Hindsight Logging for Model Training

Rolando Garcia
UC Berkeley
rogarcia@berkeley.edu

Eric Liu
UC Berkeley
rickchang@berkeley.edu

Vikram Sreekanti
UC Berkeley
vikrams@berkeley.edu

Bobby Yan
UC Berkeley
bobbyy@berkeley.edu

Anusha Dandamudi
UC Berkeley
adandamudi@berkeley.edu

Joseph E. Gonzalez
UC Berkeley
jegonzal@berkeley.edu

Joseph M. Hellerstein
UC Berkeley
hellerstein@berkeley.edu

Koushik Sen
UC Berkeley
ksen@berkeley.edu

ABSTRACT

In modern Machine Learning, model training is an iterative, experimental process that can consume enormous computation resources and developer time. To aid in that process, experienced model developers log and visualize program variables during training runs. Exhaustive logging of all variables is infeasible, so developers are left to choose between slowing down training via extensive *conservative* logging, or letting training run fast via minimalist *optimistic* logging that may omit key information. As a compromise, optimistic logging can be accompanied by program checkpoints; this allows developers to add log statements post-hoc, and "replay" desired log statements from checkpoint—a process we refer to as *hindsight* logging. Unfortunately, hindsight logging raises tricky problems in data management and software engineering. Done poorly, hindsight logging can waste resources and generate technical debt embodied in multiple variants of training code. In this paper, we present methodologies for efficient and effective logging practices for model training, with a focus on techniques for hindsight logging. Our goal is for experienced model developers to learn and adopt these practices. To make this easier, we provide an open-source suite of tools for Fast Low-Overhead Recovery (`flor`) that embodies our design across three tasks: (i) efficient background logging in Python, (ii) adaptive periodic checkpointing, and (iii) an instrumentation library that codifies hindsight logging for efficient and automatic record-replay of model-training. Model developers can use each `flor` tool separately as they see fit, or they can use `flor` in hands-free mode, entrusting it to instrument their code end-to-end for efficient record-replay. Our solutions leverage techniques from physiological transaction logs and recovery in database systems. Evaluations on modern ML benchmarks demonstrate that `flor` can produce fast checkpointing with small user-specifiable overheads (e.g. 7%), and still provide hindsight log replay times orders of magnitude faster than restarting training from scratch.

PVLDB Reference Format:
Rolando Garcia, Eric Liu, Vikram Sreekanti, Bobby Yan, Anusha Dandamudi, Joseph E. Gonzalez, Joseph M. Hellerstein, and Koushik Sen. Hindsight Logging for Model Training. PVLDB, 14(4): 682-693, 2021.
doi:10.14778/3436905.3436925

Proceedings of the VLDB Endowment, Vol. 14, No. 4 ISSN 2150-8097.
doi:10.14778/3436905.3436925

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/ucbrise/flor.

1 INTRODUCTION

Due to the growing scale and complexity of sophisticated models [13, 42, 54], exploratory model development increasingly poses data management problems [51]. At every step of exploration, model developers routinely track and visualize time series data to diagnose common training problems such as exploding/vanishing gradients [18], dead ReLUs [31], and reward hacking [5]. Model developers use state-of-the-art loggers specialized to machine learning (e.g. TensorBoard [15], and WandB [3]) to efficiently trace and visualize data as it changes over time. The following are common examples of times series data logged in model training:

- **Training Metrics**: The loss, accuracy, learning rate, and other metrics as they change over time.
- **Tensor Histograms**: Histograms of weights, gradients, activations, and other tensors as they change over time.
- **Images & Overlays**: Segmentation masks, bounding boxes, embeddings, and other transformed images as they change over time.

In our experience, all model developers log some training metrics by default. Whether their logging practice is conservative or optimistic depends on whether they log additional training data by default. Next, we illustrate the relevant differences between conservative and optimistic logging, with the aid of three fictitious characters: Mike for methodical conservative logging, Chuck for ad-hoc optimistic logging, and Judy for methodical optimistic logging.

