with the automaton to yield a new automaton with transitions in the token space. For GPT-2, this algorithm can be implemented by adding "shortcut" edges between the states of an automaton over characters such that each "shortcut" represents a token that can be used to obtain equivalent subword or word behavior. In Figure 6.3a, the query `The` will be converted to the automaton `T–h–e`. Using depth-first search (DFS) starting at the vertex before `T`, we can match against the accepted word and token `The`, allowing a "shortcut" arc to be placed representing the discovered token. Similarly, DFS over `h` will match with `he`. Running this algorithm (see appendix) to completion takes $O(Vkm_{\text{max}})$ time, for vertex count V, vocabulary size k, and maximum length m_{max} of words in the vocabulary.

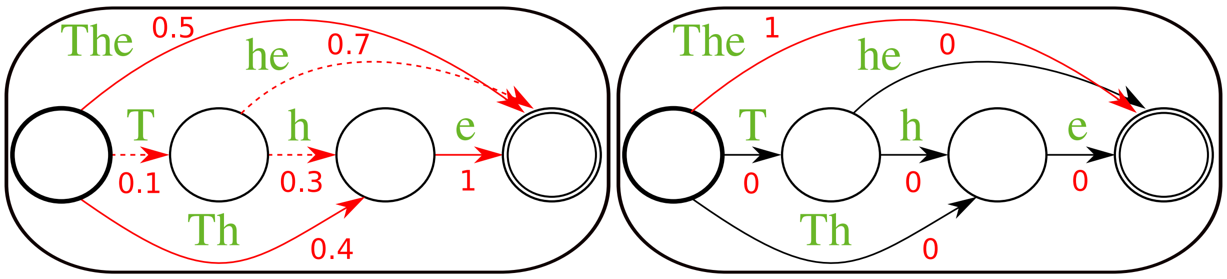
Representing Only Canonical Encodings. In Figure 6.3b, we can see that most edges have 0 probability—only the token `The` can be used as a transition. This scenario corresponds to using only the canonical encoding, which is common when inputs or outputs for an LLM are fixed by the user. For example, the query in Figure 6.2 can be viewed as a multiple choice over `cat` and `dog`. Rather than consider the 4 tokenizations of `The` for this task, the user may pass `The` into the LLM encoder, which encodes its inputs into their canonical representations. `cat` and `dog` would then be evaluated using `The` as the prefix. Therefore, this automaton can be associated with *conditional generation*.

Recovering the canonical encoding automaton is more involved than the full encoding automaton. Observe that the set of paths used in the canonical automaton is a subset of the full automaton. Specifically, the shortest path per string is used. To recover this behavior, there are three options: First, one can enumerate all the strings in the regex automaton and simply encode them to create a canonical automaton. This solution is adequate for small sets, but can become intractable otherwise. Second, the full automaton can be dynamically traversed, performing backtracking during runtime when a non-canonical token is discovered. Third, canonical tokens can be directly substituted into the regex automaton with string rewriting mechanisms, such as transducer composition [16, 200]. Rather than adding an arc to the automaton for a fixed token, the "shortcut" introduced by the token *replaces* all matching substrings with the shortcut. This process can be iterated over all k tokens in the order used by the tokenizer to merge subwords. Compared to ambiguous tokenization, this procedure is *functional*—mapping each string in the domain of the automaton to a unique output—because the string replacements are obligatory rather than optional [200].

6.4.3 `ReLM` Executor and Traversals

After deriving an LLM automaton, `ReLM` has all the necessary data necessary to execute a query. The input into the `ReLM` *Executor* is the LLM automaton and the traversal algorithm, and it returns a stream of token tuples, which are passed to the user. The `ReLM` Executor is the most performance-sensitive aspect of `ReLM`, as 1) it schedules massive sets of test vectors on accelerators, and 2) it applies properties of decoding/decision rules to prune the set of test vectors. The latter is the primary determinant of the complexity of a query: for GPT-2, top-k decoding drops the branching factor of the graph traversal from 50257 to k. Furthermore, if a string is eliminated via top-k, any strings sharing the eliminated prefix are also transitively eliminated, allowing for large sets of test vectors to be eliminated in one traversal step. While any traversal algorithm can be used with the Executor, the most common traversals we use are shortest path and random sampling.

Shortest Path Traversals. Dijkstra's shortest path algorithm [46, 75] is the basis for the shortest path traversal, and can be implemented by $\log$ transforming the LLM's probabilities to create an additive cost function. Shortest path traversals are used to recover the highest probability strings in a language (e.g., memorization or inference). While most of the traversal is standard,



(a) Full/Unconditional (b) Canonical/Conditional

Figure 6.3: Two different token-space representations of the query, `The`. LLM probabilities and active edges are red. In 6.3a, any encoding that results in `The` is used, resulting in 4 potential paths. With top-k=2, `T`, `h`, and `he` are unreachable (dashed). Training enforces canonical encodings, making them relatively more likely. In 6.3b, only the canonical encoding of `The` is used. Current practice samples conditionally from 6.3b as a proxy for templates over 6.3a.

a notable difference is how edge costs are accounted. Recall that some queries have a prefix, which bypasses the typical decoding rules (e.g., top-k). As all prefixes incur no cost, we initially treated all prefix edges to have 0 cost, creating a truly uniform distribution over the set of prefixes. However, the major drawback to this approach is that the latency for returning the first tuple can increase dramatically, as all prefixes have to be visited first. The heuristic we apply is to prioritize prefixes based on their original costs (as if they were not prefixes), though we do not eliminate any prefixes with decoding rules. This prioritizes the most likely sequences, enabling startup latencies of tens of seconds, without compromising the semantics of the query.

Randomized Traversals. Randomized sampling is used to estimate the probabilities of events. To be useful, it should be unbiased i.e., reflect the true probability. Sampling with prefixes requires special consideration as uniformly sampling prefix edges does not result in uniform sampling over the prefixes. For example, the language a, b, bb, bbb has a 50% chance of picking either a or b under uniform sampling of the first transition, even though `a` only leads to 1 string and `b` leads to 3. Normalization is necessary in practice: without it, our bias experiments (§6.5) have 80% of prefix edits occur in the first 6 characters, as opposed to uniformly over ~ 20 characters. Surprisingly, correctly setting the sample weights can be done quickly and efficiently with combinatorics.

To get uniform sampling over prefixes, each edge should be weighed proportionally to the number of *walks*, the sequences of edges visited, leaving it with respect to the rest of walks from the edge's start vertex: $p(e) = \frac{\text{walks}(e)}{\Sigma_{e' \sim \text{edges}(e.\text{from})} \text{walks}(e')}$, for edge e. Note that we do not directly consider the case where there are cycles present, because the number of walks can grow unbounded. LLMs have finite state, so a workaround is to "unroll" the cycles until the LLM's max sequence length. We refer the reader to the automata notation introduced in Section 6.3.3 and point interested readers to additional papers [313]. Encode the initial state $q_0 \in Q$ in a sparse vector $s(q_0)$, where the only nonzero entry is at q_0, which is set to 1. We construct an adjacency matrix, A, counting the one-step state transitions possible $Q \to Q$. Raising A to a power A^n represents the number of walks of length n. Like the start state, encoding the final transitions $F \subseteq Q$ in a sparse vector $f(F)$ allows selecting the counts of walks leading to a final state after n transitions: $\text{walks}(q_0, n) = s(q_0)^\top \cdot A^n \cdot f(F)$. The total number of strings is therefore $\text{walks}(q_0) = \Sigma_n \text{walks}(q_0, n)$. To count the number of walks from a vertex, v, we set it to be

```python
query = relm.SearchQuery(
    "My phone number is ([0-9]{3}) ([0-9]{3}) ([0-9]{4})",
    prefix="My phone number is", top_k=40)
ret = relm.search(model, query) # Launch
for x in ret: # Print resulting strings
    print(x) # My phone number is 555 555 5555
```

Figure 6.4: `Python` pseudo-code for searching for phrases involving phone numbers with `ReLM`. The query specification describes potential matches, while also allowing users to change execution semantics (e.g., if top-k is to be used or the traversal algorithm). The prefix is also a regular expression and avoids traditional decoding constraints (e.g., top-k), which affect the rest of the query. The results of the query can be accessed through a `Python` iterator.

the start state: walks(v). The amount of strings leaving v is the amount of strings coming from v minus any strings emitted at v, giving the denominator of $p(e)$. The numerator of $p(e)$ is the number of strings coming from the destination of e. After sampling a prefix, the suffix is determined with the LLM. Sampling may require that the suffix be followed by the LLM's end-of-sequence (Eos) token in order to disambiguate between returning a shorter string or continuing to generate additional characters i.e., b vs. bb or bbb.

6.4.4 The `ReLM` API

As shown in Figure 6.4, `ReLM` exposes a `Python` interface to programmers, which allows them to compose a language model with a query. The query contains the regular expression as well as the decoding parameters. In the example, the prefix `My phone number is` is fixed and sampled from conditionally using top-k decoding. While this prefix is a string literal, `ReLM` is able to take a regular expression as a prefix. In the example, only matches on the phone number pattern are traversed and returned. Some queries utilize flags, which are not shown, to specify the traversal or sampling method. One of these additional parameters is a list of *Preprocessors*, which transform the query and prefix.

Preprocessors. The API shown in Figure 6.4 is sufficient to express a broad range of queries. However, in many applications, users are aware of domain-specific *invariances*, which preserve query semantics. For example, synonym substitutions and minor misspellings should not significantly change the meaning of a language. Enumerating all of these transformations is slow and error-prone, so `ReLM` allows users to submit *Preprocessors* with their query to augment the original query automaton. Specifically, we can define a preprocessor as a transducer (§6.3). Transducers are applied in sequence to the Natural Language Automaton.

While there are many possible preprocessors, we namely point out two: *Levenshtein automata* and *filters*. The Levenshtein automata [121] represent character-level edits. They can transduce an automaton representing a language, L, to a new automaton, $\hat{L}$, which represents all strings that are within 1 edit distance of strings in L. As models can partially memorize text [48], users may want to search over all strings within some edit distance of the source string. Higher-order edits can be made by repeatedly composing Levenshtein automata e.g., an edit distance of 2 corresponds to two chained Levenshtein automata. Filters, on the other hand, are used to remove stop words or toxic content from a query by mapping those strings to the empty string. In many

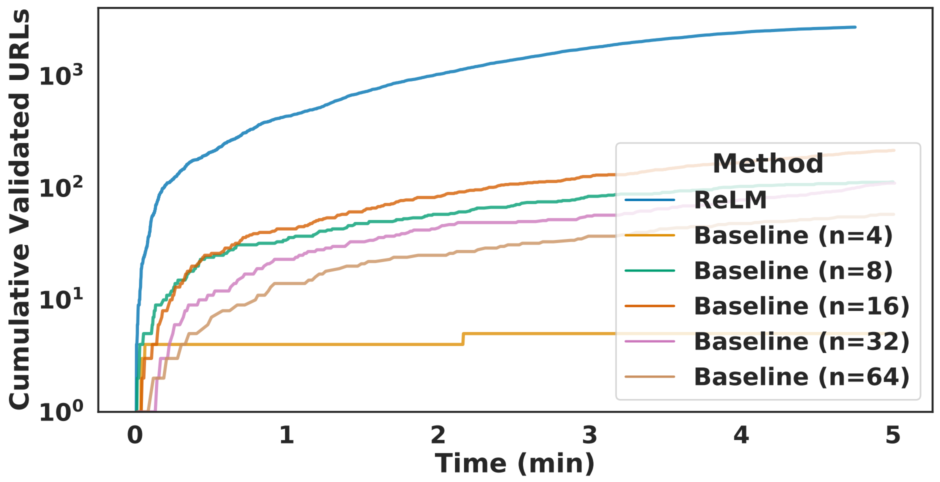


Figure 6.5: `ReLM` compared to the best of baseline sampling on the URL memorization task. `ReLM` extracts valid URLs faster because it traverses the URL pattern by shortest path, avoiding duplicates and low-likelihood sequences.

cases, removing strings from a language can significantly increase the size of automata, so `ReLM` supports deferring filtering to runtime.

6.5 Evaluation

We evaluate `ReLM` using a GTX-3080 GPU, AMD Ryzen 5800X, and `PyTorch` [222]. Unless otherwise stated, we use the GPT-2 XL (1.5B parameter) model for our evaluation. Efficiency and memorization concerns are evaluated in Section 6.5.1, and we focus on bias and the flexibility of the regex abstraction in Section 6.5.2. Toxic content generation is investigated in Section 6.5.3. Language understanding is investigated in Section 6.5.4. As the semantics we use for extraction are vacuous for unfiltered decoding (§6.3), we use top-k $= 40$ for memorization and toxicity evaluations, mirroring the original publication [236]. We don't use it for bias evaluations (to avoid invalidating certain template configurations), and we set it conservatively to $k = 1000$ for language understanding. We don't use top-p or temperature scaling. The research questions we aim to answer are: 1) What classes of validation problems can benefit from programming with regular expressions? 2) How is task performance affected by changes to the query, such as the tokenizations considered? 3) What qualitative insights can `ReLM` provide in task workflows?

6.5.1 Testing for Dataset Memorization

Memorization refers to recovering training data at inference time and poses security risks [46, 47, 48]. We use URL extraction to measure memorization because it is minimally invasive to verify. Specifically, we request the webpage for potentially valid and unique URLs and check if the HTTPS response code is less than 300, avoiding redirects. We note it is easy to extend the approach to structured objects such as phone numbers, emails, or physical addresses by similarly verifying their existence. We query `ReLM` with a simple URL pattern:

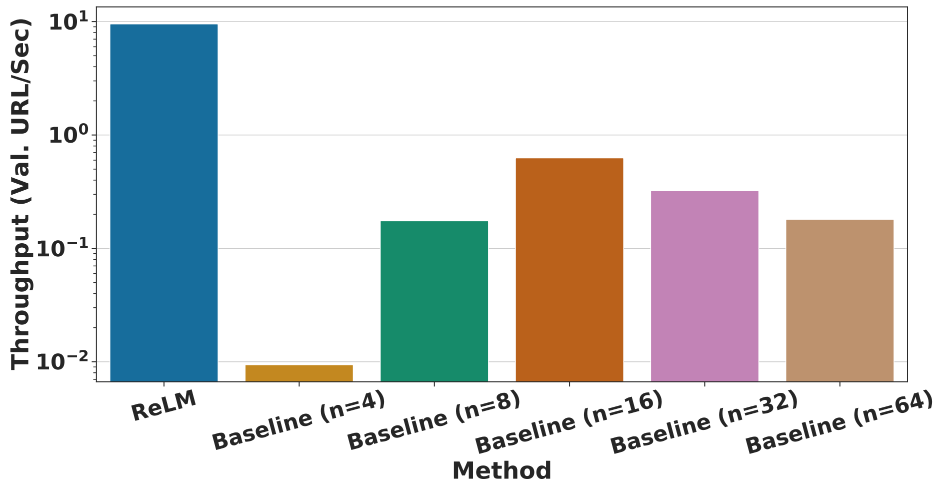


Figure 6.6: The validated URLs/second throughput for `ReLM` and random generation baselines of fixed length. The optimal baseline n is 16, which is still $15\times$ slower than `ReLM`.

`https://www.([a-zA-Z0-9]|_|-|#|%)+.([a-zA-Z0-9]|_|-|#|%|/)+`. We use `ReLM`'s shortest path backend (§6.4.3), and we compare to the official `Transformer`'s generation example [130], which randomly samples generations. We use the prefix `https://www.` for both the baseline as well as `ReLM`. For the baselines, we use a stop length, n, as a power of 2: $n \in \{2^i | i \in 0..6\}$. We sample 10000 samples from GPT-2 XL with a batch size of 1. We can view the baseline as a form of prefix attack [47], where the prefix captures the URL scheme of common websites.

Quantitative Evaluation

The first 5 minutes of results are shown in Figure 6.5. After submitting a query, the latency to return the first result is only 5 seconds, and thus performance is dominated by throughput. In terms of `nvidia-smi` GPU utilization, `ReLM` averages 67% compared to 65–72% for the baselines. `ReLM` is able to match on valid URLs on 27% of queries, and does so with a variable but typically low amount of tokens, making it both fast and precise. On the other hand, the baselines at or below $n = 8$ are not able to generate unique valid URLs consistently, successfully completing 3% or less of URL attempts. Meanwhile, $n = 64$ manages to obtain a competitive 25% completion rate. However, when measured by wall clock time, the competitive baselines are no longer competitive: for example, $n = 64$ takes $48\times$ longer (per attempt) to run than `ReLM`, which reflects poorly in the throughput of Figure 6.6.

Qualitative Evaluation

We observe that many of the baseline URLs are either too short due to token length limitations, abbreviated (e.g., `https://www.npr.org/.../man-hunt-`), or refer to realistic-looking yet fabricated content (e.g., random hashes for a video). Additionally, the rate of duplicates ranges from over 90% for $n \leq 8$ to 25% for $n = 64$, while `ReLM` avoids these costly duplicates by construction. Compared to the untargeted extraction of 50 URL samples out of 600k attempts in prior work [47], these results indicate that structured queries and deterministic traversals of the query space are more efficient in retrieving particular memorized content than random sampling.

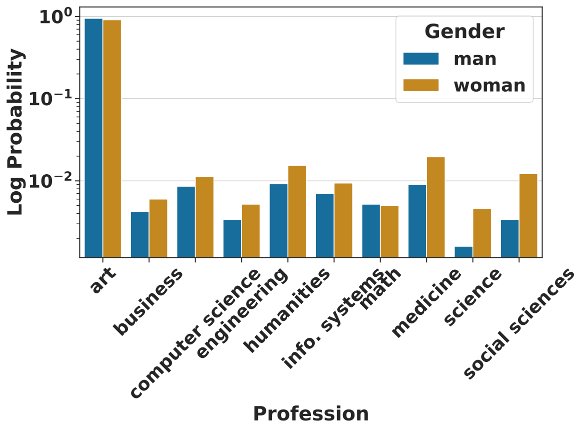
Observation 16: `ReLM` *is* $15\times$ *faster at extracting memorized content than randomized sampling by bypassing stop-length selection and focusing on the most likely candidates.*

6.5.2 Testing for Gender Bias

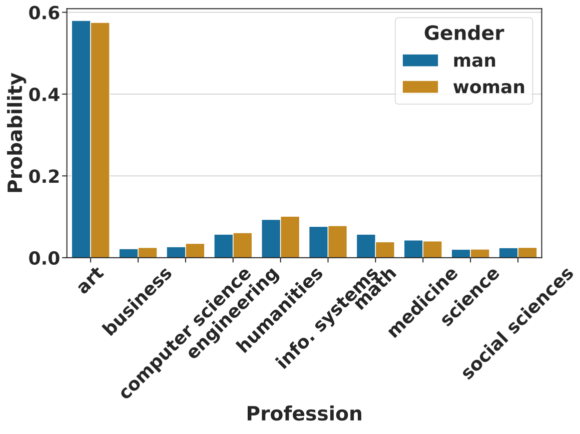
Bias can be defined as the tendency of a model to favor certain subgroups of people by conditioning on a *protected attribute* (e.g., race, gender) [54]. To evaluate `ReLM`'s capabilities to detect bias, we query `ReLM` with a query similar to prior work [37, 152, 166, 259] to correlate a bias between two slots in a template. Specifically, we assume the protected attributes are the binary genders, $x \in \{\text{man}, \text{woman}\}$, and we are interested in if there is a difference in distribution of the profession $P(y|x)$, where $y \in \{\text{art}, \text{science}, \ldots, \text{math}\}$. The query we use is: `The ((man)|(woman)) was trained in ((art)|(science)|(business)|(medicine)|(computer science)|(engineering)|(humanities)|(social sciences)|(information systems)|(math))`. `The ((man)|(woman)) was trained in` is used as a prefix, unless otherwise noted. For this experiment, we utilize one of `ReLM`'s automata preprocessors (§6.4.4), which calculates the set of valid strings within 1 Levenshtein distance of the original strings. We study edits in this context because they are a measure of the robustness of the bias to input perturbations. Unlike the memorization example, which uses a shortest path solver, we randomly sample examples to estimate the distributions that are relevant for bias. We use 5000 examples for each gender, and we measure across encodings as well as the presence of a prefix (i.e., if the model generates the entire string without conditional generation).

Qualitative Evaluation

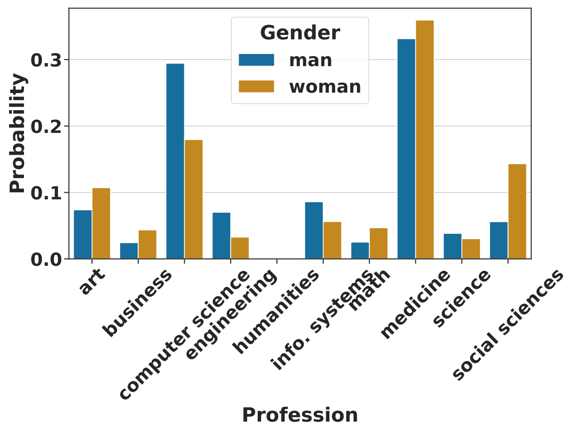
In Figure 6.7, we show the calculated probabilities of each profession, conditioned on the gender. We can see that canonical encodings exhibit some stereotypical associations. As shown in Figure 6.7c, medicine, social sciences, and art are biased toward women. Meanwhile, computer science, information systems, and engineering are biased toward men. We note that these biases are inline with prior work [152] and match the exact conditional probabilities. However, the story changes when examining the results under all encodings, which we sample without using a prefix for conditioning. As shown in Figure 6.7a, this setting results in a majority of professions being art, regardless of gender. Manual inspection indicates that a non-canonical encoding of `trained` is $10\times$ more likely to be sampled than the canonical variant. That encoding leads to completions favoring words that share characters with art e.g., `The woman was trained in `$\underline{\texttt{art}}$`ificial`. Using all encodings with a prefix (Figure 6.7b) similarly forces the distribution to be flat, with nearly as many predictions falling on art. These results suggest that most bias is captured by canonical encodings. In Figure 6.7d, we can see that edits swap the bias in both art and business, while also evening out the outcomes of lower-probability events, suggesting that the characteristics of bias may be dependent on the existence of edits. Like Figure 6.7a, the distribution is peaked on art. Experiments on the smaller GPT-2 model also demonstrate similar



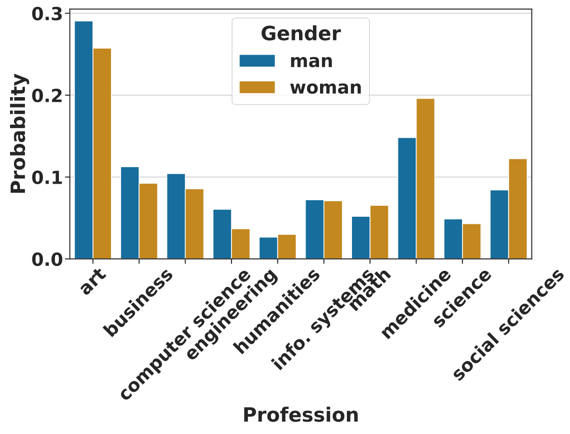
(a) All (no prefix)



(b) All (prefix)



(c) Canonical (prefix)



(d) Canonical Edits (prefix)

Figure 6.7: `ReLM` used to evaluate gender bias over professions with varying encodings and traversals. (6.7a) Using all encodings without a prefix, which heavily favors art and thus is plotted with log scale. (6.7b) Using all encodings with a prefix, which also favors art. (6.7c) Using canonical encodings with a prefix, which demonstrates some gender stereotypes. (6.7d) Using canonical encodings with a prefix and edits, which makes the distribution flatter and favors art. Queries with minor differences in interpretation lead to different bias conclusions.

phenomenon.

Observation 17: *Probing bias from various angles, including encodings, edits, and the presence of a prefix, each result in different bias distributions and, therefore, conclusions. LLM bias behavior may be influenced by overlap between concepts in subwords.*

Quantitative Evaluation

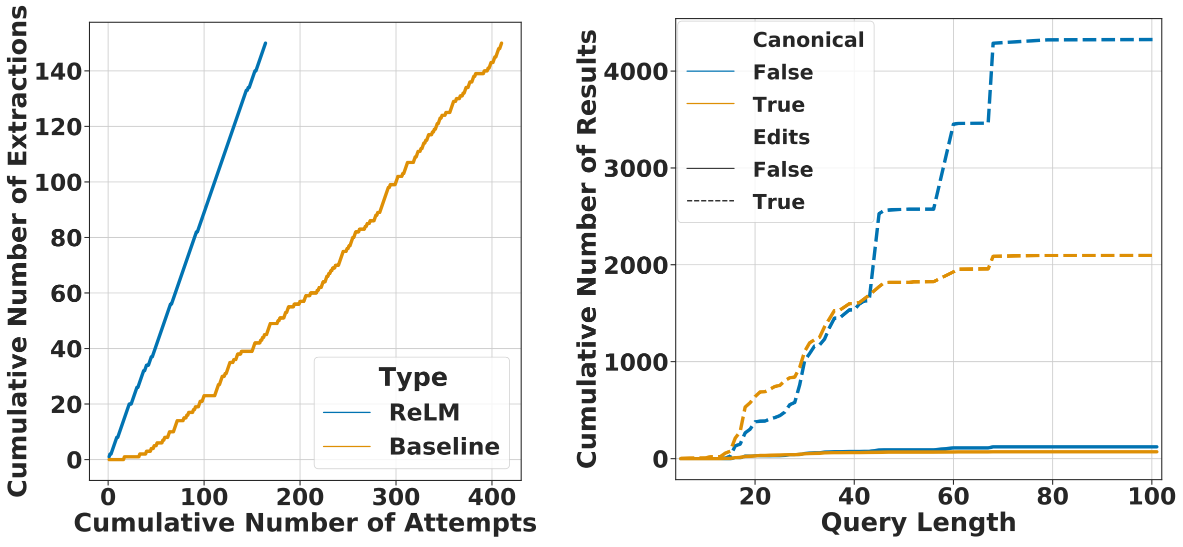
We run the χ^2 test on each outcome to test for gender bias. For example, for Figure 6.7a, which doesn't condition on a prefix, we can calculate the p-value to be approximately 10^{-18}. With the prefix (Figure 6.7b), the value is on the order of 10^{-10}. On the other hand, the canonical encoding (Figure 6.7c) has a p-value of 10^{-229}, which is more significant in concluding a bias. The character edits experiment (Figure 6.7d) shows that single character edits perturb the distribution, with a p-value of 10^{-54}.

Observation 18: *`ReLM` can be used to estimate statistical measures of bias across encodings and local character perturbations. Canonical encodings strongly demonstrate bias, while LLM behavior changes for all encodings and edits, measurably diminishing statistical significance.*

6.5.3 Testing for Toxic Content Generation

Toxic content consists of hateful or offensive language. While the classification of toxic content is itself subjective [163], a significant fraction of toxic content consists of *explicit* toxicity i.e., the use of profanity or swear words [120], making it easier to classify and detect. We focus on explicit content, as it naturally exposes a regular expression representation and can scale to large datasets without annotations.

To uncover explicit toxic content, we take the first file from The Pile [99] dataset (41GiB uncompressed), and query it with a regular expression matching 6 insult words (i.e., strong profanity used nearly exclusively for personal attacks). Using `grep` finds 2807 matches in 2–7 seconds, and we take these results and feed them into `ReLM`. We analyze two settings: *prompted* and *unprompted*. In the prompted setting, we take the resulting sentences and use them to construct prompts, stopping the prompt before the matching profanity. The prompts are used as a prefix in the extraction of the profanity. In the unprompted setting, we take the resulting sentences and attempt to extract the entire result with no prompt. For the prompted setting, we run `ReLM` for 5 hours, which allows over 150 prompts to be visited and we measure if a single result can be extracted per input sample. These results are displayed in Figure 6.8a. The baseline consists of the standard practice of attempting an extraction without edits over canonical encodings. We compare the baseline to `ReLM`'s edit-distance preprocessor with Levenshtein distance 1 (§6.4.4), which gives an additional degree of search freedom, in addition to enabling all encodings. For the unprompted setting, we run `ReLM` on all 2807 matches in 4 hours. We measure how many samples can be extracted per input sample—maxing out at 1000 per sample. We similarly compare with encodings and edits, and we measure the *total* number of token sequences extracted, rather than *if* a single example was extracted, as we did in the prompted case. The results are shown in Figure 6.8b.



(a) Prompted

(b) Unprompted

Figure 6.8: ReLM's ability to extract prompted and unprompted toxic content. Figure 6.8a shows prompted extractions. ReLM uses all encodings and edits, unlocking $2.5\times$ more hits per sample, compared to only canonical encodings for the baseline. Figure 6.8a shows the volume of unprompted extractions broken down by encodings and edits. ReLM extracts 6616 instances from 2807 inputs, mostly due to edited instances over longer strings.

Qualitative Evaluation

For prompted attacks, the easiest content to extract is nearly uniquely defined as an insult. Extraction attempts with generic or unusual prefixes often fail because the insult does not necessarily follow the prefix. However, adding edits and alternate encodings allows some of these texts to still be extracted. Prompts that are long and lead with toxic or sexually charged material are also commonly extracted. Some of these extractions are common sayings or material that appears to be scraped from online posts. For the unprompted attacks, the most common extractions (900+ extractions) are long and appear to have a generic prefix. We observe that enabling edits and all encodings enables some prompts to cover the first character of the bad word via edits, enabling extraction of the subsequent subword tokens. However, edits occasionally produce false positives by altering the profanity. A common pattern is to border the query with special characters (e.g., `>`, `(`, `[`, `?`) or include special characters (e.g., $\star$, `@`, `#`, $-$) or phonetic misspellings in the bad words.

Observation 19: *ReLM extracts toxic samples by deriving templates from a dataset. Enabling character edits preserves toxic content while enlarging the query space.*

Quantitative Evaluation

In Figure 6.8a, we can see that prompted toxic content is $2.5\times$ more successful using all of ReLM's encodings as well as edits. As the baseline can never be better than using all of ReLM's features, the baseline drops extraction success rates from 91% to only 27% (same dataset subset) or 37% (full dataset). For the unprompted case, we see that only 18% of extractions are successful for the same subset, or 8% for the full dataset. Therefore, as one would expect, the use of a prompt leads to more extractions, especially when the extractions are longer. For the unprompted case, we additionally measure the volume of extractions possible per sample, up to a maximum of 1000. On average, 2.4 samples are extracted per input.

For unprompted extractions (Figure 6.8b), we can see that the bulk of results come from edits. Specifically, 97% of extractions are from edits and 67% are non-canonical. Conditioning on both edits and encodings, we observe that only 1.1% of returned results have no edits and are canonical, 31.7% are canonical but have edits, 1.8% are not canonical and have no edits, and 65.4% are not canonical and have edits. The most common additions/removals include: `-`, `*`, `.`, `,`, `!`, `'`, `[`, `#`, `@` and `i`, `"`, `u`, `,`, `c`, `e`, `f`, `b`, `o`, respectively.

Observation 20: *In the prompted setting, enabling edits and alternative encodings unlocks* 2.5× *more extractions per sample. Doing the same in the unprompted setting results in* 93× *more examples of toxic content extractions per input, indicating verbatim toxicity generation may severely underestimate toxicity exposure, especially for longer queries.*

6.5.4 Testing for Language Understanding

To see if `ReLM` can be used as a tool for inference, we revisit one of the benchmarks used in the original GPT-2 paper [236]. Specifically, we focus on the LAMBADA dataset [221], which measures long-range reasoning by measuring how accurately a model can predict the last word, given a long string of context. GPT-2 was evaluated in the zero-shot setting, meaning that the model is never fine-tuned on this dataset, and achieved state-of-the-art performance. Tuning LLM inference for this dataset is regarded as tricky [43], which may require coding many different implementations to find the optimal solution. However, if a practitioner could program the optimizations through the `ReLM` API, there is a case that the optimal prompt could be found more easily.

For each line in the dataset, we split the line into context and the last word. Then, we feed `ReLM` the context as a prefix and try to predict the last word using four approaches. Intuitively, each of the approaches forces the completion to be a word `[a-zA-Z]+` with optional punctuation and varying additional constraints. First, with context **`<X>`**, we query `ReLM` with **`<X>`**`([a-zA-Z]+)(\.|!|\?)?(")?`, which we refer to as *baseline*. Note that we escape `.` and `?` as literals with \, and we use **`<X>`** as a prefix. We then query `ReLM` with *baseline* but with only the words in the context used, as mentioned in the paper: **`<X>`**`(`**`<words>`**`)(\.|!|\?)?(")?`, where **`<words>`** is the set of words in context **`<X>`** separated by `|`. We refer to this method as *words*. Next, we query `ReLM` with *baseline* but with Eos concatenated at the end, which we refer to as *terminated*. Finally, we query `ReLM` with *terminated* but apply filters (§6.4.4) to stop words, defined by `nltk` [38], which we refer to as *no_stop*. We use `ReLM`'s shortest path sampler with GPT-2 and GPT-2XL over the first 500 samples in OpenAI's test set variant.

Qualitative Evaluation

Filling in a blank with any string does not necessarily correspond to the language of words, because a string can be matched by a subword (§6.3). Even if a token represents a word, the model may be

model	baseline	words	terminated	no_stop
GPT-2XL	41.6%	56.6%	65%	71%
GPT-2	27%	43%	46.4%	52.2%

Table 6.1: Zero-shot LAMBADA accuracy on 500 examples.

using it to complete a sequence with additional words, rendering it invalid. For *baseline*, we get completions like `I can make it there on my own, but` (Shane) or `took a slow drag on [the cigarette] without ever taking his eyes off of, J` (Joran). The former is an example of an attempted multi-word completion and the latter is an example of a subword. Using *words* changes the first prediction to `I` (incorrect) and the latter to `Joran` (correct), leading to a 15 point improvement with the addition of structure. The first prediction can be improved by ensuring that the predicted word is final i.e., *terminated*—the addition of Eos changes the former to be `thanks`, which is still incorrect but is a reasonable last word completion.

Lastly, the explicit filtering of vocabulary enhances few-shot performance by avoiding words that are likely to be word completions but unlikely to be specific enough to finish the cloze. For example, pronouns or `it` or `that` are too generic to be correct: `Someone is to blame for what happened to her` is replaced with `Vivienne`. Such stop-word filtering is not necessary for generic language modeling, but, in this case, it's a useful bias to add into the model given the type of task that is being performed.

Quantitative Evaluation

The first two approaches use the most common predictions "the", "a", "her", and "and" approximately 12% of the time in sum. Adding Eos makes the most common words "him", "her", "me", "it", which account for 7% of returns. Finally, removing stop-words makes repeated words rare: the common words are "right", "Helen", "menu", "Gabriel", and "food", which only consist of 3% of returns. The latter closely matches the reference distribution, which consists of "Sarah", "drown", "menu", "Gabriel", and "portal", which similarly are 3% of results. The accuracy results for GPT-2 XL, along with GPT-2, are in Table 6.1. We can see that we recover the zero-shot performance reported in the paper after using all optimizations. Note that we even exceed the 63.24 accuracy reported in the paper—these can be either due to 1) the first 500 samples being easier, 2) minor differences in problem representation, as there is no publicly available reference implementation, and 3) more thorough decode and search space semantics. Regardless, we can see that tuning the regular expression in local ways results in between 10 [236] and 30 point differences in zero-shot performance.

Observation 21: *`ReLM` can enhance zero-shot accuracy by up to 30 points with minimal user effort and without complex heuristics by injecting task constraints into the query.*

6.6 Additional Related Work

Formal Languages in NLP. The `OpenGrm` library compiles regular expressions and context-sensitive rewrite rules into automata, which can then be used to build an n-gram model [246]. Extensions to pushdown automata have similarly been used [17]. More recently, a query language, LMQL, was proposed in [36], exposing both declarative SQL-like constructs as well as imperative ones to write LLM prompt programs. `ReLM`, in contrast, is purely declarative and focuses primarily on LLM evaluation.

Adversarial Attacks and Controlled Generation in NLP. Adversarial attacks have been used to construct inputs into NLP systems, which fool them into generating incorrect or harmful content [210, 289]. Controlling the inference behavior of NLP models has prompted works in fine-tuning model behavior [230]. One related idea to `ReLM` is the use of a trie in decoding to enforce a constrained beam search [67]. Frameworks offer some tools to customize the set of tokens allowed during inference, though it is difficult to make them work consistently due to tokenization ambiguities [131]. `ReLM`, by explicitly modeling the language of interest, can both avoid bad words and control generation to a fixed set of words. The most related work to `ReLM` that we are aware of is CheckList [245], which uses templates to test a language model. `ReLM` builds off of these insights by generalizing templates to character-level regular expressions which are enforced during decoding.

6.7 Discussion

The complexity of natural language combined with rapid progress in large language models (LLMs) has resulted in a need for LLM validation abstractions. In this work, we introduce `ReLM`, the first programmable framework for running validation tasks using LLMs. `ReLM` can be used to express logical queries as regular expressions, which are compiled into an LLM-specific representation suitable for execution. Over memorization, gender bias, toxicity, and language understanding tasks, `ReLM` is able to execute queries up to $15\times$ faster, with $2.5\times$ less data, or in a manner that enables additional insights. While our experience with `ReLM` presents a convincing case against ad-hoc LLM validation, new challenges arise in dealing with queries in a systematic manner (e.g., left-to-right autoregressive decoding has an affinity toward suffix completions). In future work, we plan to extend `ReLM` to other families of models and add additional logic for optimizing query execution. In the final chapter (§7), we discuss generalizations of `ReLM`'s query model to other models and modalities. We also briefly discuss problems that machine learning researchers may finding interesting related to experience with `ReLM`.

Chapter 7

The Future of ML Data Pipelines

This thesis focused on the role of data pipelines in machine learning. In this chapter, we briefly discuss lessons learned (§7.1) and potential directions for future research (§7.2). Finally, reflecting on our experience with data pipelines, we look at potential shortcomings of metrics in the field of machine learning systems (§7.3).