1.1 Conservative Logging

Conservative logging is eager and characterized by stable expectations about what data (and how much of it) will be necessary for analysis [62]. It is especially well-suited to later stages of the machine learning lifecycle, where models are periodically trained

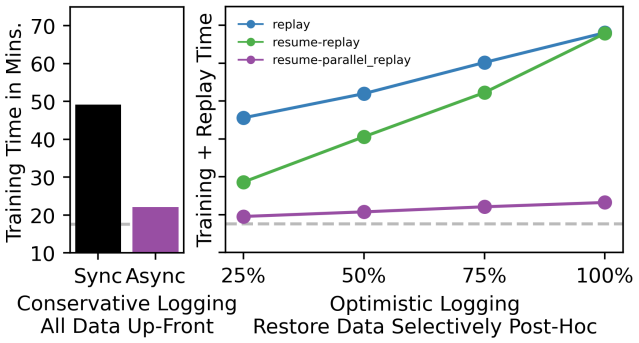

Figure 1: Conservative v. Optimistic logging performance at 100 epochs of Squeezenet on CIFAR-100 [22]. All workloads log tensor histograms for the activations, weights, and gradients 4× per epoch. The gray horizontal line corresponds to the same training job but without any logging. Both, the purple bar and line, correspond to `flor` logging.

for many hours on fresh data [53], and refinements of the model training pipeline are usually light and limited to some tuning [14].

1.1.1 ***Mike*** *records everything (mnemonic for microphone).* Mike is a model developer for a major tech company. His organization's policy is that model developers should log training metrics, tensor histograms, and some images and overlays by default. Although his logging practices can add substantial overhead to training (black bar in Figure 1), his jobs usually run as offline batches, and his productivity is not blocked on the results of training. Moreover, when he receives an alert from the training or monitoring system, the execution data he needs for post-hoc analysis will be ready.

Although reducing logging overhead is not a high priority for Mike, he is not the only developer in his organization using high-end GPU clusters. At scale, even minor improvements to training efficiency will translate into measurable benefits for the organization. Later in this paper, we will present a tool for **low-overhead materialization in the background (Section 3.1)**. Our tool enables Mike to continue to log data at the same rate and with his logger of choice (e.g. `tensorboardx`), at a fraction of the original overhead (purple bar in Figure 1).

1.2 Optimistic Logging

In contrast to conservative logging, optimistic logging is an agile and lazy practice especially well-suited to early and unstructured stages of exploratory model development. In optimistic logging, model developers log training metrics such as the loss and accuracy by default, and defer collection of additional data until analysis time, when they may restore it selectively. Execution data is restored by adding logging statements to training post-hoc, and replaying—possibly from checkpoint. We refer to this practice as *hindsight logging*. Optimistic logging consists of (i) logging some training metrics by default, and (ii) selectively restoring additional training data post-hoc with hindsight logging. Model developers gain agility in exploration from optimistic logging in three ways:

- **Deferred Overhead**: Each training batch executes and produces results as quickly as possible. Faster evaluations means more trials in the same time span.
- **Flexible Cost Schedule**: Model developers can selectively restore *just* the data they need post-hoc. The fewer epochs they need to probe; the fewer resources they burn.
- **Separation of Concerns**: Concerns about what data to log and how much of it do not burden the developer during design and tuning—these are postponed until analysis time.

In the fast path of the common case, model developers get all the relevant information from the training loss, and move on. In exceptional cases, however, training replay may be necessary for post-hoc data restoration. Compare how Chuck and Judy would restore data.

1.2.1 ***Chuck*** *doesn't record anything (mnemonic for toss).* Chuck is a first year graduate student in Astronomy who is knowledgeable about the latest developments in machine learning, but ill-versed in software engineering practices. Chuck logs the training loss and accuracy by default, but does not save checkpoints during training.

1.2.2 ***Judy*** *uses good judgment (mnemonic for judge).* Judy is an experienced model developer with a strong software engineering background. Like Chuck, she only logs the training loss and accuracy by default; unlike Chuck, she also checkpoints model training periodically, and manages the many versions of her code and data.

When either Chuck or Judy need to restore training data post-hoc—say, the tensor histograms for the gradients at a rate of 4× per epoch—they will selectively add the necessary logging statements to their training code, and re-train. In many cases, Chuck and Judy will only want to restore data for a small fraction of training (e.g. 25% of the epochs), near the region where the loss exhibits an anomaly (e.g. near the middle of training). Because Judy checkpointed her training execution, she is able to resume training at an arbitrary epoch. Chuck, on the other hand, must retrain from the start. In the right pane of Figure 1, we plot training plus replay times for Chuck (blue line) and Judy (green line).