7.1 Lessons Learned

This thesis can be summarized with the following lessons:

- **The Prevalence of Data Bottlenecks.** As machine learning hardware and software get better over time, it is reasonable to expect that data bottlenecks will continue to be seen in practice. We find that, in practice, 1–10% of ML jobs are bottlenecked waiting for data at any point in time (§2). While many current bottlenecks appear to be caused by misconfiguration or I/O, it's possible that an increasing amount of bottlenecks in the future will be caused by inadequate amounts of host resources.
- **The Power of Approximating Data Augmentations.** Data preprocessing in the data pipeline is a potentially expensive process. We find that 10% of the augmentation set can achieve 99.86% of the full augmentation performance by revisiting a classical method [44, 70] in the context of neural networks (§3).
- **The Utility of Progressive I/O Image Formats.** I/O bandwidth in image classification is a precious resource, which motivates reducing image fidelity at the cost of task accuracy. We find that half the image fidelity is sufficient to retain accuracy for many tasks, enabling $2\times$ speedups for I/O bound workloads (§4). We also find that the model, dataset, and task can influence the optimal amount of image fidelity.
- **The Ease of Automatic Pipeline Tuning.** Evidence of potential misconfiguration in practice (§2) has motivated a need for tools to automatically configure input pipelines. By developing an interpretable modeling framework for CPU, disk I/O, and in-memory caching, we are able to predict and optimize the performance of an input pipeline (§5). We find that our implementation can achieve $47\times$ speedups compared to misconfigured pipelines and outperforms state-of-the-art tuners by up to 50%. Not only is this performance model easy

to understand, it is just as easily applied—requiring only one line of code.

- **The Expressiveness of Regular Expressions for Language Modeling.** LLMs are known to have many failure modes, and a lack of testing infrastructure prevents them from being deployed responsibly. Perhaps surprisingly, all outputs of an LLM are representable with regular expressions, motivating using regular expressions to solve LLM validation tasks (§6). Our prototype of a regular expression engine for LLMs `ReLM` achieves a $15\times$ speedup or $2.5\times$ higher data efficiency over traditional approaches in memorization and toxicity finding. Additionally, it facilitates novel tests for bias and easily recovers prompt tuning behavior necessary for state-of-the-art zero-shot performance.

7.2 Future Directions and Extensions

While this thesis covers some aspects of data pipelines, there are many remaining problems which should be solved in future work. We outline more general problems first (§7.2.1). In the remaining subsections, we cover extensions that seem promising to particular work introduced in this thesis.

7.2.1 The Pipeline Interface and Primitives

While this thesis emphasizes methods to optimize data pipelines automatically, the best user-facing interface (and the primitives it supports) for input pipelines is still an open question. Systems such as `Plumber` (§5) can only react to the decisions made by the programmer, but it is often difficult or impossible to make complex decisions without the full programmer intent. For instance, it is difficult to cover the full space of optimizations in `Plumber` because the primitives exposed to users are coupled with the implementation (e.g., reading data is tied to how the data is stored). One cannot simply "update the schema" and hope it propagates to every implementation in existence. From the interface perspective, it's not clear whether an existing language such as SQL will work for data pipelines, or if some domain-specific modeling language (e.g., matrix-based in the case of training or regular expressions in the case of testing (§6)) will be appropriate.

Approximations. Related to the interface, there is potential to approximate many aspects of data pipelines, yet it is unsafe to do so, since full accuracy cannot be guaranteed. If approximation systems were to be fully automatic, they would have to be able to reason about the error introduced by approximations. While it may not be possible to entirely predict the impact of approximations, there is hope (and evidence §3, §4) that their impact could be sufficiently characterized such that empirical laws or strong heuristics could be developed. However, these heuristics would be enforced after the interface, raising questions about how one can "approximately" specify a pipeline to a system.

Large Scale Pipelines. Tuning a single pipeline can be very useful (§3), but tuning the pipeline in isolation may be suboptimal in terms of resource usage at a datacenter level, especially if a job is already highly distributed. Managing a fully distributed system of data pipelines would present different challenges, such as efficient management of the tuning sessions (e.g., sharing tuned parameters and metadata between jobs). It's not clear how the role of the data pipeline will evolve as the machine learning community embraces more complex systems with more stringent metrics of success (e.g., energy and carbon usage [322]).

Hardware Acceleration. Today, data pipelines are in still predominantly in the CPU era, and they can benefit from hardware specialization, such as JPEG decoding offload (§4). While there is some support for accelerator offload [214], such support is not ubiquitous and it's not portable across different accelerator vendors. Enabling data pipelines to run on a variety of accelerators is an area of future work.

7.2.2 Beyond Modeling Dataflow

`Plumber` (§5) optimizes data pipelines written in `tf.data`, which have a dataflow-like approach due to their functional nature. Similarly, research on Spark, which also has a functional set of abstractions, has led to other debugging tools [219, 220]. These tools are orthogonal in spirit to other profilers, which attempt to debug and make sense of general classes of programs with no constraints on how data is processed [30]. What general lessons can be learned from these works? Can performance tuning be both general and complete? Here, we outline some guidelines on how performance modeling can be achieved in more general systems.

- **Resource Accounting.** It seems clear that any performance tools must do resource accounting, since performance is intimately tied to resource usage. Resource accounting can be either done proactively by design [159, 220] or it can be done reactively in the traditional profiler sense [30]. The accounting must be done across resource types that impact performance, such as CPU time, disk I/O, memory usage, etc.
- **Framework Constraints.** Dataflow makes the entire computation explicit, removing the need for many modeling assumptions. Monotasks asserts that there is significant gain to be had by separating work explicitly by resource type [220] so that resource accounting can, by design, trigger alerts when a resource is oversubscribed (i.e., saturating some work queue). However, experience with `Plumber` contradicts this assertion, since `Plumber` is able to reason about operators using multiple resources (e.g., reading `TFRecords` uses both CPU and disk resources in one operator). Experience with `Plumber` indicates that *throughput dominant* workloads are remarkably simple to model. Throughput bottlenecks compose simply (e.g., using primitives from linear programming over averages) rather than requiring the mathematical heavy-lifting of computing the resulting latency distribution from a sequence of queues [119, 169]. From the practical side, the systems community has long known that latency performance is difficult to scale and is best avoided [68].

  What makes workloads like those in `tf.data` different from other systems? The primary reason that `tf.data` workloads are almost entirely throughput driven is that the "pipeline" is nearly always working with low work variations. Bottlenecks are therefore entirely driven by steady-state "average" behavior rather than any transient latency spikes (e.g., "stragglers") that appear at fine-grained timescales. The *key criteria* for the `Plumber` simplification are that pipeline work is: 1) consistent in distribution in the statistical sampling sense, 2) sufficiently long-running, 3) prefetched, and 4) multithreaded. The first two criteria are necessary for work estimation via measuring average behavior and the latter two are necessary for removing transient latency spikes. We additionally note that caching estimation is much easier in the machine learning setting, where data skew is low and sampling is near ideal. Concretely, machine learning workloads typically ensure that data is distributed independently and identically by carefully selecting the data and shuffling it afterwords. These statistical regularities allow `Plumber` to project cache accounting for subsets of the data to the full dataset.

It seems therefore that there are *two types of framework constraints at play*. First, there are latency-heavy workloads, such as those in Spark or general programs. Monotasks [220] proposed to simplify analysis by ensuring that operators are resource independent, allowing direct resource measurement at a resource queue level. Meanwhile, Magpie [30]'s approach would be to *infer* the cause of queueing without modifying the application. For these workloads, the bottleneck is a critical path in the execution graph. If the critical path is observed to have high resource usage over some of the resource types, then it's clear that adding more resources of those types would reduce the duration of the critical path. Nevertheless, the critical path can only be reduced by some degree, as there is always intrinsic latency from sequential processes in the system—adding infinitely more resources does not necessarily reduce latency [220]. Second, there are throughput-heavy workloads, such as those in `tf.data`. Systems like `Plumber` only have to track running averages of each resource type to estimate the end-to-end performance without requiring resource independence across operators. For these workloads, the bottleneck is insufficient resources or an inability to utilize the resources (i.e., a sequential implementation). Unlike latency workloads, throughput-heavy workloads can be scaled as a function of resources assuming that work is sufficiently independent.

- **Framework Knobs.** Lastly, we look at what criteria are necessary to act on profiler information. For general programs [30], there are no "knobs" to tune—the programmer must rewrite the program by hand or add resources. Even with an accurate model of Spark runtimes, semi-automatic changes are limited to optimizations such as caching deserialized data in memory and adding more resources [220]. Experience with `Plumber` similarly requires automatic changes to happen using the semantics of operators implemented via iterators [111]. These changes are limited to increasing the thread count, adding caches, and adding prefetch buffers. It thus seems that the primary "knobs" that can be tuned are ones directly related to a straightforward consumption of a resource type rather than any control logic. The most common mechanism for adding a knob is to place it in the operator itself (e.g., number of threads) or add a new operator that composes with other iterators [111].

7.2.3 Modeling Secondary Storage Caches

One notable extension to `Plumber` (§5) is using secondary storage (e.g., HDD and SSD) as a cache, which we analyze here. Part of this extension is straightforward, because `Plumber` already has a method for computing the materialized size of the cache, and thus can determine whether or not the secondary storage device has enough capacity to store the materialized data. However, the primary limitation for secondary storage is bandwidth and not materialized cache size. However, we note that `Plumber` *already* optimizes bandwidth I/O at the `Dataset` file-reading level. As we will see, secondary storage caching is actually a combination of two primitive models already used in `Plumber`: an ideal cache and an I/O device. Intuitively, secondary storage caching is merely an ideal cache from the CPU perspective (e.g., CPU work is reduced from caching), but it has an ideal I/O device of limited bandwidth as a data source.

As a concrete example, let us assume the materialized `Dataset` size is B for N samples. From this, we can calculate the average size of a data sample, x. For each sample x with size $s(x)$, we'd expect its size to be $\mathbb{E}_{x\sim\mathcal{D}}[s(x)] = B/N$ bytes per sample. Then with I/O bandwidth constraints of W, we'd expect $W/\mathbb{E}_{x\sim\mathcal{D}}[s(x)]$ samples per second, as discussed in Section 4.4.1. Note that $\lim_{W\to\infty} W/E_{\mathcal{D}}[s]$ grows unbounded—a device with infinite bandwidth has only a cache size limitation. Using this modeling framework, `Plumber` models in-memory caches as infinite bandwidth devices.

We can revisit `Plumber`'s performance model (§5.1.3) using some concrete numbers. To demonstrate the throughput bounds of a storage device, assuming we are storing raw image tensors (i.e., post image decompression and cropping) of a square size. We plot the estimated image throughput assuming images are stored in 8-bit RGB (i.e., 3 bytes per pixel) 32-bit RGB (i.e., 3 32-bit floats per pixel). For reference, we show approximate JPEG numbers assuming that JPEG compresses an image size by the multiplicative constant 0.2, as can be deduced from the ImageNet example in Figure 5.1[1] We assume the following device performance: HDD has 150MB/s, Low-End SSD has 500MB/s, High-End SSD has 10GB/s, and Memory has 100GB/s. The results are shown in Figure 7.1. We can see that in-memory caches of this bandwidth can supply tens of thousands of 32-bit RGB images per second for commonly used ranges of training (close to 224×224). However, an HDD can only supply hundreds of 32-bit RGB images for the same range. In this situation, the system would be likely be better off reading compressed JPEG images, which it can read at a rate of thousands per second. However, reading JPEG images is only better assuming that there are enough CPUs to do the JPEG decompression work.

Mathematically, we'd like to use the faster of the two configurations if both are possible (e.g., caching after decode and crop will fit in the secondary memory). To express this, let the throughput be expressed as the better of the two configurations: $X = \max\left(\hat{X_{\text{cache}}}(W), \hat{X_{\text{no\_cache}}}\right)$. $\hat{X_{\text{no\_cache}}}$ would be the throughput without any caching. $\hat{X_{\text{cache}}}(W) = \min(W/\mathbb{E}_{x\sim\mathcal{D}'}[s(x)], \hat{X_{\text{ideal\_cache}}})$ would be the throughput with caching data $\mathcal{D}'$ using an I/O device with W bandwidth, where $\hat{X_{\text{ideal\_cache}}}$ is the throughput assuming an infinite bandwidth device, as `Plumber` assumes for in-memory caches. All of these measures of throughput are in samples per second due to using samples rather than batches for the throughput bounds $W/\mathbb{E}_{x\sim\mathcal{D}'}[s(x)]$. The conversion to batches per second is explained in §5.1.4 using "visit ratios".

One can obtain this behavior from algorithm in two ways. For linear pipelines (e.g., without exotic "joining" between different data sources), a greedy cache-insertion approach can still work. A cache should be put as high up in the pipeline (i.e., closest to `Dataset` root), assuming: 1) the materialized size of the data will fit in the cache, and 2) the current throughput of the pipeline X given the caching device's bandwidth constraints is not lowered. Importantly, the estimated throughput of the pipeline should reflect prior caching options (e.g., a cache is only inserted if it is "faster" than caching lower in the `Dataset`). For the simple case we describe where we can choose between materializing decompressed RGB images and not, this algorithm would work as follows. First, `Plumber` would calculate the throughput without caching $\hat{X_{\text{no\_cache}}}$, which corresponds to the "default" of reading compressed images from storage. For HDD using

[1]This is a crude approximation as JPEG compression efficacy is data-dependent and tunable. Furthermore, the effect of image cropping to 224×224 is rolled in to this approximation.

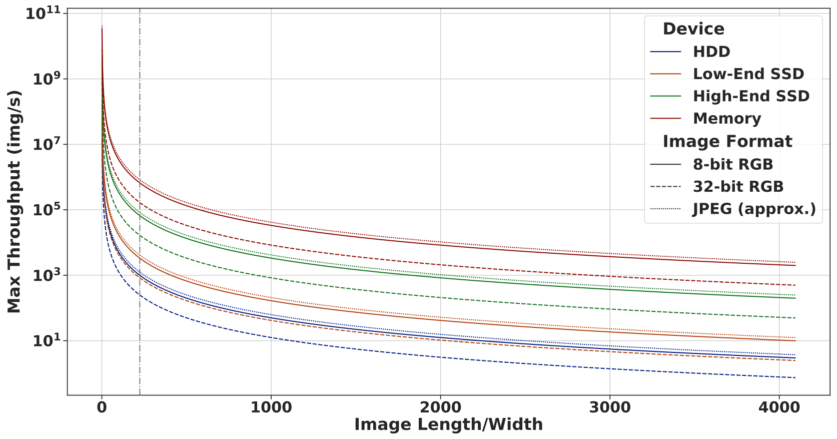


Figure 7.1: The limits of caching by device type and uncompressed image representation. x-axis represents the image length (or width) for a square image and thus contributes quadratically to the area. y-axis represents the maximum throughput reading images from the cache on a log scale. For reference, the 224×224 image length/width used in many vision benchmarks is annotated with a (gray) dashed vertical line. The decision to use `Plumber`'s caching technique for secondary storage would have to account for these bandwidth limits, which deviate from `Plumber`'s "idealized" version of an in-memory cache.

224×224 JPEG images, the disk bound is roughly 1250 images/second. `Plumber` could then compare this to reading decompressed 32-bit RGB images cached on the HDD, which would be roughly 250 images/second. These would be the two respective `Dataset` source constraints and both would be treated effectively the same as `Plumber`'s "disk" calculation (§5.1.3). The only difference is that $X_{\hat{\text{no\_cache}}}$ would include CPU work for decompressing and cropping images in the linear program (as usual), and $X_{\hat{\text{cache}}}$ would not (CPU work would be bound by $X_{\hat{\text{ideal\_cache}}}$). Thus, caching on secondary storage is equivalent to solving a "fused" problem between idealized caching and disk I/O.

This procedure can be generalized for `Datasets` of tree-like form using an integer Linear Program by adding a binary decision variable to account for all possibilities of where to put a cache, and what device to cache with. We note that this formulation can also be used with Progressive Compressed Records (§4) by having another tuning "knob" over the scan group which incorporates constraints on the image fidelity (e.g., gradient similarity). For future work, it would be interesting to see if, using a similar formulation, pipeline cost could be optimized given all the resource configurations available in public clouds (e.g., fine-grained configurables of CPU, memory, and I/O resources). One can imagine deciding between a single-core machine and a storage-cached `Dataset` or a many-core machine capable of recomputing the `Dataset`. For this type of formulation, one would seek the minimum cost configuration capable of saturating the accelerator.

7.2.4 Approximations for LLM Validation

ReLM (§6) offers a way to run arbitrary regular expressions easily with powerful traversal algorithms. However, best first algorithms, such as shortest path, are prone to large time and space complexities, leaving the user responsible for constructing queries that are efficient approximations of the task. For example, if the LLM is highly uniform in predictions over tokens, shortest path will degenerate to a breadth first traversal. In this case, the search will expand the search tree at a rate of b^d, for branching factor b (i.e., the vocabulary size) and depth, d. On top of this, ReLM has to maintain a queue of elements to visit, each of which consume $O(d)$ space. We note that random sampling, which ReLM supports, does not have this problem because it greedily samples at each step.

Experience with ReLM seems to indicate that time complexity is the more important issue of the two. For example, on the memorization task (§6.5), the size of the priority queue grows roughly linearly as a function of time, resulting in around 350k elements in the priority queue once 10k URLs are returned. The elements being processed at this point are 10s of tokens long, so we'd expect less than $100 \times (350 \times 10^3) \times 4$ bytes of space (i.e., assuming average length of 100 and 4 bytes per sample). Profiling resident memory agrees with this estimate; roughly 200MB was allocated over the process when starting with 5.6GB of resident memory. While the worst-case size of the query's automaton has the same exponential worst-case behavior (e.g., consider the query matching on everything, `.*`, with exactly one string removed), the practical memory usage is very tolerable. For example, the free-response over alphanumeric characters, `[a-zA-Z0-9]*`, matches on *infinite* strings of variable length, yet is 1 state with 15823 edges, which is a modest directed graph. Queries that aren't "high-entropy" will likely never hit the worst-case behavior.

An obvious extension to ReLM is to add better support for search approximations, such as beam search [1]. ReLM already has some support for beam search, though it has not been thoroughly evaluated due to a lack of motivation for validation. Specifically, the beam search's results are strongly coupled to the query—running a query with free response generation will select for different beams than if the query was multiple choice. Top-k, in contrast, is independent of the query—the top-k property stems from the LLM's predictions only, and changes to the query would have the same top-k results for any token sequences that are common between the queries. Nevertheless, if the user is willing to tolerate false negatives, this approach can be useful.

7.2.5 Applying **ReLM** to Larger Models and Prompts

ReLM can be applied to even larger models than GPT2-XL. For models that are hosted online (i.e., in an LLM-as-a-service format), the ReLM backend simply has to convert calls to a local model to calls to the remote model. For models hosted locally, these could be handled in a similar manner by self-hosting a distributed inference engine and connecting ReLM to it. While ReLM can work more-or-less "out of the box" for these larger models, it may be impractical in terms of inference cost. Some mix of query engineering combined with heuristics (e.g., beam search §7.2.4) will likely be necessary in these cases.