In this paper, we will concretely define the methodology that enables Judy to achieve effective record-replay of model training (Section 2). Our goal is for experienced model developers to learn and adopt these best practices, for their numerous benefits. One surprising consequence of Judy's approach is that she can **parallelize replay of model training** with her periodic checkpoints (Section 2.2). The purple line in Figure 1 represents Judy's parallel replay. Additionally, we evaluate (Section 4) and open-source our Fast Low-Overhead Recovery suite (abbreviated as `flor`) for hindsight logging—with the following set of tools:

- An optimized materialization library for low-overhead logging and checkpointing in Python (Section 3.1).
- An adaptive periodic checkpointing mechanism, so record overhead never exceeds a specifiable limit (Section 3.2).
- An instrumentation library that can transparently transform Python training code to conform with the methodical hindsight logging approach (Section 3.3). This protects model developers from incurring technical debt as a consequence of lapses in discipline, and enables novices to restore time series data as efficiently as experts.

2 METHODICAL HINDSIGHT LOGGING

In hindsight logging, model developers can choose what to log long after model training: at analysis time and with a question in mind. In essence, we want to query past execution state, without versioning that state in full. We draw inspiration from the rich body of work in databases dedicated to fast recovery [37, 60, 65]. Although that work focuses mostly on transactions, the lessons and trade-offs transfer naturally to execution recovery for arbitrary programs. There are two means for recovering execution data: physically, by reading it from disk; and logically, by recomputing it. Both a purely physical approach and a purely logical approach are unattractive in our setting, due to prohibitive overhead on record and prohibitive latency on replay, respectively. Instead, hindsight logging—like transaction logging—embraces a hybrid "physiological" [16] approach that takes partial checkpoints on the first pass (henceforth the *record* phase), and uses those checkpoints to speedup redo (henceforth the *replay* phase). In this section, we give a high-level overview of the enabling methodology behind efficient hindsight logging:

(1) First and foremost, **checkpoint periodically** during training. At least once per epoch for partial replay, but much less frequently is sufficient for parallel replay.
(2) Additionally, **enclose long-running code inside a conditional statement to exploit memoization** speedups. On record, the execution materializes the side-effects of each memoized block. On replay, model developers will run their code from the start without modification, and the execution will intelligently skip the recomputation of some blocks by loading their side-effects from disk.
(3) Finally, include logic to **resume training from a checkpoint**. Replay of model training is embarrassingly parallel given periodic checkpoints. To parallelize replay of model training, a model developer dispatches multiple training jobs in parallel, each loading checkpoints to resume training from a different epoch and terminating early.

If at any point through our forthcoming discussion the programming burden seems too high, the reader should note that we also provide a tool that codifies and automatically applies these methods for the benefit of the user: an instrumentation library that features a hands-free mode for convenience and robustness (Section 3.3).

2.1 Periodic Checkpointing & Memoization

Many model developers already checkpoint training periodically. This is traditionally done for warm-starting training as well as for fault tolerance. In this section, we show how to exploit further benefits from periodic checkpointing, without incurring additional overheads. In Figure 2, we provide an example of how a model developer would materialize the model and optimizer state once per epoch (line 10). This state serves a dual purpose. First, it comprises the relevant side-effects of the preceding code block (lines 4-9), so it serves a memoization purpose (computation skipping). Second, it captures all of the state that is modified every epoch of training, so it comprises a valid and complete checkpoint. This dual purpose of selective state capture is a fortunate coincidence that arises naturally from the nested loops structure of model training.

```
1  init_globals()
2  checkpoint_resume(args, (net, optimizer))
3  for epoch in range(args.start, args.stop):
4    if skipblock.step_into(...):
5      for batch in training_data:
6        predictions = net(batch.X)
7        avg_loss = loss(predictions, batch.y)
8        avg_loss.backward()
9        optimizer.step()
10   skipblock.end(net, optimizer)
11   evaluate(net, test_data)
```

Figure 2: Training script prepared for methodical hindsight logging: checkpoint resume (line 2), block memoization (lines 4 - 10), and periodic checkpointing (line 10). The semantics of SkipBlock are covered in subsection 2.1.

We make use of the SkipBlock language construct [10], to denote block memoization. The first two requirements for efficient record-replay are periodic checkpointing and block memoization. Both are achievable by the following functionality, which is encapsulated by the SkipBlock for ease of exposition:

- **Parameterized Branching**: SkipBlock always applies the side-effects of the enclosed block to the program state, but does so in one of two ways: (a) by executing the enclosed block, or (b) by skipping the block and instead loading the memoized side-effects from its corresponding checkpoint. SkipBlock automatically determines whether to execute or skip the enclosed block. It is parameterized by relevant execution state: i.e. record execution, replay resume, replay execution, and whether the enclosed block is probed.
- **Side-Effect Memoization (i.e. Periodic Checkpointing)**: When the enclosed block is executed, SkipBlock materializes its side-effects (the arguments passed to the call in line 10, Figure 2). It is possible to optimize the SkipBlock for low-overhead background materialization (Section 3.1), and adaptive periodic materialization (Section 3.2), but these optimizations do not alter the semantics of SkipBlock.
- **Side-Effect Restoration**: Whenever the enclosed block is skipped, SkipBlock restores its side-effects from its corresponding checkpoint (line 10, Figure 2). SkipBlock is able to efficiently locate an execution's corresponding checkpoint on disk, and apply its side-effects to the program state.

A block may not be skipped on replay when the model developer adds a hindsight logging statement inside the block. Although SkipBlock memoizes the block's final state (i.e. state that is visible to subsequent program statements), it does not materialize intermediate state, such as the activations of the model (e.g. line 6 in Figure 2). Consequently, if the model developer wishes to restore the model activations post-hoc, it will not be possible to skip the nested training loop. To restore data in such cases, parallel replay is the only option for reducing the latency of hindsight logging.

```
1  init_globals()
2  for epoch in range(0, args.stop):
3    if skipblock.step_into(...
4        && epoch >= args.start):
5      for batch in training_data:
6        predictions = net(batch.X)
7        avg_loss = loss(predictions, batch.y)
8        avg_loss.backward()
9        optimizer.step()
10   skipblock.end(net, optimizer)
11   lr_scheduler.step()
12   evaluate(net, test_data)
```

Figure 3: Training script prepared for methodical hindsight logging. Training can resume from a partial checkpoint (no `lr_scheduler` in checkpoint).

2.2 Parallel Replay by Checkpoint Resume

As we saw in the previous subsection, our approach cannot avoid expensive recomputation when intermediate training state, such as the gradients or activations are logged post-hoc. In such cases, model developers will want to reduce replay latency by utilizing additional resources—specifically, more GPUs for parallelism. Although auto-parallelizing arbitrary sequential code remains an open challenge [7], the replay of checkpointed model training is a special case: training replay is embarrassingly parallel given periodic checkpoints. As we multi-purposed periodic checkpointing in the previous section for memoization, so too we now multi-purpose checkpoint resume—a current staple in the training code of many model developers—for parallel replay. Parallel replay enables us to substantially cut hindsight logging latency, and due to the prevalence of checkpoint resume, this is possible without incurring a programming burden. To parallelize replay, a model developer simultaneously resumes training from various checkpoints:

(1) First, they dispatch multiple training jobs in parallel.
(2) Then, each job loads the checkpoint (line 2 in Figure 2) that corresponds to the epoch it is resuming from. For example, to resume training at epoch 25, the job loads the checkpoint stored at the end of epoch 24.
(3) Finally, each job independently works on its share of work (see the `range` in line 3 of Figure 2).

2.2.1 ***Pseudoresuming*** *from partial checkpoints.* When model developers write code for periodic checkpointing themselves, they can ensure that the objects they capture constitute a complete checkpoint. However, when model developers entrust `flor` to instrument their code for automatic periodic checkpointing, `flor` will not be able to automatically determine whether the checkpoint is complete or partial with respect to training. As we will discuss in Section 3.3, `flor` can only estimate the side-effects of blocks of code enclosed by SkipBlocks: a restriction we use to render our static analysis tractable. `flor` will not estimate the side-effects of the program at arbitrary points, and it will not check whether the data materialized constitutes a complete (or partial) checkpoint with respect to training, since doing so statically (i.e. with low overhead) would be intractable in Python [19, 39, 50].