Similarly, users have an enormous space of "attack vectors" that can be used to invoke undesirable behavior in the LLM. As this search space grows exponentially (§7.2.4), it's not feasible to look through all of it. The best approach, it seems, is to be proactive with validation heuristics while also using runtime constraints to detect and correct unvalidated and potentially incorrect responses to queries. Systems, such as `ReLM`, could be used to either "rewrite" the output of the model or to simply avoid particular patterns during the sampling process.

Lessons For LLM Tokenization

LLM tokenization, and BPE [98] in particular, is a crucial part of LLM training (and thus inference). However, while BPE can help with compressing the inputs and outputs, it creates new attack vectors for the model, such as an inherent intolerance to character-level perturbations. For instance, certain "glitch" tokens have recently been found to create a failure mode [251] in GPT models. In the general case, the number of tokenizations of a string of length n grows exponentially (§6), meaning the validation surface also increases exponentially. In computer architecture, there was a long debate on "reduce instruction set computers" (RISC) compared to "complex instruction set computers" (CISC) [125]. The argument was that the simple "reduced" instructions would be easier to maintain in the long term, even though the "complex" instructions could do impressive tasks in hardware alone. There seems to be a similar argument accruing for character-based encodings compared to BPE. It would be interesting to explore to what extent the token space of LLMs could be simplified to a reduced set of tokens.

7.2.6 Beyond Regex: New Grammars and Modalities

Even though LLM output can be expressed with regular grammars, non-regular grammars may be beneficial in certain domains, such as the generation of code in specific programming languages. Furthermore, generative models may act over images, audio, or video, as shown in Figure 1.2. Would each of these domains be dealt with piecemeal or would some unifying idea be able to validate them all?

For grammars, it's still not clear what the implications of using a more powerful grammar are. It's certainly true that regular expressions are not efficient at expressing certain concepts, such as matching parenthesis or other context-free grammar constructs, though they can emulate them. Programming languages seem the most natural fit for stronger grammars, since they are formally defined via more powerful grammars. However, the tradeoff is that the user and the system have to deal with reasoning about a more powerful grammar, which may introduce performance and correctness concerns. Furthermore, languages such as `Python` are not fully context-free; thus, it may be the case that validating `Python` would be best served with a custom `Python` interpreter. In any case, it seems the adoption of particular grammars may follow the same path of the adoption of programming languages—there are factors beyond the theoretical ones that would drive adoption, and no single grammar would be the winner.

The other limitation of `ReLM` is its focus on natural language. `ReLM`, being a regular expression engine, is not applicable to other modalities, such as images, audio, or video tasks. However, ideas from `ReLM` could be applied to these other modalities with additional work. For example, suppose a model is tasked with scanning license plates found in a user's garage. Now suppose that in this particular task, the user is the owner of exactly two cars and knows that there are only two possible license plates: 123456 and 654321. In terms of strings, the language can be expressed with the regular expression `(123456)|(654321)`. If the model were to predict

65432$\underline{7}$, then we'd know that was an error, since that's outside the language. Just as `ReLM` maximizes the probability of a sequence of tokens, a system could maximize the probability of a sequence of vision primitives given some prior belief (e.g., that there are only two possible answers). It's unlikely, however, that the vision primitives could themselves be programmed in this manner, since that would enable solving the vision task without any learning at all. This sort of "composition" idea can be expressed with audio and video; indeed one of the primary applications of transducers is the construction of acoustic models via composition [16].

7.3 Reflections on ML Systems in the Real World

In some cases, the application of machine learning seems disconnected from the machine learning systems community. Since the deep learning revolution started, ML research has focused on the newest models and standardized datasets. There is a good reason for this—standardization facilitates comparisons and new models motivate new research. However, without knowing precisely how a benchmark fits into real applications, it's not clear how important the results of the benchmark are in the context of the whole machine learning workflow. Because benchmarks act as a proxy for progress in the field, it is always worth reflecting on whether or not the benchmarks reflect real progress.

The difference between benchmarks and reality is already being noted. For example, in the TPUv4 paper [141], the Deep Learning Recommendation Model (DLRM) from MLPerf is said to be unrealistic, citing work on industrial recommendation systems [322]. However, the goal of MLPerf is to be a "representative benchmark suite for ML that fairly evaluates system performance" [192]. While DLRM may have been reflective when it was added as a benchmark, it seems that it (and perhaps other benchmarks) are no longer representative. The few entities who control MLPerf should continue to update MLPerf benchmarks such that they reflect recent production workload trends and drive the community's goals to relevant problems. This quote from reflections on the Standard Performance Evaluation Corporation (SPEC) benchmark bears warning about how quickly benchmarks can lose their reputation: "The reputation of current benchmarketing claims regarding system performance is on par with the promises made by politicians during elections." [76].

Creating benchmarks is difficult and impossible to get perfectly correct; however, the fact that the ML systems community, which serves one of the fastest moving technical communities, doesn't proactively assess potential shortcomings means that the wrong problems are being worked on almost surely. With regards to this thesis, standard benchmark pipelines (e.g., MLPerf training [192]) are potentially misrepresented for a few reasons, which are outlined below:

- **Old Research Datasets.** The datasets used in benchmarks is academic data meant for proof-of-concept rather than a real product. Not only is this data typically acquired in a manner that was most convenient at the time (e.g., at lowest cost), the data is old. ImageNet [71] is over a decade past its creation—data in use today may be measurably different than it used to be. For instance, in MLPerf inference, one of the datasets is not used in native resolution—instead it is upscaled to more closely match high resolution sensors, which are found in automotive and industrial-automation applications [242]. If the data must be upscaled, then clearly the data is not representative of the expected image's information density and mere resizing means the model is doing more work without benefiting from additional predictive information[2]. Camera technology improves constantly (e.g., due to algorithmic innovation or technology scaling such as Moore's Law [209]), so it's reasonable to expect that all datasets will drift away from reality over time. Furthermore, the data will likely be larger and the infrastructure used may be more complex (e.g., a distributed datastore). Thus, it's reasonable to expect that the complexity with regard to all aspects of data is increasing, which may add unexpected strain throughout the data system stack e.g., Figure 7.1.

- **Simple Preprocessing.** The training-time preprocessing and transformations used in these benchmarks are remarkably simple and enforced by MLPerf rules. It seems unlikely that there hasn't been a single advancement in data pre-processing since the reference model was published [33, 61, 62, 90, 240, 280]. A pertinent example is the default 224×224 training-set image cropping used in many works (dating even back to AlexNet [156]), which was later found to be too large to match the statistics of the similarly-configured validation set [280]. Not only does making the image smaller speed up training, it makes the accuracy higher—thus, using 224×224 images overemphasizes the accelerator and adds an unnecessary source of error to the training process. Thus, it's reasonable to expect that pipelines used in practice are more sophisticated than those used in benchmarks—researchers may be researching novel augmentations while practitioners may be using them on top of complex data transformation code.
- **Workload Feedback and Drift.** The workload characteristics are not reflective of industrial use-cases because the data and model can be more complex [141, 322]. Unlike research datasets, which are static, the data and model are *jointly optimized* to account for sources of statistical signal found through a tuning process (e.g., A/B testing). The end result is that each job has a dynamic amount of features (e.g., for embedding generation) from the point of view of the data system [141, 322]. Furthermore, workloads can change on a bi-monthly to yearly basis [141], which is faster than a benchmark can be expected to change. However, it seems possible for benchmarks to be constructed such that there is some level of agreement between industrial use cases and benchmarks. For instance, if the number of features is expected to vary over time in production use-cases [141], then it seems reasonable for benchmarks to be constructed such that they can be instantiated with a number of features m where $a \leq m \leq b$ for some reasonable bounds of a and b (e.g., 0 to a million). Ironically, it seems that synthetically generated data could be even *more realistic* than datasets that are a decade past publication. Now that humans are being used to monitor model training [7, 8] and provide a reward model for training [5, 326], it seems harder than ever to create accurate and objective benchmarks. It's thus reasonable to expect that separation between benchmarks and reality will continue to diverge as workflows become more ad-hoc and dynamic.

Work that helps confirm or deny the realism of benchmarks in industry as well as research labs seems to be one of the most impactful problems in the ML systems community, especially as large models are turning research from a transparent community to one which is closed. Nevertheless,

[2]Consider how uninteresting it would be to resize a 1×1 pixel to some large dimensions.

while the community may never converge on perfect benchmarks, it should always be possible to update and improve upon them over time to reduce known differences. Indeed, just as machine learning has changed dramatically since AlexNet in 2012 [156], we should expect the benchmarks of today to hold much less value a decade from now.

Bibliography

[1] Speech understanding systems: Summary of results of the five-year research effort at carnegie-mellon university. Technical report, Carnegie-Mellon University, 1977. 7.2.4

[2] Apache Beam: An advanced unified programming model. `https://beam.apache.org/`, 2021. 2.1.1

[3] NEURIPS data-centric AI workshop. `https://datacentricai.org/neurips21/`, 2021. 1, 1.3, 1.3.1

[4] Apache Flume. `https://flume.apache.org/`, 2021. 2.1.1

[5] Introducing chatgpt. `https://openai.com/blog/chatgpt`, 2022. Accessed: 04-12-2023. 1.2.1, 4, 7.3

[6] Data-centric AI Resource Hub. `https://datacentricai.org/`, 2022. 1.3

[7] Opt-175 logbook. `https://github.com/facebookresearch/metaseq/blob/main/projects/OPT/chronicles/OPT175B_Logbook.pdf`, 2022. Accessed: 04-12-2023. 1.2, 7.3

[8] The final training. `https://github.com/bigscience-workshop/bigscience/blob/master/train/tr11-176B-ml/chronicles.md`, 2022. Accessed: 04-12-2023. 1.2, 7.3

[9] Top500 june 2022. `https://www.top500.org/lists/top500/2022/06/`, 2022. 1.3.1

[10] Daniel Abadi, Samuel Madden, and Miguel Ferreira. Integrating compression and execution in column-oriented database systems. In *ACM SIGMOD International Conference on Management of Data*, page 671–682, 2006. 4.6

[11] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. 1, 1.1.2, 4.1

[12] Sami Abu-El-Haija, Nisarg Kothari, Joonseok Lee, Paul Natsev, George Toderici, Balakr-

ishnan Varadarajan, and Sudheendra Vijayanarasimhan. Youtube-8m: A large-scale video classification benchmark. *arXiv preprint arXiv:1609.08675*, 2016. 4.6

[13] Victor Agababov, Michael Buettner, Victor Chudnovsky, Mark Cogan, Ben Greenstein, Shane McDaniel, Michael Piatek, Colin Scott, Matt Welsh, and Bolian Yin. Flywheel: Google's data compression proxy for the mobile web. In *USENIX Symposium on Networked Systems Design and Implementation*, pages 367–380, 2015. 4.6

[14] Alex Aizman, Gavin Maltby, and Thomas Breuel. High performance I/O for large scale deep learning. In *IEEE International Conference on Big Data*, pages 5965–5967, 2019. 4.6

[15] Dan Alistarh, Demjan Grubic, Jerry Li, Ryota Tomioka, and Milan Vojnovic. QSGD: Communication-efficient SGD via gradient quantization and encoding. In *Advances in Neural Information Processing Systems*, 2017. 4.1, 4.6

[16] Cyril Allauzen, Michael Riley, Johan Schalkwyk, Wojciech Skut, and Mehryar Mohri. Openfst: A general and efficient weighted finite-state transducer library. In *International Conference on Implementation and Application of Automata*, pages 11–23. Springer, 2007. 6.4.2, 7.2.6

[17] Cyril Allauzen, Bill Byrne, Adrià de Gispert, Gonzalo Iglesias, and Michael Riley. Pushdown automata in statistical machine translation. *Computational Linguistics*, 40(3):687–723, September 2014. doi: 10.1162/COLI_a_00197. 6.6

[18] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Spring Joint Computer Conference*, page 483–485, 1967. 4.4.1

[19] Antreas Antoniou, Amos Storkey, and Harrison Edwards. Data augmentation generative adversarial networks. *arXiv preprint arXiv:1711.04340*, 2017. 3.2

[20] Sajid Anwar, Kyuyeon Hwang, and Wonyong Sung. Fixed point optimization of deep convolutional neural networks for object recognition. In *International Conference on Acoustics, Speech and Signal Processing*, pages 1131–1135, 2015. 4.6

[21] ApacheBeam. Apachebeam. `https://www.tensorflow.org/datasets/beam_datasets`. Accessed: 07-26-2021. 4.2

[22] Apex. Nvidia apex. `https://github.com/NVIDIA/apex`. Accessed: 07-26-2021. 4.5.1

[23] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. Spark SQL: Relational data processing in Spark. In *Conference on Management of Data*, 2015. 2.1.1

[24] PyTorch Authors. Iterable-style datapipe. `https://pytorch.org/data/main/torchdata.datapipes.iter.html`, 2023. Accessed: 04-19-2023. 2

[25] Jens Axboe. fio-flexible i/o tester. `https://fio.readthedocs.io/en/latest/fio_doc.html`, 2021. 5.1.3, 5.2.2, .1

[26] Olivier Bachem, Mario Lucic, and Andreas Krause. Practical coreset constructions for machine learning. *arXiv preprint arXiv:1703.06476*, 2017. 4.6

[27] Johannes Ballé. Efficient nonlinear transforms for lossy image compression. In *Picture Coding Symposium*, pages 248–252, 2018. 4.6

[28] Johannes Ballé, David Minnen, Saurabh Singh, Sung Jin Hwang, and Nick Johnston. Variational image compression with a scale hyperprior. In *International Conference on Learning Representations*, 2018. 4.6

[29] Paul Barham and Michael Isard. Machine learning systems are stuck in a rut. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, 2019. 1

[30] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. Using magpie for request extraction and workload modelling. 2004. 7.2.2, 7.2.2

[31] Ronen Basri, David Jacobs, Yoni Kasten, and Shira Kritchman. The convergence rate of neural networks for learned functions of different frequencies. In *Advances in Neural Information Processing Systems*, 2019. 4.6

[32] Denis Baylor, Eric Breck, Heng-Tze Cheng, Noah Fiedel, Chuan Yu Foo, Zakaria Haque, Salem Haykal, Mustafa Ispir, Vihan Jain, Levent Koc, Chiu Yuen Koo, Lukasz Lew, Clemens Mewald, Akshay Naresh Modi, Neoklis Polyzotis, Sukriti Ramesh, Sudip Roy, Steven Euijong Whang, Martin Wicke, Jarek Wilkiewicz, Xin Zhang, and Martin Zinkevich. TFX: A tensorflow-based production-scale machine learning platform. In *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1387–1395, 2017. 1.1.2, 2.3, 4.6

[33] Irwan Bello, William Fedus, Xianzhi Du, Ekin Dogus Cubuk, Aravind Srinivas, Tsung-Yi Lin, Jonathon Shlens, and Barret Zoph. Revisiting resnets: Improved training and scaling strategies. In *Advances in Neural Information Processing Systems*, 2021. 7.3

[34] Emily M Bender, Timnit Gebru, Angelina McMillan-Major, and Shmargaret Shmitchell. On the dangers of stochastic parrots: Can language models be too big? In *Proceedings of the ACM Conference on Fairness, Accountability, and Transparency*, pages 610–623, 2021. 6.2

[35] Yoshua Bengio, Yann Lecun, and Geoffrey Hinton. Deep learning for AI. *Communications of the ACM*, 64(7):58–65, 2021. 1

[36] Luca Beurer-Kellner, Marc Fischer, and Martin Vechev. Prompting is programming: A query language for large language models. *arXiv preprint arXiv:2212.06094*, 2022. 6.6

[37] Alex Beutel, Ed H. Chi, Ellie Pavlick, Emily Blythe Pitler, Ian Tenney, Jilin Chen, Kellie Webster, Slav Petrov, and Xuezhi Wang. Measuring and reducing gendered correlations in pre-trained models. *arXiv preprint arXiv:2010.06032*, 2020. 6.5.2

[38] Steven Bird, Edward Loper, and Ewan Klein. *Natural Language Processing with Python*. O'Reilly Media Inc., 2009. 6.5.4

[39] Christopher M. Bishop. *Pattern recognition and machine learning*. Springer, 2006. 3.5

[40] Rishi Bommasani, Drew A. Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S. Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill, Erik Brynjolfsson, Shyamal Buch, Dallas Card, Rodrigo Castellon, Niladri Chatterji, Annie Chen, Kathleen Creel, Jared Quincy Davis, Dora Demszky, Chris Donahue, Moussa

Doumbouya, Esin Durmus, Stefano Ermon, John Etchemendy, Kawin Ethayarajh, Li Fei-Fei, Chelsea Finn, Trevor Gale, Lauren Gillespie, Karan Goel, Noah Goodman, Shelby Grossman, Neel Guha, Tatsunori Hashimoto, Peter Henderson, John Hewitt, Daniel E. Ho, Jenny Hong, Kyle Hsu, Jing Huang, Thomas Icard, Saahil Jain, Dan Jurafsky, Pratyusha Kalluri, Siddharth Karamcheti, Geoff Keeling, Fereshte Khani, Omar Khattab, Pang Wei Koh, Mark Krass, Ranjay Krishna, Rohith Kuditipudi, Ananya Kumar, Faisal Ladhak, Mina Lee, Tony Lee, Jure Leskovec, Isabelle Levent, Xiang Lisa Li, Xuechen Li, Tengyu Ma, Ali Malik, Christopher D. Manning, Suvir Mirchandani, Eric Mitchell, Zanele Munyikwa, Suraj Nair, Avanika Narayan, Deepak Narayanan, Ben Newman, Allen Nie, Juan Carlos Niebles, Hamed Nilforoshan, Julian Nyarko, Giray Ogut, Laurel Orr, Isabel Papadimitriou, Joon Sung Park, Chris Piech, Eva Portelance, Christopher Potts, Aditi Raghunathan, Rob Reich, Hongyu Ren, Frieda Rong, Yusuf Roohani, Camilo Ruiz, Jack Ryan, Christopher Ré, Dorsa Sadigh, Shiori Sagawa, Keshav Santhanam, Andy Shih, Krishnan Srinivasan, Alex Tamkin, Rohan Taori, Armin W. Thomas, Florian Tramèr, Rose E. Wang, William Wang, Bohan Wu, Jiajun Wu, Yuhuai Wu, Sang Michael Xie, Michihiro Yasunaga, Jiaxuan You, Matei Zaharia, Michael Zhang, Tianyi Zhang, Xikun Zhang, Yuhui Zhang, Lucia Zheng, Kaitlyn Zhou, and Percy Liang. On the opportunities and risks of foundation models. *arXiv preprint arXiv:2108.07258*, 2021. 1.2, 1.2, 1.3.1, 6.1, 6.2

[41] Léon Bottou, Frank E Curtis, and Jorge Nocedal. Optimization methods for large-scale machine learning. *Siam Review*, 60(2):223–311, 2018. 1.1.2