Consequently, `flor` assumes checkpoints materialized automatically are partial with respect to training. By partial, we mean that there are objects modified every epoch that are not stored by the checkpoints (e.g. the `lr_scheduler` in Figure 3). As a result, it is not possible to resume training from an arbitrary epoch merely by loading a partial checkpoint (i.e. a physical recovery approach). Instead, we start training from the beginning (line 2 in Figure 3), and use the partial checkpoints to skip recomputation of memoized blocks during the initialization — or *pseudoresume* — phase (lines 3-4 in Figure 3). This approach is characteristically physiological because it relies on a combination of recomputation and disk reads for recovery. Although *pseudoresume* is especially important for auto-parallelizing replay of model training, we share this method here because novice model developer may accidentally store partial checkpoints. This technique allows them to resume training efficiently all the same. For illustration, suppose, that the model developer wants to resume training from epoch 25, using the script in Figure 3. The SkipBlock would be initialized to a *pseudoresume* state, and then toggle to an execution state.

Pseudoresume phase for epochs in the range 0-24 (inclusive):

(1) Skip the nested training loop (lines 3-9 in Figure 3).
(2) Load the side-effects of the skipped block (line 10 in Figure 3).
(3) All other statements execute normally.

Execution phase from epoch 25 onward:

(1) Step into the nested training loop (lines 3-9 in Figure 3).
(2) All other statements execute normally.

In summary, memoization can resume model training from an arbitrary epoch, even in the absence of complete checkpoints. As we will show in the evaluation, the overhead of *pseudoresume* is amortized in parallel replay, so that the difference between checkpoint resume and *pseudoresume* is imperceivable to the end-user. This result is important because it enables us to efficiently auto-parallelize the replay of model training, even with partial checkpoints.

3 TOOLING FOR HINDSIGHT LOGGING

Model developers may adopt the methods described in Section 2 to achieve efficient hindsight logging. To facilitate this adoption, we provide a suite of *Fast Low-Overhead Recovery* tools—`flor` for short—as aid to the developer. Model developers may use each tool separately as they see fit, or they may use `flor` in hands-free mode, entrusting it to instrument their code end-to-end for efficient record-replay. `flor` provides the following tools:

- An optimized materialization library for low-overhead logging and checkpointing (Section 3.1).
- An adaptive periodic checkpointing mechanism, so record overhead never exceeds a specifiable limit (Section 3.2).
- An instrumentation library that can transparently transform training code to conform to the methodical hindsight logging approach (Section 3.3). This protects model developers from incurring technical debt as a consequence of lapses in discipline, and enables novices to restore time series data as efficiently as experts.

3.1 Background Logging

`flor` provides a background materialization mechanism optimized for PyTorch, which is compatible with model developer's machine learning logging service of choice (for example, TensorBoard [15], MLFlow [63], and WandB [3]). Background logging is used natively by SkipBlocks for low-overhead periodic checkpointing (Section 2.1). It is also available separately as a library for end-users.

Both logging and checkpointing can add measurable overhead to training because they require moving data from GPU memory to DRAM, serializing it into byte arrays, and then writing those arrays to disk. Of the latter two, serialization is typically much more expensive than I/O: by an average factor of 4.3× according to our microbenchmarks [28]. Consequently, after copying select data from GPU memory to DRAM (so it is protected from overwrites), we would like to take materialization (both serialization and I/O) off the main thread—which is dedicated to model training—and do it in the background. Despite its maturity and widespread popularity, Python makes this very difficult.

The Python interpreter has a notorious Global Interpreter Lock that prevents parallelism among Python threads. Unfortunately, the Python IPC schemes also require serialization by the sending process—returning us to our original problem. To avoid serialization we could use a solution like Apache Plasma, but it only avoids serialization for a subset of Python data types (notably dataframes and arrays) and actually cannot serialize other data types including Pytorch tensors. We eventually found a workaround at the operating system level, using `fork()` as a mechanism to achieve efficient one-shot, one-way IPC between a parent and child process, with copy-on-write concurrency. To materialize a record checkpoint, the main process forks and then immediately resumes model training; the child process serializes the checkpoint, writes it to disk, and then terminates. To prevent too many calls to `fork()`, we buffer up checkpoints and process them in batches of 5000 objects. Given the short lifespan of these child processes and an infrequent rate of forking due to batching, we have never seen more than two live children at any point in our evaluations—including in models that ran for many hours (Section 4).

In a technical report [28], we provide a more detailed discussion of the design and performance of our background materialization mechanism. This mechanism cuts logging overheads by 73.5% on average, according to our microbenchmarks [28]. Execution speedups due to background logging are modest for workloads whose logging overheads are dominated by periodic checkpointing ($\mu = 4.76\%$ overhead down to $\mu = 1.74\%$). This is because periodic checkpointing is already light and doesn't add much overhead to training. However, as we saw in Figure 1, background logging can have a drastic effect when used for conservative logging (180% overhead down to 26%), since logging overheads account for a much larger fraction of end-to-end training times in those cases.