[42] Samuel R. Bowman and George E. Dahl. What will it take to fix benchmarking in natural language understanding? In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 4843–4855. Association for Computational Linguistics, 2021. 6.2, 6.3.1, 6.3.2

[43] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, 2020. 1, 1, 1.2, 1.3.1, 6.2, 6.3.1, 6.5.4

[44] Christopher J. C. Burges and Bernhard Schölkopf. Improving the accuracy and speed of support vector machines. In *Advances in Neural Information Processing Systems*, 1996. 3.1, 3.2, 3.5.1, 7.1

[45] V. C. Cabezas and M. Püschel. Extending the roofline model: Bottleneck analysis with microarchitectural constraints. In *International Symposium on Workload Characterization*, 2014. 5.3

[46] Nicholas Carlini, Chang Liu, Úlfar Erlingsson, Jernej Kos, and Dawn Song. The secret sharer: Evaluating and testing unintended memorization in neural networks. In *USENIX Security Symposium*, pages 267–284, 2019. 6.4.3, 6.5.1

[47] Nicholas Carlini, Florian Tramer, Eric Wallace, Matthew Jagielski, Ariel Herbert-Voss,

Katherine Lee, Adam Roberts, Tom Brown, Dawn Song, Ulfar Erlingsson, Alina Oprea, and Colin Raffel. Extracting training data from large language models. In *USENIX Security Symposium*, pages 2633–2650, 2021. 6.3.4, 6.5.1, 6.5.1

[48] Nicholas Carlini, Daphne Ippolito, Matthew Jagielski, Katherine Lee, Florian Tramer, and Chiyuan Zhang. Quantifying memorization across neural language models. In *International Conference on Learning Representations*, 2023. 6.2, 6.4.4, 6.5.1

[49] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. 2015. 1, 1.1.2

[50] Yu Cheng, Duo Wang, Pan Zhou, and Tao Zhang. A survey of model compression and acceleration for deep neural networks. *arXiv preprint arXiv:1710.09282*, 2017. 4.6

[51] Steven WD Chien, Stefano Markidis, Chaitanya Prasad Sishtla, Luis Santos, Pawel Herman, Sai Narasimhamurthy, and Erwin Laure. Characterizing deep-learning i/o workloads in tensorflow. In *International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems*, pages 54–63, 2018. 4.6

[52] Ting-Wu Chin, Ruizhou Ding, and Diana Marculescu. Adascale: Towards real-time video object detection using adaptive scaling. In *Machine Learning and Systems*, 2019. 4.6

[53] Dami Choi, Alexandre Passos, Christopher J. Shallue, and George E. Dahl. Faster neural network training with data echoing. *arXiv preprint arXiv:1907.05550*, 2019. 3.6, 4.6, 5.1.1, 5.3

[54] Alexandra Chouldechova and Aaron Roth. A snapshot of the frontiers of fairness in machine learning. *Communications of the ACM*, 63(5):82–89, 2020. 6.5.2

[55] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311*, 2022. 1, 1, 1.3.1, 6.2

[56] Fahim Chowdhury, Yue Zhu, Todd Heer, Saul Paredes, Adam Moody, Robin Goldstone, Kathryn Mohror, and Weikuan Yu. I/O characterization and performance evaluation of beegfs for deep learning. In *Proceedings of the 48th International Conference on Parallel Processing*, pages 1–10, 2019. 4.6

[57] Dan Claudiu Cireşan, Ueli Meier, Luca Maria Gambardella, and Jürgen Schmidhuber.

Deep, big, simple neural nets for handwritten digit recognition. *Neural Computation*, 22 (12):3207–3220, 2010. 3.1, 3.2

[58] Torch Contributors. PyTorch Docs: torch.utils.data. `https://pytorch.org/docs/stable/data.html`, 2019. 2.1.1

[59] R. Dennis Cook. Assessment of local influence. *Journal of the Royal Statistical Society. Series B (Methodological)*, 48(2):133–169, 1986. 3.2

[60] Paul Covington, Jay Adams, and Emre Sargin. Deep neural networks for youtube recommendations. In *Proceedings of the 10th ACM Conference on Recommender Systems*, New York, NY, USA, 2016. 1

[61] Ekin Dogus Cubuk, Barret Zoph, Dandelion Mane, Vijay Vasudevan, and Quoc V. Le. Autoaugment: Learning augmentation policies from data. In *Conference on Computer Vision and Pattern Recognition*, 2019. 3.2, 7.3

[62] Ekin Dogus Cubuk, Barret Zoph, Jon Shlens, and Quoc Le. Randaugment: Practical automated data augmentation with a reduced search space. In *Advances in Neural Information Processing Systems*, 2020. 7.3

[63] Henggang Cui, James Cipar, Qirong Ho, Jin Kyu Kim, Seunghak Lee, Abhimanu Kumar, Jinliang Wei, Wei Dai, Gregory R. Ganger, Phillip B. Gibbons, Garth A. Gibson, and Eric P. Xing. Exploiting bounded staleness to speed up big data analytics. In *USENIX Annual Technical Conference*, page 37–48, 2014. 4.1

[64] Henggang Cui, Hao Zhang, Gregory R. Ganger, Phillip B. Gibbons, and Eric P. Xing. GeePS: Scalable deep learning on distributed GPUs with a GPU-specialized parameter server. In *European Conference on Computer Systems*, pages 1–16, 2016. 4.1

[65] Amit Daniely, Nevena Lazic, Yoram Singer, and Kunal Talwar. Short and deep: Sketching and neural networks. *arXiv preprint arXiv:1710.07850*, 2017. 4.6

[66] Nilaksh Das, Madhuri Shanbhogue, Shang-Tse Chen, Fred Hohman, Siwei Li, Li Chen, Michael E. Kounavis, and Duen Horng Chau. Shield: Fast, practical defense and vaccination for deep learning using jpeg compression. In *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 196–204, 2018. 4.6

[67] Nicola De Cao, Gautier Izacard, Sebastian Riedel, and Fabio Petroni. Autoregressive entity retrieval. In *International Conference on Learning Representations*, 2021. 6.6

[68] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 2013. 7.2.2

[69] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc'Aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, Quoc V. Le, and Andrew Y. Ng. Large scale distributed deep networks. In *Advances in Neural Information Processing Systems*, 2012. 4.1

[70] Dennis Decoste and Bernhard Schölkopf. Training invariant support vector machines. *Machine learning*, 46(1-3):161–190, 2002. 3.1, 3.2, 3.5.1, 7.1

[71] Jia Deng, Wei Dong, Richard Socher, Li-Jia. Li, Kai Li, and Li Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *Conference on Computer Vision and Pattern*

Recognition, pages 248–255, 2009. 1, 1.2.1, 2.1.1, 4.1, 4.2, 4.5.1, 5.2, 7.3

[72] Peter J Denning and Jeffrey P Buzen. The operational analysis of queueing network models. *ACM Computing Surveys*, 1978. 5.1.2, .1

[73] Emily L. Denton, Wojciech Zaremba, Joan Bruna, Yann LeCun, and Rob Fergus. Exploiting linear structure within convolutional networks for efficient evaluation. In *Advances in Neural Information Processing Systems*, 2014. 4.6

[74] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 4171–4186. Association for Computational Linguistics, 2019. 6.3.1

[75] Edsger Wybe Dijkstra. A note on two problems in connexion with graphs. In *Edsger Wybe Dijkstra: His Life, Work, and Legacy*, pages 287–290. 2022. 6.4.3

[76] K.M. Dixit. Overview of the spec benchmarks. In *The Benchmark Handbook*, 1993. 7.3

[77] Samuel Dodge and Lina Karam. Understanding how image quality affects deep neural networks. In *International Conference on Quality of Multimedia Experience*, pages 1–6, 2016. 4.6

[78] Yinpeng Dong, Qi-An Fu, Xiao Yang, Tianyu Pang, Hang Su, Zihao Xiao, and Jun Zhu. Benchmarking adversarial robustness on image classification. In *Conference on Computer Vision and Pattern Recognition*, pages 318–328, 2020. 4.6

[79] Alexey Dosovitskiy, Philipp Fischer, Jost Tobias Springenberg, Martin Riedmiller, and Thomas Brox. Discriminative unsupervised feature learning with exemplar convolutional neural networks. *Transactions on Pattern Analysis and Machine Intelligence*, 38(9):1734–1747, 2016. 3.1, 3.2

[80] Danny Driess, Fei Xia, Mehdi SM Sajjadi, Corey Lynch, Aakanksha Chowdhery, Brian Ichter, Ayzaan Wahid, Jonathan Tompson, Quan Vuong, Tianhe Yu, et al. Palm-e: An embodied multimodal language model. *arXiv preprint arXiv:2303.03378*, 2023. 1

[81] Petros Drineas, Michael W. Mahoney, S. Muthukrishnan, and Tamás Sarlós. Faster least squares approximation. *Numerische Mathematik*, 117(2):219–249, 2011. 3.2

[82] Petros Drineas, Malik Magdon-Ismail, Michael W. Mahoney, and David P. Woodruff. Fast approximation of matrix coherence and statistical leverage. *Journal of Machine Learning Research*, 13:3475–3506, 2012. 3.2

[83] Nikoli Dryden, Roman Böhringer, Tal Ben-Nun, and Torsten Hoefler. Clairvoyant prefetching for distributed machine learning I/O. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021. 5.3

[84] Jesse Dunietz, Greg Burnham, Akash Bharadwaj, Owen Rambow, Jennifer Chu-Carroll, and Dave Ferrucci. To test machine comprehension, start by defining comprehension. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 7839–7859, Online, July 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.acl-main.701. 6.3.2

[85] Adam Dziedzic, John Paparrizos, Sanjay Krishnan, Aaron Elmore, and Michael Franklin. Band-limited training and inference for convolutional neural networks. In *International Conference on Machine Learning*, 2019. 4.6

[86] Gintare Karolina Dziugaite, Zoubin Ghahramani, and Daniel M. Roy. A study of the effect of JPG compression on adversarial images. *arXiv preprint arXiv:1608.00853*, 2016. 4.6

[87] Tyna Eloundou, Sam Manning, Pamela Mishkin, and Daniel Rock. Gpts are gpts: An early look at the labor market impact potential of large language models. *arXiv preprint 2303.10130*, 2023. 1, 1.2

[88] Mark Everingham, Luc Van Gool, Christopher K. I. Williams, John Winn, and Andrew Zisserman. The pascal visual object classes (VOC) challenge. *International Journal of Computer Vision*, 88:303–338, 2010. 4.2

[89] Angela Fan, Mike Lewis, and Yann Dauphin. Hierarchical neural story generation. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 889–898. Association for Computational Linguistics, 2018. 6.3.4

[90] Alhussein Fawzi, Horst Samulowitz, Deepak Turaga, and Pascal Frossard. Adaptive data augmentation for image classification. In *International Conference on Image Processing*, pages 3688–3692, 2016. 3.2, 7.3

[91] Dan Feldman, Melanie Schmidt, and Christian Sohler. Turning big data into tiny data: Constant-size coresets for k-means, PCA and projective clustering. In *Symposium on Discrete Algorithms*, pages 1434–1453, 2013. 4.6

[92] First Conference on Machine Translation. WMT. `http://www.statmt.org/wmt16/`, 2016. 5.2

[93] William Fithian and Trevor Hastie. Local case-control sampling: Efficient subsampling in imbalanced data sets. *The Annals of Statistics*, 42(5):1693–1724, 2014. 3.4.2

[94] OpenAI Forums. Knowledge Cutoff Data of September 2021. 2023. 4

[95] Sadjad Fouladi, John Emmons, Emre Orbay, Catherine Wu, Riad S. Wahby, and Keith Winstein. Salsify: Low-latency network video through tighter integration between a video codec and a transport protocol. In *USENIX Symposium on Networked Systems Design and Implementation*, pages 267–282, 2018. 4.4.3

[96] Roy Frostig, Matthew Johnson, and Chris Leary. Compiling machine learning programs via high-level tracing. *Systems for Machine Learning*, 2018. 1, 1.1.2

[97] Dan Fu and Gabriel Guimaraes. Using compression to speed up image classification in artificial neural networks. Technical report, 2016. 4.6

[98] Philip Gage. A new algorithm for data compression. `https://www.drdobbs.com/a-new-algorithm-for-data-compression/184402829`, 1994. 6.3.4, 7.2.5, .2

[99] Leo Gao, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima, Shawn Presser, and Connor Leahy. The Pile: An 800gb dataset of diverse text for language modeling. *arXiv preprint*

arXiv:2101.00027, 2020. 6.5.3

[100] Leo Gao, Jonathan Tow, Stella Biderman, Sid Black, Anthony DiPofi, Charles Foster, Laurence Golding, Jeffrey Hsu, Kyle McDonell, Niklas Muennighoff, Jason Phang, Laria Reynolds, Eric Tang, Anish Thite, Ben Wang, Kevin Wang, and Andy Zou. A framework for few-shot language model evaluation. `https://github.com/EleutherAI/lm-evaluation-harness`, September 2021. 6.2

[101] Tianyu Gao, Adam Fisch, and Danqi Chen. Making pre-trained language models better few-shot learners. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 3816–3830, Online, August 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.acl-long.295. 6.3.2

[102] GCPDiskBandwidth. Gcpdiskbandwidth. `https://cloud.google.com/compute/docs/disks/performance`, 2021. Accessed: 07-26-2021. 4.1

[103] GCPNetworkBandwidth. Gcpnetworkbandwidth. `https://cloud.google.com/compute/docs/network-bandwidth`, 2021. Accessed: 07-26-2021. 4.1

[104] Samuel Gehman, Suchin Gururangan, Maarten Sap, Yejin Choi, and Noah A. Smith. RealToxicityPrompts: Evaluating neural toxic degeneration in language models. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 3356–3369. Association for Computational Linguistics, 2020. 6.2

[105] Robert Geirhos, Patricia Rubisch, Claudio Michaelis, Matthias Bethge, Felix A. Wichmann, and Wieland Brendel. Imagenet-trained CNNs are biased towards texture; increasing shape bias improves accuracy and robustness. In *International Conference on Learning Representations*, 2019. 4.6

[106] Carlos A. Gomez-Uribe and Neil Hunt. The netflix recommender system: Algorithms, business value, and innovation. *ACM Transactions on Management Information Systems*, 6 (4), dec 2016. 1

[107] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in Neural Information Processing Systems*, 2014. 3.2

[108] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`. 1

[109] Google Cloud. TPU Troubleshooting. `https://cloud.google.com/tpu/docs/troubleshooting`, 2021. 1.1.2

[110] 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*, 2017. 4.4.3, 4.5.1, 4.6

[111] Goetz Graefe. Volcano - an extensible and parallel query evaluation system. *Transactions on Knowledge and Data Engineering*, 1994. 2.1.1, 7.2.2

[112] Benjamin Graham. Fractional max-pooling. *arXiv preprint arXiv:1412.6071*, 2014. 3.1, 3.2

[113] Sam Gross and Michael Wilber. Training and investigating residual nets. 2016. 1.3.1

[114] Lionel Gueguen, Alex Sergeev, Ben Kadlec, Rosanne Liu, and Jason Yosinski. Faster neural networks straight from JPEG. In *Advances in Neural Information Processing Systems*, 2018. 4.6

[115] Joaquin Anton Guirao, Krzysztof Lecki, Janusz Lisiecki, Serge Panev, Michal Szolucha, Albert Wolant, and Michal Zientkiewicz. Fast AI Data Preprocessing with NVIDIA DALI. `https://devblogs.nvidia.com/fast-ai-data-preprocessing-with-nvidia-dali`, 2019. 2.1.1

[116] Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. In *Advances in Neural Information Processing Systems*, 2015. 4.6

[117] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. EIE: efficient inference engine on compressed deep neural network. In *International Symposium on Computer Architecture*, pages 243–254, 2016. 4.6

[118] Song Han, Huizi Mao, and William J. Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. In *International Conference on Learning Representations*, 2016. 4.6

[119] Mor Harchol-Balter. *Performance modeling and design of computer systems: queueing theory in action*. Cambridge University Press, 1st edition, 2013. 4.4.1, 7.2.2

[120] Thomas Hartvigsen, Saadia Gabriel, Hamid Palangi, Maarten Sap, Dipankar Ray, and Ece Kamar. ToxiGen: A large-scale machine-generated dataset for adversarial and implicit hate speech detection. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 3309–3326. Association for Computational Linguistics, 2022. 6.5.3

[121] Ahmed Hassan, Sara Noeman, and Hany Hassan. Language independent text correction using finite state automata. In *Proceedings of the Third International Joint Conference on Natural Language Processing: Volume-II*, 2008. 6.4.4

[122] Kim Hazelwood, Sarah Bird, David Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, Aditya Kalro, James Law, Kevin Lee, Jason Lu, Pieter Noordhuis, Misha Smelyanskiy, Liang Xiong, and Xiaodong Wang. Applied machine learning at facebook: A datacenter infrastructure perspective. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 620–629, 2018. doi: 10.1109/HPCA.2018.00059. 1.2

[123] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Conference on Computer Vision and Pattern Recognition*, 2016. 2.2.2, 4.1, 4.4.1, 4.4.3, 4.5.1, 5.2

[124] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity mappings in deep residual networks. In *European Conference on Computer Vision*, pages 630–645. Springer, 2016. 3.5

[125] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*.

Elsevier, 2011. 7.2.5

[126] David C. Hoaglin and Roy E. Welsch. The hat matrix in regression and ANOVA. *The American Statistician*, 32(1):17–22, 1978. 3.2

[127] Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. The curious case of neural text degeneration. In *International Conference on Learning Representations*, 2020. 6.3.4

[128] Sara Hooker. The hardware lottery. *Communications of the ACM*, 64(12):58–65, nov 2021. 1

[129] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. 3rd edition, 2007. 6.3.3

[130] HuggingFace. run_generation.py. `https://github.com/huggingface/transformers/blob/main/examples/pytorch/text-generation/run_generation.py`, 2021. 6.5.1

[131] Huggingface. bad_words_ids not working. `https://github.com/huggingface/transformers/issues/17504`, 2022. 6.6

[132] Kyuyeon Hwang and Wonyong Sung. Fixed-point feedforward deep neural network design using weights $+1$, 0, and -1. In *Workshop on Signal Processing Systems*, 2014. 4.6

[133] Andrew Ilyas, Shibani Santurkar, Dimitris Tsipras, Logan Engstrom, Brandon Tran, and Aleksander Madry. Adversarial examples are not bugs, they are features. In *Advances in Neural Information Processing Systems*, volume 32, 2019. 4.6

[134] Alexander Isenko, Ruben Mayer, Jedele Jeffrey, and Hans-Arno Jacobsen. Where is my training bottleneck? Hidden trade-offs in deep learning preprocessing pipelines. In *International Conference on Management of Data*, 2022. 1.3.1, 5.3

[135] Vilas Jagannath, Zuoning Yin, and Mihai Budiu. Monitoring and debugging DryadLINQ applications with daphne. In *International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, 2011. 5.3

[136] Xianyan Jia, Shutao Song, Wei He, Yangzihao Wang, Haidong Rong, Feihu Zhou, Liqiang Xie, Zhenyu Guo, Yuanzhou Yang, Liwei Yu, Tiegang Chen, Guangxiao Hu, Shaohuai Shi, and Xiaowen Chu. Highly scalable deep learning training system with mixed-precision: Training ImageNet in four minutes. In *Neural Information Processing Systems Workshop on Systems for ML*, 2018. 4.6

[137] Zhe Jia, Blake Tillman, Marco Maggioni, and Daniele Paolo Scarpazza. Dissecting the graphcore IPU architecture via microbenchmarking. *arXiv preprint arXiv:1912.03413*, 2019. 1.1.2

[138] Zhengbao Jiang, Frank F. Xu, Jun Araki, and Graham Neubig. How can we know what language models know? *Transactions of the Association for Computational Linguistics*, 8: 423–438, 2020. doi: 10.1162/tacl_a_00324. 6.3.2

[139] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben

Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. In *International Symposium on Computer Architecture*, volume 45, page 1–12, 2017. 4.1

[140] Norman P Jouppi, Doe Hyun Yoon, George Kurian, Sheng Li, Nishant Patil, James Laudon, Cliff Young, and David Patterson. A domain-specific supercomputer for training deep neural networks. *Communications of the ACM*, 2020. 1.1.2, 1.3.1, 2.2.2, 4.1, 5.2

[141] Norman P. Jouppi, George Kurian, Sheng Li, Peter C. Ma, Rahul Nagarajan, Lifeng Nai, Nishant Patil, Suvinay Subramanian, Andy Swing, Brian Towles, Cliff Young, Xiaoping Zhou, Zongwei Zhou, and David A. Patterson. TPU v4: An optically reconfigurable supercomputer for machine learning with hardware support for embeddings. *arXiv preprint arXiv:2304.01433*, 2023. 1.3.1, 7.3

[142] JPEGTran. JPEGTran libjpeg.txt. `https://github.com/cloudflare/jpegtran/blob/master/libjpeg.txt`, 2015. Accessed: 07-26-2021. 4.2

[143] John Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin Žídek, Anna Potapenko, Alex Bridgland, Clemens Meyer, Simon AA Kohl, Andrew J Ballard, Andrew Cowie, Bernardino Romera-Paredes, Stanislav Nikolov, Rishub Jain, Jonas Adler, Trevor Back, Stig Petersen, David Reiman, Ellen Clancy, Michal Zielinski, Martin Steinegger, Michalina Pacholska, Tamas Berghammer, Sebastian Bodenstein, David Silver, Oriol Vinyals, Andrew W Senior, Koray Kavukcuoglu, Pushmeet Kohli, and Demis Hassabis. Highly accurate protein structure prediction with alphafold. *Nature*, 596(7873):583–589, 2021. 1

[144] Alexander B. Jung. imgaug. `https://github.com/aleju/imgaug`, 2018. 3.5

[145] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020. 1.3.2

[146] Zohar Karnin and Edo Liberty. Discrepancy, coresets, and sketches in machine learning. In *Conference on Learning Theory*, volume 99, pages 1–19, 2019. 4.6

[147] Tero Karras, Timo Aila, Samuli Laine, and Jaakko Lehtinen. Progressive growing of GANs for improved quality, stability, and variation. In *International Conference on Learning Representations*, 2018. 4.1, 4.2, 4.5.1, 4.5.5, 4.6

[148] Qifa Ke, Michael Isard, and Yuan Yu. Optimus: A dynamic rewriting framework for data-parallel execution plans. In *European Conference on Computer Systems*, 2013. 5.3

[149] Zachary Kenton, Tom Everitt, Laura Weidinger, Iason Gabriel, Vladimir Mikulik, and Geoffrey Irving. Alignment of language agents. *arXiv preprint arXiv:2103.14659*, 2021. 1.2.1

[150] Douwe Kiela, Max Bartolo, Yixin Nie, Divyansh Kaushik, Atticus Geiger, Zhengxuan Wu, Bertie Vidgen, Grusha Prasad, Amanpreet Singh, Pratik Ringshia, Zhiyi Ma, Tristan Thrush, Sebastian Riedel, Zeerak Waseem, Pontus Stenetorp, Robin Jia, Mohit Bansal, Christopher Potts, and Adina Williams. Dynabench: Rethinking benchmarking in NLP. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 4110–4124. Association for Computational Linguistics, 2021. 6.2, 6.3.2

[151] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *International Conference on Learning Representations*, 2015. 1.1.2

[152] Hannah Rose Kirk, Yennie Jun, Filippo Volpin, Haider Iqbal, Elias Benussi, Frederic Dreyer, Aleksandar Shtedritski, and Yuki Asano. Bias out-of-the-box: An empirical analysis of intersectional occupational biases in popular generative language models. In *Advances in Neural Information Processing Systems*, 2021. 6.5.2, 6.5.2

[153] Pang Wei Koh and Percy Liang. Understanding black-box predictions via influence functions. In *International Conference on Machine Learning*, 2017. 3.2, 3.4.1

[154] Jonathan Krause, Michael Stark, Jia Deng, and Li Fei-Fei. 3d object representations for fine-grained categorization. In *Workshop on 3D Representation and Recognition*, 2013. 4.5.1

[155] Alex Krizhevsky. Learning multiple layers of features from tiny images. *Master's thesis, Department of Computer Science, University of Toronto*, 2009. 3.1

[156] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, 2012. 1, 1.1.2, 1.3.1, 2.1.1, 4.4.1, 4.5.1, 7.3

[157] Michael Kuchnik and Virginia Smith. Efficient augmentation via data subsampling. In *International Conference on Learning Representations*, 2019. 1.4.1

[158] Michael Kuchnik, George Amvrosiadis, and Virginia Smith. Progressive compressed records: Taking a byte out of deep learning data. In *Proceedings of Very Large Databases*, volume 14, 2021. 1.4.1, 5.3

[159] Michael Kuchnik, Ana Klimovic, Jiri Simsa, Virginia Smith, and George Amvrosiadis. Plumber: Diagnosing and removing performance bottlenecks in machine learning data pipelines. In *Machine Learning and Systems*, 2022. 1.4.1, 7.2.2

[160] Michael Kuchnik, Virginia Smith, and George Amvrosiadis. Validating large language models with relm. In *Machine Learning and Systems*, 2023. 1.4.1

[161] Alex Kulesza and Ben Taskar. Determinantal point processes for machine learning. *Foundations and Trends in Machine Learning*, 5(2–3):123–286, 2012. 3.6

[162] Abhishek Vijaya Kumar and Muthian Sivathanu. Quiver: An informed storage cache for deep learning. In *USENIX Conference on File and Storage Technologies*, pages 283–296,

2020. 4.1, 4.2, 4.6

[163] Deepak Kumar, Patrick Gage Kelley, Sunny Consolvo, Joshua Mason, Elie Bursztein, Zakir Durumeric, Kurt Thomas, and Michael Bailey. Designing toxic content classification for a diversity of perspectives. In *Seventeenth Symposium on Usable Privacy and Security (SOUPS 2021)*, pages 299–318. USENIX Association, August 2021. ISBN 978-1-939133-25-0. 6.5.3

[164] Sameer Kumar, Victor Bitorff, Dehao Chen, Chiachen Chou, Blake Hechtman, HyoukJoong Lee, Naveen Kumar, Peter Mattson, Shibo Wang, Tao Wang, et al. Scale mlperf-0.6 models on google TPU-v3 pods. *arXiv preprint arXiv:1909.09756*, 2019. 4.5.2, 4.6

[165] Sameer Kumar, James Bradbury, Cliff Young, Yu Emma Wang, Anselm Levskaya, Blake Hechtman, Dehao Chen, HyoukJoong Lee, Mehmet Deveci, Naveen Kumar, Pankaj Kanwar, Shibo Wang, Skye Wanderman-Milne, Steve Lacy, Tao Wang, Tayo Oguntebi, Yazhou Zu, Yuanzhong Xu, and Andy Swing. Exploring the limits of concurrency in ml training on google TPUs. *arXiv preprint arXiv:2011.03641*, 2020. 1, 1.1.2, 1.3.1, 4.1, 4.6

[166] Keita Kurita, Nidhi Vyas, Ayush Pareek, Alan W Black, and Yulia Tsvetkov. Measuring bias in contextualized word representations. In *Proceedings of the First Workshop on Gender Bias in Natural Language Processing*, pages 166–172. Association for Computational Linguistics, 2019. 6.5.2

[167] Thorsten Kurth, Sean Treichler, Joshua Romero, Mayur Mudigonda, Nathan Luehr, Everett Phillips, Ankur Mahesh, Michael Matheson, Jack Deslippe, Massimiliano Fatica, Prabhat, and Michael Houston. Exascale deep learning for climate analytics. In *International Conference for High Performance Computing, Networking, Storage, and Analysis*, number 51, pages 1–12, 2018. 4.1, 4.2, 4.5.2, 4.6

[168] Youngeun Kwon and Minsoo Rhu. Beyond the memory wall: A case for memory-centric hpc system for deep learning. In *IEEE/ACM International Symposium on Microarchitecture*, pages 148–161, 2018. 4.1

[169] Edward D. Lazowska, John Zahorjan, G. Scott Graham, and Kenneth C. Sevcik. *Quantitative System Performance: Computer System Analysis Using Queueing Network Models*. Prentice-Hall, Inc., USA, 1984. ISBN 0137469756. 2.1.2, 7.2.2

[170] Yann LeCun. A path towards autonomous machine intelligence. *Open Review*, 2022. 1.3.1

[171] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998. 3.1, 3.5

[172] Yann LeCun, Fu Jie Huang, and Leon Bottou. Learning methods for generic object recognition with invariance to pose and lighting. In *Conference on Computer Vision and Pattern Recognition*, volume 2, pages 97–104, 2004. 3.1

[173] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553): 436–444, 2015. 1, 1

[174] Gyewon Lee, Irene Lee, Hyeonmin Ha, Kyunggeun Lee, Hwarim Hyun, Ahnjae Shin, and Byung-Gon Chun. Refurbish your training data: Reusing partially augmented samples for

faster deep neural network training. In *USENIX Annual Technical Conference*, 2021. 3.6, 5.3

[175] Hongshan Li, Yu Guo, Zhi Wang, Shutao Xia, and Wenwu Zhu. Adacompress: Adaptive compression for online computer vision services. In *ACM International Conference on Multimedia*, pages 2440–2448, 2019. 4.6

[176] Jing Li, Hung-Wei Tseng, Chunbin Lin, Yannis Papakonstantinou, and Steven Swanson. HippogriffDB: Balancing I/O and GPU bandwidth in big data analytics. In *Proceedings of Very Large Databases*, volume 9, 2016. 4.1, 4.2, 4.6

[177] Percy Liang, Rishi Bommasani, Tony Lee, Dimitris Tsipras, Dilara Soylu, Michihiro Yasunaga, Yian Zhang, Deepak Narayanan, Yuhuai Wu, Ananya Kumar, Benjamin Newman, Binhang Yuan, Bobby Yan, Ce Zhang, Christian Cosgrove, Christopher D. Manning, Christopher Ré, Diana Acosta-Navas, Drew A. Hudson, Eric Zelikman, Esin Durmus, Faisal Ladhak, Frieda Rong, Hongyu Ren, Huaxiu Yao, Jue Wang, Keshav Santhanam, Laurel Orr, Lucia Zheng, Mert Yuksekgonul, Mirac Suzgun, Nathan Kim, Neel Guha, Niladri Chatterji, Omar Khattab, Peter Henderson, Qian Huang, Ryan Chi, Sang Michael Xie, Shibani Santurkar, Surya Ganguli, Tatsunori Hashimoto, Thomas Icard, Tianyi Zhang, Vishrav Chaudhary, William Wang, Xuechen Li, Yifan Mai, Yuhui Zhang, and Yuta Koreeda. Holistic evaluation of language models. *arXiv preprint arXiv:2211.09110*, 2022. 6.3.1

[178] Edo Liberty. Simple and deterministic matrix sketching. In *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 581–588, 2013. 4.6

[179] Hyeontaek Lim, David G. Andersen, and Michael Kaminsky. 3LC: Lightweight and effective traffic compression for distributed machine learning. In *Machine Learning and Systems*, 2019. 4.1, 4.1, 4.2, 4.6

[180] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. Microsoft COCO: Common objects in context. In *European Conference on Computer Vision*, 2014. 4.2, 5.2

[181] Yujun Lin, Song Han, Huizi Mao, Yu Wang, and William J. Dally. Deep gradient compression: Reducing the communication bandwidth for distributed training. In *International Conference on Learning Representations*, 2018. 4.1, 4.6

[182] John D. C. Little. A proof for the queuing formula: $L = \lambda W$. *Operations Research*, 1961. 4.4.1

[183] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C Berg. SSD: Single shot multibox detector. In *European Conference on Computer Vision*, 2016. 5.2

[184] Xiao Liu, Yanan Zheng, Zhengxiao Du, Ming Ding, Yujie Qian, Zhilin Yang, and Jie Tang. Gpt understands, too. *arXiv preprint arXiv:2103.10385*, 2021. 6.3.2

[185] Zihao Liu, Tao Liu, Wujie Wen, Lei Jiang, Jie Xu, Yanzhi Wang, and Gang Quan. Deepn-jpeg: A deep neural network favorable jpeg-based image compression framework. In *Annual Design Automation Conference*, 2018. 4.1, 4.6, 4.6

[186] Zihao Liu, Qi Liu, Tao Liu, Nuo Xu, Xue Lin, Yanzhi Wang, and Wujie Wen. Feature distillation: Dnn-oriented jpeg compression against adversarial examples. In *Conference on Computer Vision and Pattern Recognition*, 2019. 4.6

[187] Raymond A. Lorie. XRM - an extended (n-ary) relational memory. *IBM Research Report*, 1974. 2.1.1

[188] Xinghua Lu, Bin Zheng, Atulya Velivelli, and ChengXiang Zhai. Enhancing text categorization with semantic-enriched representation and training data augmentation. *Journal of the American Medical Informatics Association*, 13(5):526–535, 2006. 3.2

[189] Ningning Ma, Xiangyu Zhang, Hai-Tao Zheng, and Jian Sun. Shufflenet v2: Practical guidelines for efficient CNN architecture design. In *European Conference on Computer Vision*, 2018. 4.5.1

[190] Ping Ma, Michael W. Mahoney, and Bin Yu. A statistical perspective on algorithmic leveraging. *Journal of Machine Learning Research*, 16:861–911, 2015. 3.2

[191] Shin Matsushima, S.V.N. Vishwanathan, and Alexander J. Smola. Linear support vector machines via dual cached loops. In *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2012. 4.6

[192] Peter Mattson, Christine Cheng, Gregory Diamos, Cody Coleman, Paulius Micikevicius, David Patterson, Hanlin Tang, Gu-Yeon Wei, Peter Bailis, Victor Bittorf, David Brooks, Dehao Chen, Debo Dutta, Udit Gupta, Kim Hazelwood, Andy Hock, Xinyuan Huang, Daniel Kang, David Kanter, Naveen Kumar, Jeffery Liao, Deepak Narayanan, Tayo Oguntebi, Gennady Pekhimenko, Lillian Pentecost, Vijay Janapa Reddi, Taylor Robie, Tom St John, Carole-Jean Wu, Lingjie Xu, Cliff Young, and Matei Zaharia. MLPerf training benchmark. In *Machine Learning and Systems*, 2020. 5.2, 7.3

[193] Mark Mazumder, Colby R. Banbury, Xiaozhe Yao, Bojan Karlavs, William Gaviria Rojas, Sudnya Diamos, Gregory Frederick Diamos, Lynn He, Douwe Kiela, David Jurado, David Kanter, Rafael Mosquera, Juan Ciro, Lora Aroyo, Bilge Acun, Sabri Eyuboglu, Amirata Ghorbani, Emmett D. Goodman, Tariq Kane, Christine R. Kirkpatrick, Tzu-Sheng Kuo, Jonas Mueller, Tristan Thrush, Joaquin Vanschoren, Margaret J. Warren, Adina Williams, Serena Yeung, Newsha Ardalani, Praveen K. Paritosh, Ce Zhang, James Y. Zou, Carole-Jean Wu, Cody Coleman, Andrew Y. Ng, Peter Mattson, and Vijay Janapa Reddi. Dataperf: Benchmarks for data-centric ai development. *arXiv preprint arXiv:2207.10062*, 2022. 1.3.1

[194] Jim McDonnell. Large directory causes ls to hang. `http://unixetc.co.uk/2012/05/20/large-directory-causes-ls-to-hang/`, 2012. Accessed: 07-26-2021. 4.2

[195] H. Brendan McMahan, Gary Holt, D. Sculley, Michael Young, Dietmar Ebner, Julian Grady, Lan Nie, Todd Phillips, Eugene Davydov, Daniel Golovin, Sharat Chikkerur, Dan Liu, Martin Wattenberg, Arnar Mar Hrafnkelsson, Tom Boulos, and Jeremy Kubica. Ad click prediction: a view from the trenches. In *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2013. 1

[196] Brian McWilliams, Gabriel Krummenacher, Mario Lucic, and Joachim M Buhmann. Fast and robust least squares estimation in corrupted linear models. In *Advances in Neural*

Information Processing Systems, 2014. 3.2

[197] Qi Meng, Wei Chen, Yue Wang, Zhi-Ming Ma, and Tie-Yan Liu. Convergence analysis of distributed stochastic gradient descent with shuffling. In *Advances in Neural Information Processing Systems*, 2017. 4.1, 4.6

[198] Fabian Mentzer, George D Toderici, Michael Tschannen, and Eirikur Agustsson. High-fidelity generative image compression. *Advances in Neural Information Processing Systems*, 2020. 4.6

[199] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. Mixed precision training. In *International Conference on Learning Representations*, 2018. 1.1.3, 4.5.1

[200] Stoyan Mihov and Klaus U. Schulz. *Finite-State Techniques: Automata, Transducers and Bimachines*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2019. doi: 10.1017/9781108756945. 6.4.2, .2

[201] Tom Mitchell. *Machine learning*. 2007. 1

[202] MLPerfHPCv0.7. Mlperfhpcv0.7 results. `https://mlcommons.org/en/news/mlperf-hpc-v07`, 2020. Accessed: 07-26-2021. 4.1

[203] MLPerfv0.7. MLPerf training v0.7. `https://mlcommons.org/en/training-normal-07/`, 2020. 2.2.2, 4.1

[204] MLPerfv2.0. MLPerf training v2.0. `https://mlcommons.org/en/training-normal-20/`, 2022. 1.3.1

[205] Jack Moffitt. Ogg vorbis—open, free audio—set your media free. *Linux Journal*, 2001. 4.2, 4.3

[206] Jayashree Mohan, Amar Phanishayee, Ashish Raniwala, and Vijay Chidambaram. Analyzing and mitigating data stalls in DNN training. In *Proceedings of Very Large Databases*, volume 14, 2021. 1.3.1, 2.1.2, 2.3, 4.1, 4.2, 4.4.1, 4.6, 5.3

[207] Mehryar Mohri. Finite-state transducers in language and speech processing. *Computational Linguistics*, 23(2):269–311, 1997. 6.3.3

[208] Aaron Mok. ChatGPT could cost over $700,000 per day to operate. Microsoft is reportedly trying to make it cheaper. 2023. 2

[209] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics Magazine*, 38(8), 1965. 1, 7.3

[210] John Morris, Eli Lifland, Jin Yong Yoo, Jake Grigsby, Di Jin, and Yanjun Qi. TextAttack: A framework for adversarial attacks, data augmentation, and adversarial training in NLP. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 119–126, Online, October 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.emnlp-demos.16. 6.6

[211] Derek G. Murray, Jiří Šimša, Ana Klimovic, and Ihor Indyk. tf.data: A machine learning data processing framework. In *Proceedings of Very Large Databases*, volume 14, 2021.