3.2 Adaptive Periodic Checkpointing

In this section, we present a decision rule for dynamically calculating an appropriate checkpointing *period* or frequency. This condition is automatically tested by SkipBlocks to adapt the frequency of checkpointing to each training workload. For many developers, checkpointing once per epoch is a good default, but in general, the right checkpointing frequency depends on the training workload: e.g. how fast or slow the code executes relative to the size of its checkpoints. The goal of adaptive periodic checkpointing is to automatically materialize checkpoints as frequently as will increase expected replay speedups, subject to the constraint that record overhead does not exceed a user-specifiable limit. Next we derive the invariants we use for adaptive periodic checkpointing. We refer the reader to the notation in Table 1.

Table 1: Symbol table for Adaptive Periodic Checkpointing

Symbol	Description
M_i	time to materialize side-effects of block identified by i
R_i	time to restore side-effects of block identified by i
C_i	time to compute block identified by i
n_i	count of executions (so far) for block i
k_i	count of checkpoints (so far) for block i
G	degree of replay parallelism
c	constant scaling factor
ϵ	tunable parameter denoting overhead tolerance

3.2.1 ***The Record Overhead Invariant.*** We require that the materialization overhead of a block is at most a small fraction of its computation time: $M_i < \epsilon C_i$. This simplistic invariant is enough to ensure that record never exceeds a user-specifiable overhead (ϵ), but it is all-or-nothing: a block is memoized always or never. Since blocks are often nested inside loops, and model developers may parallelize replay even with a small number of checkpoints (e.g. 2 checkpoints: 3× parallelism), we need to relax our invariant to account for periodic checkpointing. Specifically, due to the nested loops structure of model training, we introduce n_i and k_i:

$$k_i M_i < n_i \epsilon C_i \quad \Rightarrow \quad \frac{M_i}{C_i} < \frac{n_i \epsilon}{k_i} \tag{1}$$

3.2.2 ***The Replay Latency Invariant.*** To avoid regret, record-replay should always be faster than two vanilla executions (with neither overhead nor speedups). Even for hindsight logging workloads that do not permit partial replay, the speedups from parallel replay alone should more-than-offset the overhead incurred on record. Accounting for record overhead, we can assess each block i for this condition as follows:

$$M_i + R_i + \left(\frac{n_i}{G} - 1\right) C_i < n_i C_i \tag{2}$$

The -1 in Equation 2 accounts for the fact that each parallel worker resumes from a stored checkpoint and does not need to compute its first iteration. Because G is determined on replay and is not known during record, we satisfy the Replay Latency Invariant by testing Equation 3 instead. Equation 3 guarantees the Replay Latency Invariant as long as there is some parallelism ($G \geq 2$); we omit the details for brevity.

$$\begin{aligned} M_i + R_i < \frac{n_i}{k_i} C_i \quad \text{and} \quad R_i = c M_i \\ \Rightarrow \frac{M_i}{C_i} < \frac{n_i}{k_i (1 + c)} \end{aligned} \tag{3}$$

Because the time to restore is not known at record time, we assume that it is proportional to the time to materialize. Our naive

Table 2: Set of rules for static side-effect analysis. At most one rule is activated by each program statement. The rules are sorted in descending order of precedence.

Rule	Pattern	ΔChangeset
0	$v_1, ..., v_n = u_1, ..., u_m \wedge \exists v_i \in$ `Changeset`	`No Estimate`
1	$v_1, ..., v_n = obj.method(arg_1, ..., arg_m)$	$\{obj, v_1, ..., v_n\}$
2	$v_1, ..., v_n = func(arg_1, ..., arg_m)$	$\{v_1, ..., v_n\}$
3	$v_1, ..., v_n = u_1, ..., u_m$	$\{v_1, ..., v_n\}$
4	$obj.method(arg_1, ..., arg_m)$	$\{obj\}$
5	$func(arg_1, ..., arg_m)$	`No Estimate`

assumption is $c = 1.0$, and this estimate is refined after observing materialization and restoration times from record-replay. In our case, the average scaling factor over all measured workloads (Table 3) turned out to be $c = 1.38$.