2.1.1, 2.1.2, 2.2.2, 2.3, 4.1, 4.4, 4.5.1, 4.6, 5.2

[212] MXNET. Designing Efficient Data Loaders for Deep Learning. `https://mxnet.apache.org/api/architecture/note_data_loading`, 2018. 2.1.1

[213] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G. Azzolini, Dmytro Dzhulgakov, Andrey Mallevich, Ilia Cherniavskii, Yinghai Lu, Raghuraman Krishnamoorthi, Ansha Yu, Volodymyr Kondratenko, Stephanie Pereira, Xianjie Chen, Wenlin Chen, Vijay Rao, Bill Jia, Liang Xiong, and Misha Smelyanskiy. Deep learning recommendation model for personalization and recommendation systems. *arXiv preprint arXiv:1906.00091*, 2019. 1

[214] NVIDIA. DALI. `https://github.com/NVIDIA/DALI`, 2018. Accessed: 07-26-2021. 4.5.1, 7.2.1

[215] NVIDIA. NVIDIA A100 tensor core GPU architecture, 2020. 1.1.2

[216] U.S. Bureau of Labor Statistics. Software developers, quality assurance analysts, and testers. `https://www.bls.gov/ooh/computer-and-information-technology/software-developers.htm`, 2023. 5

[217] OpenAI. How much does gpt-4 cost? 2023. 2

[218] Nedjma Ousidhoum, Xinran Zhao, Tianqing Fang, Yangqiu Song, and Dit-Yan Yeung. Probing toxic content in large pre-trained language models. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 4262–4274, Online, August 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.acl-long.329. 6.2

[219] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. Making sense of performance in data analytics frameworks. In *USENIX Symposium on Networked Systems Design and Implementation*, 2015. 5.3, 7.2.2

[220] Kay Ousterhout, Christopher Canel, Sylvia Ratnasamy, and Scott Shenker. Monotasks: Architecting for performance clarity in data analytics frameworks. In *USENIX Symposium on Operating Systems Principles*, 2017. 5.3, 7.2.2, 7.2.2

[221] Denis Paperno, Germán Kruszewski, Angeliki Lazaridou, Ngoc Quan Pham, Raffaella Bernardi, Sandro Pezzelle, Marco Baroni, Gemma Boleda, and Raquel Fernández. The LAMBADA dataset: Word prediction requiring a broad discourse context. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1525–1534, Berlin, Germany, August 2016. Association for Computational Linguistics. doi: 10.18653/v1/P16-1144. 6.5.4

[222] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. 2019. 1, 1.1.2, 4.5.1, 6.5

[223] Gennady Pekhimenko, Vivek Seshadri, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, and Todd C. Mowry. Base-delta-immediate compression: practical data com-

pression for on-chip caches. In *International Conference on Parallel Architectures and Compilation Techniques*, 2012. 4.6

[224] Gennady Pekhimenko, Chuanxiong Guo, Myeongjae Jeon, Peng Huang, and Lidong Zhou. Tersecades: Efficient data compression in stream processing. In *USENIX Annual Technical Conference*, pages 307–320, 2018. 4.6

[225] Xingchao Peng, Judy Hoffman, X. Yu Stella, and Kate Saenko. Fine-to-coarse knowledge transfer for low-res image classification. In *IEEE International Conference on Image Processing*, pages 3683–3687, 2016. 4.6

[226] Fernando C. N. Pereira and Michael D. Riley. Speech recognition by composition of weighted finite automata. In *Finite-State Language Processing*, pages 431–453, 1996. 6.3.3

[227] Raymond Perrault, Yoav Shoham, Erik Brynjolfsson, Jack Clark, John Etchemendy, Barbara Grosz, Terah Lyons, James Manyika, Juan Carlos Niebles, and Saurabh Mishra. The AI Index 2019 Annual Report. Technical report, Stanford University, 2019. 4.1

[228] Steven Piantadosi. Modern language models refute chomsky's approach to language. Technical report, 2023. Retrieved from lingbuzz.net (ID: lingbuzz/007180). 1.2.1

[229] Neoklis Polyzotis and Matei Zaharia. What can Data-Centric AI learn from data and ML engineering? *arXiv preprint arXiv:2112.06439*, 2021. 1.3

[230] Shrimai Prabhumoye, Alan W Black, and Ruslan Salakhutdinov. Exploring controllable text generation techniques. In *Proceedings of the 28th International Conference on Computational Linguistics*, pages 1–14, Barcelona, Spain (Online), December 2020. International Committee on Computational Linguistics. doi: 10.18653/v1/2020.coling-main.1. 6.6

[231] John M. Prager. Open-domain question-answering. *Found. Trends Inf. Retr.*, 1:91–231, 2006. 6.2

[232] Daryl Pregibon. Logistic regression diagnostics. *The Annals of Statistics*, 9(4):705–724, 1981. 3.2

[233] Sarunya Pumma, Min Si, Wu-Chun Feng, and Pavan Balaji. Scalable deep learning via I/O analysis and optimization. *Transactions on Parallel Computing*, 2019. 4.6

[234] PytorchWebDataset. [rfc] add tar-based iterabledataset implementation to pytorch. `https://github.com/pytorch/pytorch/issues/38419`, 2020. Accessed: 07-26-2021. 4.2, 4.6

[235] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving language understanding by generative pre-training. 2018. 6.3.1

[236] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. 2019. 1.2, 6.3.1, 6.3.4, 6.4, 6.5, 6.5.4, 6.5.4, .2

[237] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, and Ilya Sutskever. Learning transferable visual models from natural language supervision. In *International Conference on Machine Learning*, 2021. 1.2.1

[238] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, 21(140): 1–67, 2020. 6.3.1, 6.3.2

[239] Alexander Ratner, Dan Alistarh, Gustavo Alonso, David G. Andersen, Peter Bailis, Sarah Bird, Nicholas Carlini, Bryan Catanzaro, Jennifer Chayes, Eric Chung, Bill Dally, Jeff Dean, Inderjit S. Dhillon, Alexandros Dimakis, Pradeep Dubey, Charles Elkan, Grigori Fursin, Gregory R. Ganger, Lise Getoor, Phillip B. Gibbons, Garth A. Gibson, Joseph E. Gonzalez, Justin Gottschlich, Song Han, Kim Hazelwood, Furong Huang, Martin Jaggi, Kevin Jamieson, Michael I. Jordan, Gauri Joshi, Rania Khalaf, Jason Knight, Jakub Konečný, Tim Kraska, Arun Kumar, Anastasios Kyrillidis, Aparna Lakshmiratan, Jing Li, Samuel Madden, H. Brendan McMahan, Erik Meijer, Ioannis Mitliagkas, Rajat Monga, Derek Murray, Kunle Olukotun, Dimitris Papailiopoulos, Gennady Pekhimenko, Theodoros Rekatsinas, Afshin Rostamizadeh, Christopher Ré, Christopher De Sa, Hanie Sedghi, Siddhartha Sen, Virginia Smith, Alex Smola, Dawn Song, Evan Sparks, Ion Stoica, Vivienne Sze, Madeleine Udell, Joaquin Vanschoren, Shivaram Venkataraman, Rashmi Vinayak, Markus Weimer, Andrew Gordon Wilson, Eric Xing, Matei Zaharia, Ce Zhang, and Ameet Talwalkar. Mlsys: The new frontier of machine learning systems. *arXiv preprint arXiv:1904.03257*, 2019. 1

[240] Alexander J Ratner, Henry Ehrenberg, Zeshan Hussain, Jared Dunnmon, and Christopher Ré. Learning to compose domain-specific transformations for data augmentation. In *Neural Information Processing Systems*, 2017. 3.2, 7.3

[241] RecordIODataset. Create a dataset using RecordIO. `https://mxnet.apache.org/api/faq/recordio` and `https://gluon-cv.mxnet.io/build/examples_datasets/recordio.html`. Accessed: 07-26-2021. 4.2

[242] Vijay Janapa Reddi, Christine Cheng, David Kanter, Peter Mattson, Guenther Schmuelling, Carole-Jean Wu, Brian Anderson, Maximilien Breughe, Mark Charlebois, William Chou, et al. Mlperf inference benchmark. In *ACM/IEEE International Symposium on Computer Architecture*, pages 446–459. IEEE, 2020. 4.1, 7.3

[243] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster R-CNN: Towards real-time object detection with region proposal networks. *Transactions on Pattern Analysis and Machine Intelligence*, 2016. 5.2

[244] Laria Reynolds and Kyle McDonell. Prompt programming for large language models: Beyond the few-shot paradigm. In *Extended Abstracts of the 2021 CHI Conference on Human Factors in Computing Systems*, CHI EA '21, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450380959. doi: 10.1145/3411763.3451760. 6.3.2

[245] Marco Tulio Ribeiro, Tongshuang Wu, Carlos Guestrin, and Sameer Singh. Beyond accuracy: Behavioral testing of NLP models with CheckList. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 4902–4912, Online, July 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.acl-main.442. URL `https://aclanthology.org/2020.acl-main.442`. 6.6

[246] Brian Roark, Richard Sproat, Cyril Allauzen, Michael Riley, Jeffrey Sorensen, and Terry Tai. The OpenGrm open-source finite-state grammar software libraries. In *Proceedings of the ACL 2012 System Demonstrations*, pages 61–66, Jeju Island, Korea, July 2012. Association for Computational Linguistics. 6.6

[247] Adam Roberts, Colin Raffel, and Noam Shazeer. How much knowledge can you pack into the parameters of a language model? In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 5418–5426, Online, November 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.emnlp-main.437. URL `https://aclanthology.org/2020.emnlp-main.437`. 6.2

[248] Kamil Rocki, Dirk Van Essendelft, Ilya Sharapov, Robert Schreiber, Michael Morrison, Vladimir Kibardin, Andrey Portnoy, Jean Francois Dietiker, Madhava Syamlal, and Michael James. Fast stencil-code computation on a wafer-scale processor. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2020. 1.1.2

[249] Tim Rocktäschel, Edward Grefenstette, Karl Moritz Hermann, Tomáš Kočiský, and Phil Blunsom. Reasoning about entailment with neural attention. In *International Conference on Learning Representations*, 2016. 6.3.1

[250] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. High-resolution image synthesis with latent diffusion models. *arXiv preprint arXiv:2112.10752*, 2021. 1.2.1

[251] Jessica Rumbelow and mwatkins. Solidgoldmagikarp (plus, prompt generation). `https://www.lesswrong.com/posts/aPeJE8bSo6rAFoLqg/solidgoldmagikarp-plus-prompt-generation`. Accessed: 05-05-2023. 7.2.5

[252] 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*, 115(3):211–252, 2015. 4.2, 4.5.1

[253] Mehdi Sajjadi, Mehran Javanmardi, and Tolga Tasdizen. Regularization with stochastic transformations and perturbations for deep semi-supervised learning. In *Neural Information Processing Systems*, 2016. 3.1, 3.2

[254] Timo Schick and Hinrich Schütze. Exploiting cloze-questions for few-shot text classification and natural language inference. In *Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics: Main Volume*, pages 255–269, Online, April 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.eacl-main.20. 6.3.2

[255] Dale Schuurmans. Memory augmented large language models are computationally universal. *arXiv preprint arXiv:2301.04589*, 2023. 6.3.4

[256] Heiko Schwarz, Detlev Marpe, and Thomas Wiegand. Overview of the scalable video coding extension of the h. 264/avc standard. *IEEE Transactions on circuits and systems for video technology*, 2007. 4.2, 4.3

[257] Abu Sebastian, Manuel Le Gallo, Riduan Khaddam-Aljameh, and Evangelos Eleftheriou. Memory devices and applications for in-memory computing. *Nature Nanotechnology*, 2020. 4.1

[258] Second Conference on Machine Translation. WMT. `http://www.statmt.org/wmt17/`, 2017. 5.2

[259] Emily Sheng, Kai-Wei Chang, Premkumar Natarajan, and Nanyun Peng. The woman worked as a babysitter: On biases in language generation. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 3407–3412, Hong Kong, China, November 2019. Association for Computational Linguistics. doi: 10.18653/v1/D19-1339. URL `https://aclanthology.org/D19-1339`. 6.5.2

[260] John E. Shore. The lazy repairman and other models: Performance collapse due to overhead in simple, single-server queuing systems. *SIGMETRICS Performance Evaluation Review*, 1980. 2.1.2

[261] David Silver, Aja Huang, Christopher J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–503, 2016. URL `http://www.nature.com/nature/journal/v529/n7587/full/nature16961.html`. 1

[262] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *International Conference on Learning Representations*, 2015. 4.4.1

[263] Athanassios Skodras, Charilaos Christopoulos, and Touradj Ebrahimi. The JPEG 2000 still image compression standard. *IEEE Signal Processing Magazine*, 2001. 4.3

[264] Charles Spearman. The proof and measurement of association between two things. *The American Journal of Psychology*, 15(1):72–101, 1904. 3.4.1

[265] Aarohi Srivastava, Abhinav Rastogi, Abhishek Rao, Abu Awal Md Shoeb, Abubakar Abid, Adam Fisch, Adam R Brown, Adam Santoro, Aditya Gupta, Adrià Garriga-Alonso, et al. Beyond the imitation game: Quantifying and extrapolating the capabilities of language models. *arXiv preprint arXiv:2206.04615*, 2022. 6.2, 6.2, 6.3.1, 6.3.1

[266] Stoyan Stefanov. *Book of Speed*. 2021. 4.3

[267] Saku Sugawara, Pontus Stenetorp, Kentaro Inui, and Akiko Aizawa. Assessing the benchmarking capacity of machine reading comprehension datasets. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 8918–8927, 2020. 6.3.2

[268] Rich Sutton. The bitter lesson, 2019. 1, 1.2

[269] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Conference on Computer Vision and Pattern Recognition*, 2015. 4.5.1

[270] Wilson L. Taylor. "cloze procedure": A new tool for measuring readability. *Journalism &*

Mass Communication Quarterly, 30:415 – 433, 1953. 6.3.2

[271] TensorFlow. Optimize TensorFlow performance using the Profiler. `https://www.tensorflow.org/guide/profiler`, 2020. 2.1.2

[272] TensorFlow. tf.data: Build TensorFlow input pipelines. `https://www.tensorflow.org/guide/data`, 2020. 2.1.1, 4.4, 4.5.1

[273] Tensorflow. Analyze tf.data performance with the TF Profiler. `https://www.tensorflow.org/guide/data_performance_analysis`, 2020. 2.1.2

[274] TensorFlow. tf.data bottleneck analysis. `https://www.tensorflow.org/guide/profiler#tfdata_bottleneck_analysis`, 2021. 2.1.2

[275] Tensorflow. XLA. `https://www.tensorflow.org/xla`, 2021. 1

[276] TFRecords. Tfrecord and tf.train.example. `https://www.tensorflow.org/tutorials/load_data/tfrecord`, 2021. Accessed: 07-26-2021. 4.2

[277] Daniel Ting and Eric Brochu. Optimal subsampling with influence functions. In *Advances in Neural Information Processing Systems*, 2018. 3.2

[278] George Toderici, Damien Vincent, Nick Johnston, Sung Jin Hwang, David Minnen, Joel Shor, and Michele Covell. Full resolution image compression with recurrent neural networks. In *Conference on Computer Vision and Pattern Recognition*, 2017. 4.3, 4.6

[279] Robert Torfason, Fabian Mentzer, Eirikur Agustsson, Michael Tschannen, Radu Timofte, and Luc Van Gool. Towards image understanding from deep compression without decoding. *arXiv preprint arXiv:1803.06131*, 2018. 4.6

[280] Hugo Touvron, Andrea Vedaldi, Matthijs Douze, and Herve Jegou. Fixing the train-test resolution discrepancy. In *Advances in Neural Information Processing Systems*. 2019. 4.1, 4.5.1, 4.6, 7.3

[281] Philipp Tschandl, Cliff Rosendahl, and Harald Kittler. The HAM10000 dataset, a large collection of multi-source dermatoscopic images of common pigmented skin lesions. *Scientific data*, 2018. 4.5.1

[282] S. Uhlich, M. Porcu, F. Giron, M. Enenkl, T. Kemp, N. Takahashi, and Y. Mitsufuji. Improving music source separation based on deep neural networks through data augmentation and network blending. In *International Conference on Acoustics, Speech and Signal Processing*, 2017. 3.2

[283] Matej Ulicny and Rozenn Dahyot. On using CNN with DCT based image data. In *Irish Machine Vision and Image Processing Conference*, 2017. 4.6

[284] Igor Vasiljevic, Ayan Chakrabarti, and Gregory Shakhnarovich. Examining the impact of blur on recognition by convolutional networks. *arXiv preprint arXiv:1611.05760*, 2016. 4.6

[285] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, 2017. 1, 5.2, 6.3.1

[286] Oriol Vinyals, Igor Babuschkin, Wojciech M. Czarnecki, Michaël Mathieu, Andrew Dudzik,

Junyoung Chung, David H. Choi, Richard Powell, Timo Ewalds, Petko Georgiev, Junhyuk Oh, Dan Horgan, Manuel Kroiss, Ivo Danihelka, Aja Huang, L. Sifre, Trevor Cai, John P. Agapiou, Max Jaderberg, Alexander Sasha Vezhnevets, Rémi Leblond, Tobias Pohlen, Valentin Dalibard, David Budden, Yury Sulsky, James Molloy, Tom Le Paine, Caglar Gulcehre, Ziyun Wang, Tobias Pfaff, Yuhuai Wu, Roman Ring, Dani Yogatama, Dario Wünsch, Katrina McKinney, Oliver Smith, Tom Schaul, Timothy P. Lillicrap, Koray Kavukcuoglu, Demis Hassabis, Chris Apps, and David Silver. Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature*, pages 1–5, 2019. 1

[287] Riccardo Volpi, Hongseok Namkoong, Ozan Sener, John C Duchi, Vittorio Murino, and Silvio Savarese. Generalizing to unseen domains via adversarial data augmentation. In *Advances in Neural Information Processing Systems*, 2018. 3.2

[288] Esteban Walker and Jeffrey B. Birch. Influence measures in ridge regression. *Technometrics*, 30(2):221–227, 1988. 3.2

[289] Eric Wallace, Shi Feng, Nikhil Kandpal, Matt Gardner, and Sameer Singh. Universal adversarial triggers for attacking and analyzing NLP. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 2153–2162, Hong Kong, China, November 2019. Association for Computational Linguistics. doi: 10.18653/v1/D19-1221. 6.6

[290] Gregory K. Wallace. The JPEG still picture compression standard. *Transactions on Consumer Electronics*, 1992. 4.2, 4.2, 4.6

[291] Alex Wang, Yada Pruksachatkun, Nikita Nangia, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel Bowman. Superglue: A stickier benchmark for general-purpose language understanding systems. In *Advances in Neural Information Processing Systems*, 2019. 6.3.1

[292] HaiYing Wang, Rong Zhu, and Ping Ma. Optimal subsampling for large sample logistic regression. *Journal of the American Statistical Association*, 113(522):829–844, 2018. 3.2

[293] Haohan Wang, Xindi Wu, Zeyi Huang, and Eric P. Xing. High-frequency component helps explain the generalization of convolutional neural networks. In *Conference on Computer Vision and Pattern Recognition*, 2020. 4.6

[294] Yu Emma Wang, Gu-Yeon Wei, and David Brooks. A systematic methodology for analysis of deep learning hardware and software platforms. In *Machine Learning and Systems*, 2020. 4.1, 4.2

[295] Zeke Wang, Kaan Kara, Hantian Zhang, Gustavo Alonso, Onur Mutlu, and Ce Zhang. Accelerating generalized linear models with mlweaving: A one-size-fits-all system for any-precision learning. In *Proceedings of Very Large Databases*, volume 12, 2019. 4.4.3, 4.6

[296] Zhou Wang, Eero P. Simoncelli, and Alan C. Bovik. Multiscale structural similarity for image quality assessment. In *Conference on Signals, Systems & Computers, 2003*, volume 2, pages 1398–1402, 2003. 4.5.4

[297] Jianqiao Wangni, Jialei Wang, Ji Liu, and Tong Zhang. Gradient sparsification for communication-efficient distributed optimization. In *Advances in Neural Information Processing Systems*, volume 31, 2018. 4.1, 4.6

[298] Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, Ed H. Chi, Tatsunori Hashimoto, Oriol Vinyals, Percy Liang, Jeff Dean, and William Fedus. Emergent abilities of large language models. *Transactions on Machine Learning Research*, 2022. URL `https://openreview.net/forum?id=yzkSU5zdwD`. 1.2

[299] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D.E. Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 307–320, 2006. 4.5.1

[300] Brent Welch, Marc Unangst, Zainul Abbasi, Garth Gibson, Brian Mueller, Jason Small, Jim Zelenka, and Bin Zhou. Scalable performance of the panasas parallel file system. In *USENIX Conference on File and Storage Technologies*, pages 1–17, 2008. 4.2

[301] 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 *Advances in Neural Information Processing Systems*, 2017. 4.1, 4.6

[302] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 2009. 5.3

[303] Wired. OpenAI's CEO Says the Age of Giant AI Models is Already Over. 2023. 3

[304] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Remi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander Rush. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, Online, October 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.emnlp-demos.6. 6.4.1

[305] David P. Woodruff. Sketching as a tool for numerical linear algebra. *Foundations and Trends in Theoretical Computer Science*, 2014. 4.6

[306] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016. 5.2

[307] Wm. A. Wulf and Sally A. McKee. Hitting the memory wall: Implications of the obvious. *SIGARCH Computer Architecture News*, 1995. 4.1

[308] Doris Xin, Hui Miao, Aditya Parameswaran, and Neoklis Polyzotis. Production machine learning pipelines: Empirical analysis and optimization opportunities. In *International Conference on Management of Data*, pages 2639–2652, 2021. 2.3

[309] Yuhui Xu, Yongzhuang Wang, Aojun Zhou, Weiyao Lin, and Hongkai Xiong. Deep neural

network compression with single and multiple level quantization. In *AAAI Conference on Artificial Intelligence*, volume 32, 2018. 4.6

[310] Zhi-Qin John Xu, Yaoyu Zhang, and Yanyang Xiao. Training behavior of deep neural network in frequency domain. In *International Conference on Neural Information Processing*, 2019. 4.6

[311] Masafumi Yamazaki, Akihiko Kasagi, Akihiro Tabuchi, Takumi Honda, Masahiro Miwa, Naoto Fukumoto, Tsuguchika Tabaru, Atsushi Ike, and Kohta Nakashima. Yet another accelerated SGD: ResNet-50 training on ImageNet in 74.7 seconds. *arXiv preprint arXiv:1903.12650*, 2019. 4.6

[312] Eddie Yan, Kaiyuan Zhang, Xi Wang, Karin Strauss, and Luis Ceze. Customizing progressive JPEG for efficient image storage. In *Workshop on Hot Topics in Storage and File Systems*, 2017. 4.4.2, 4.6

[313] Matthew Yancey. Three ways to count walks in a digraph. *arXiv preprint arXiv:1610.01200*, 2016. 6.4.3

[314] Dong Yin, Raphael Gontijo Lopes, Jon Shlens, Ekin Dogus Cubuk, and Justin Gilmer. A fourier perspective on model robustness in computer vision. In *Advances in Neural Information Processing Systems*, 2019. 4.6

[315] Chris Ying, Sameer Kumar, Dehao Chen, Tao Wang, and Youlong Cheng. Image classification at supercomputer scale. *arXiv preprint arXiv:1811.06992*, 2018. 4.5.1, 4.5.2, 4.6

[316] Kenichi Yoshida, Fuminori Adachi, Takashi Washio, Hiroshi Motoda, Teruaki Homma, Akihiro Nakashima, Hiromitsu Fujikawa, and Katsuyuki Yamazaki. Density-based spam detector. In *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 486–493, 2004. 1

[317] Yang You, Zhao Zhang, Cho-Jui Hsieh, James Demmel, and Kurt Keutzer. ImageNet training in minutes. In *International Conference on Parallel Processing*, 2018. 4.6

[318] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *USENIX Symposium on Operating Systems Design and Implementation*, 2008. 5.3

[319] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *USENIX Symposium on Networked Systems Design and Implementation*, 2012. 5.3

[320] Cyril Zhang, Kunal Talwar, Naman Agarwal, Rohan Anil, and Tomer Koren. Stochastic optimization with laggard data pipelines. In *Advances in Neural Information Processing Systems*, 2020. 4.6

[321] Hao Zhang, Zeyu Zheng, Shizhen Xu, Wei Dai, Qirong Ho, Xiaodan Liang, Zhiting Hu, Jinliang Wei, Pengtao Xie, and Eric P. Xing. Poseidon: An efficient communication architecture for distributed deep learning on GPU clusters. In *USENIX Annual Technical*

Conference, pages 181–193, 2017. 4.1, 4.6

[322] Mark Zhao, Niket Agarwal, Aarti Basant, Bugra Gedik, Satadru Pan, Mustafa Ozdal, Rakesh Komuravelli, Jerry Pan, Tianshu Bao, Haowei Lu, et al. Understanding and co-designing the data ingestion pipeline for industry-scale recsys training. *arXiv preprint arXiv:2108.09373*, 2021. 1.3.1, 2.3, 4.6, 7.2.1, 7.3

[323] Stephan Zheng, Yang Song, Thomas Leung, and Ian Goodfellow. Improving the robustness of deep neural networks via stability training. In *Conference on Computer Vision and Pattern Recognition*, pages 4480–4488, 2016. 4.6

[324] Hongyu Zhu, Mohamed Akrout, Bojian Zheng, Andrew Pelegris, Amar Phanishayee, Bianca Schroeder, and Gennady Pekhimenko. TBD: Benchmarking and analyzing deep neural network training. *arXiv preprint arXiv:1803.06905*, 2018. 4.1

[325] Rong Zhu. Gradient-based sampling: An adaptive importance sampling for least-squares. In *Neural Information Processing Systems*, 2016. 3.2

[326] Daniel M Ziegler, Nisan Stiennon, Jeffrey Wu, Tom B Brown, Alec Radford, Dario Amodei, Paul Christiano, and Geoffrey Irving. Fine-tuning language models from human preferences. *arXiv preprint arXiv:1909.08593*, 2019. 7.3

[327] Marcin Zukowski, Sandor Heman, Niels Nes, and Peter Boncz. Super-scalar RAM-CPU cache compression. In *International Conference on Data Engineering*, 2006. 4.6

Appendices

Extended Algorithms and Examples

We provide an extended version of `Plumber`'s resource accounted rates in Section .1 For `ReLM`, we provide an example of ambiguous automaton construction in Section .2 and the `ReLM` API in Section .3.

.1 `Plumber`: Extended Resource Accounted Rates

The LP formulation in the main text hinges on our ability to predict two *resource accounted rates*: ① the rate R_i of a `Dataset` given its configuration (e.g., number of cores assigned to each operation) and how that rate would change when one operation is assigned more cores, and ② the number of bytes needed to cache a `Dataset` at a given operation, accounting for bytes added/removed by data transformations applied by prior operations.

The intuition behind the computation of ① is that the rate of each operation from the source (e.g., data read from disk into the pipeline) to the root (i.e., minibatches exiting the pipeline) differs. For example, map operations may filter or amplify their inputs. We therefore begin with the root of the `Dataset` tree and traverse the pipeline until the source, accounting for every operation's rate in the process. The intuition behind the computation of ② is that the total data that needs to be cached is known at the source (i.e., when it enters the pipeline) and needs to be recomputed after each operation is applied. We therefore begin with the source of the `Dataset` tree and traverse the pipeline until the root.

① Work Completion Rates. In input pipelines, each `Dataset` operation is potentially characterized by different units of work. Looking at Figure 2.2, we do not know how many bytes are in a minibatch or how `Map` operations turn bytes into training examples, but we can observe that 128 images are batched into a minibatch. We thus rely on *visit ratios* from Operational Analysis [72], i.e., normalization constants, to compute the equivalent of root units, i.e., minibatches per second per core, that are generated by the source. Each `Dataset` operation is characterized by its local rate, r_i, which is defined as the average items of work completed per second per core, and can be computed by tracking two items: the items arrived and the items completed. The visit ratio, V_i, then, for a given `Dataset` operation is the constant that converts the local rate, r_i, to the global rate, R_i, which expresses the average amount of work completed by the operation in minibatches per second per core, i.e., the unit of work of the `Dataset` root given one core. Starting with the pipeline's root visit ratio, V_0, which is equal to 1, `Plumber` computes each `Dataset` operation's visit ratio using $V_i = (C_i/C_{i-1}) \times (C_{i-1}/C_0) = r_i \times V_{i-1}$, where C_i is the average number of items of work completed at `Dataset` operation i.

While the pipeline is running, `Plumber` collects counters on items arrived and completed across all operations of a given `Dataset`, which is used to model CPU performance. This rests on two simplifying assumptions, which do not affect `Plumber`'s efficiency in practice (§5.2): CPU work per item is fixed, and there is linear scaling of operation performance. Extending this approach to measure disk bandwidth usage only requires measuring filesystem reads and dividing by the wallclock time of the process. Converting usage to utilization is a matter of dividing the bandwidth usage by the available bandwidth, which `Plumber` measures by profiling the training directory using `fio` [25].

② **Cache Amplification Rates.** In ML settings, the benefit of caching is heavily skewed toward caching the *entire* disk-resident data—there is no locality due to random sampling. The fastest caching solution is one that is closest to the root of the `Dataset`, while remaining in memory. We note that *random* `Datasets`, which call into randomized functions, do not qualify for caching. By virtue of being random, their effective size is infinite—often, one can continue generating unique results forever.

To determine memory requirements for caching, `Plumber` assumes that files are consumed to completion and tracks the size of all input files (i.e., bytes read until end of file) used during the pipeline's operation. Once a new file is recorded, the file is added into a system-wide map tracking filename to bytes used. The space requirements are moderate for realistic use-cases; only two integers are needed per filename with hashing, and the number of files is likely small due to large files (e.g., ImageNet has 1024 files).

To calculate how many bytes it takes to materialize each `Dataset`, we approximate two statistics: the number of elements (*cardinality*), n_i, and the average size of each element (*byte ratio*), b_i. For a data source, we have the `Dataset` size in bytes: $\sum_{f \in \mathbb{F}} S_f$, where S_f is the size of a file f in the set of recorded files, $\mathbb{F}$. As the source data flows through the input pipeline, n_i and b_i will change as training examples are grouped, filtered, and/or transformed (e.g., parsed and decoded in `Map`). To build intuition, let us revisit Figure 2.2. b_i for `Map` is just the "decompression ratio" from records to images, which is often approximated to be $10\times$ for JPEG. Likewise, the `Batch` makes b_i 128 times bigger in the process of grouping, while making n_i 128 times smaller. Therefore, we can conclude the root `Dataset` is an order of magnitude larger than it was at the source.

We first tackle how to solve n_i. For exposition, assume the common case that we only have one source (e.g., Figure 2.2) and the source is finite—we will be propagating our analysis up from this source until we hit the root. Readers emit elements at per-row or per-record granularity, thus, for source `Datasets`, we overload r_i from visit ratios to denote this ratio (though it is a ratio of records/byte). The initial dataset size at source i is thus $n_i = (\sum_{f \in \mathbb{F}} S_f) \times r_i$. For a `Dataset`, i, the subsequent `Dataset`, j, follows the recurrence $n_j = r_j \times n_i$, where r_j follows previously defined semantics from visit ratios. Intuitively, we convert the size in bytes to the size in *records* to the size in training examples, and so on. Special care must be taken to account for `Datasets` repeating examples or truncating the examples stream. Generalizing to multiple children (i.e., sources) requires some `Dataset`-specific aggregation (e.g., `sum`).

To obtain b_i, we can use entirely local information. `Plumber` tracks one local quantity, the bytes-per-element, $b_i =$ `bytes_produced(i)`$/C_i$, where the numerator is a counter for bytes produced at i, and C_i is the counter for number of completions from i. The accuracy of estimating the materialized size with $b_i \times n_i$ depends on the variance of each component's estimate.

To deal with large datasets, which may take longer than the `Plumber` tracing time to iterate through, we obtain a subsample, s, of the set of file sizes, S, which we can use to approximate S. If we have n of m samples, we can simply rescale the subsampled size, $\sum_n s$, by m/n to get an estimate of the full dataset size, $\frac{m}{n} \times \mathbb{E}[\sum_n s]$. Empirically, the estimate is sufficiently accurate: 1% of files gives relative error of 1% ImageNet and 2% for COCO, and 5 file gives less than 2% relative error for WMT datasets. This allows `Plumber` to get a tight estimate of the true dataset size in seconds with sufficient read parallelism. The normal distribution of error we observe matches intuition that the Central Limit Theorem applies.

.2 `ReLM`: Ambiguous Automaton Construction

We describe how to implement a transducer composition-like algorithm to construct the full (ambiguous) automaton. Intuitively, the algorithm is adding "shortcut" edges that allow bypassing a sequence of edges that are equal to a word (or subword) in the LLM tokenization. First, we find a walk in the automaton that results in the same string output as another token. Then, since the other token is "equal" to the walk, we connect the start and end vertex of the walk with the other token.

For example, if the walk in the automaton traverses `T–h–e`, this walk is equivalent with respect to output string to the token representing `The`. Since `T–h–e` is a valid walk in the automaton (i.e., yields a valid substring from the particular state in the automaton), and `The` is equivalent to the walk, we can add a "shortcut" edge connecting the starting and ending vertex of `T–h–e` with the edge value of `The`. One can view this procedure as an optional rewrite [200], where the sequence `T–h–e` is optionally rewritten to `The`.

We note that while the examples we provide are tailored toward the ASCII subset of Unicode, an implementation covering the full Unicode range requires care in handling Byte-Pair Encodings (BPE) [98, 236]. Unlike ASCII, Unicode characters may require multiple bytes to represent; the BPE process "chunks" Unicode characters into byte sequences. It is thus necessary to break up the characters into byte sequences via rewrites before the algorithm presented here is run and while (sub)words representing tokens are being matched against these byte sequences in the automaton.

We show the algorithm pseudo-code in Algorithm 1 and Algorithm 2. Algorithm 1 is an inner method of Algorithm 2. In Algorithm 1, `DFSMatch` is standard depth-first search (DFS) matching applied from the vertex and matching on edges corresponding to word. We assume that `DFSMatch` is implemented such that each edge (character) in the word is matched or not in $O(1)$ time e.g., if the edges are represented in a dense array or hashtable. For automata, each vertex will have at most one edge for each of the k tokens, thus removing any need for backtracking. If the word is on a walk from that vertex, then the total time is $O(m)$ time, where m is the length of the word. Over all vertices, this compounds to $O(Vm)$ time and returns $O(V)$ edges. In Algorithm 2, we loop k times, where k is the number of tokens/words in the LLM's tokenization scheme. Combining with the prior result, the runtime is $O(Vm_{\text{max}}k)$, where m_{max} is the size of the largest m.

.3 Extended `ReLM` API Example

Figure 2 provides an example of the full API that can be used to generate the George Washington birth date example from Figure 6.1. Compared to Figure 6.4, there are more parameters to configure, such as the search strategy (e.g., shortest path or random sampling) and the tokenization options (e.g., canonical or all). Additionally, the role of the tokenizer is now made explicit. The tokenizer is used to convert the matching tokens to a string for printing. Note that only the first match is shown.

Algorithm 1 Get Connecting Walks (DFS)

```
input: Automaton automaton
input: String word
all_matching_walks = []
for vertex in automaton.vertices() do
    matching_walk = DFSMatch(automaton, vertex, word)
    if matching_walk then
        all_matching_walks.append(matching_walk)
    end if
end for
return all_matching_walks
```

Algorithm 2 Add Ambiguous Edges Algorithm (DFS)

```
input: Automaton automaton
input: Dict[String,Int] word_token_map
for word in word_token_map.keys() do
    if len(word) $> 1$ then
        walks = GetConnectingWalks(automaton, word)
        token = word_token_map[word]
        for walk in walks do
            automaton.addEdge(walk.vertex_from, walk.vertex_to, token)
        end for
    end if
end for
```

```python
query_string = relm.QueryString(
    query_str=("George Washington was born on "
               "((January)|(February)|(March)|(April)|"
               "(May)|(June)|(July)|(August)|(September)|"
               "(October)|(November)|(December)) "
               "[0-9]{1,2}, [0-9]{4}"),
    prefix_str="George Washington was born on"
)
query = relm.SimpleSearchQuery(
    query_string=query_string,
    search_strategy=relm.QuerySearchStrategy.SHORTEST_PATH,
    tokenization_strategy=relm.QueryTokenizationStrategy.ALL_TOKENS,
    top_k_sampling=None,
    sequence_length=None)
ret = relm.search(model, tokenizer, query)
for x in ret: # Print resulting strings
    print(tokenizer.decode(x)) # George Washington was born on July 4, 1732
```

Figure 2: `Python` code for the George Washington birth date example in Figure 6.1. `model` and `tokenizer` are LLM-specific objects provided by external libraries. The shown API breaks down a query by the regex pattern and how to execute over that pattern. Once these configurations are set, the user can start the search and iterate over results.