3.2.3 ***The Joint Invariant.*** The Joint Invariant is automatically checked by SkipBlocks at record time for adapting the frequency of checkpointing. Blocks are tested after executing, but before materialization. By restricting memoization to blocks that pass the Joint Invariant test, `flor` simultaneously satisfies the Record Overhead and Replay Latency invariants. This follows from the fact that the Joint Invariant is derived algebraically from the two invariants.

$$\frac{M_i}{C_i} < \frac{n_i}{k_i + 1} \min\left(\frac{1}{1 + c}, \epsilon\right) \quad c = 1.38, \epsilon = 0.0667 \tag{4}$$

Note the $k_i + 1$ in Equation 4: this accounts for the fact that the test is performed after the execution of the block but before the materialization of its checkpoint. The goal is for the invariant to continue to hold if the checkpoint is materialized. We derive the Joint Invariant, Equation 4, from Equation 1 and Equation 3. Both invariants are satisfied when the computed ratio, M_i/C_i, is less than the minimum of both thresholds.

3.3 Instrumentation for Hands-Free Mode

As desired by the user, `flor` can instrument their model training code for automatic and efficient record-replay. The principal objectives of `flor` instrumentation are twofold:

(1) Memoization and periodic checkpointing by nesting loops inside a SkipBlock, and statically estimating their side-effects.
(2) Auto-parallelization of training replay by a syntax-directed transformation of loop iterators, enabling *pseudoresume* from partial checkpoints (Subsection 2.2.1).

3.3.1 ***Autorecording Model Training.*** The first goal of instrumentation is to efficiently and correctly memoize loop executions for the model developer—without their intervention. `flor` memoizes loops because, in machine learning, they correspond to long-running code, and unlike arbitrary "long-running code", loops can be detected statically. Ensuring correct and efficient memoization requires (i) capturing all of the loop's side-effects, and (ii) avoiding the capture of too many redundancies. Unfortunately, due to the language's dynamic features and extensive reliance on (compiled) C extensions, an exact and efficient side-effect analysis in Python is intractable [19, 39, 50]. Past work overcomes Python's analysis limitations by restricting the expressiveness of the language [6, 24], making some assumptions (e.g. that the variables don't change types [8]), or relying on user source annotations [58]. In a similar vein, we achieve efficient side-effect analysis by assuming that loop bodies in model training are predominantly written in PyTorch [41]. To the extent that loops deviate from our assumption, our static analysis will be unsafe (i.e. may misdetect side-effects), so we will automatically perform deferred correctness checks after replay and report any anomalies to the programmer. We find that our assumption holds frequently enough to be useful for hindsight logging purposes. Model developers do not typically build models or write training algorithms from scratch. Instead, they rely on popular machine learning frameworks such as PyTorch. Like many 3rd-party libraries, PyTorch has a well-defined interface by which it modifies the user's program in limited ways [2]. The effects of PyTorch on the user's program are limited to (i) assignments and (ii) encapsulated state updates from method calls. As a result, all the side-effects of PyTorch code can be detected statically, with two notable exceptions: when an optimizer modifies a model, and when a learning rate scheduler modifies an optimizer [2].

First, `flor` estimates a set of changes ("changeset") for each block using the six rules in Table 2. `flor` walks the abstract syntax tree statement by statement, testing which rule is activated by each statement. The changeset for a block accumulates the individual changes of its member statements. Rules have a precedence such that at most one is activated per statement. Statements that activate no rule are ignored. Next, `flor` performs a filtering step on the changeset to remove variables that are scoped to the body of the loop. `flor` removes from the changeset any variable that is defined in the body of the loop (henceforth "loop-scoped variable"), under the assumption that this variable is local to the loop and is not read after the end of the loop. Finally, we make use of our encoded library-specific knowledge to augment the changeset at runtime (this is the only step that is not done statically). For PyTorch, it suffices to encode two facts [2]: (i) the model may be updated via the optimizer; and (ii) the optimizer may be updated via the learning rate schedule. `flor` augments the changeset to include side-effects which were not detected by the rules, but which can be inferred from other elements in the changeset.

3.3.2 ***Autoparallelizing Replay.*** The second goal of instrumentation is to autoparallelize replay of model training, assuming partial checkpoints exist. Replay instrumentation consists of wrapping the main loop's iterator inside a `flor.generator` (line 10 in Figure 4), to model the *pseudoresume* behavior we covered earlier (in Subsection 2.2.1 and Figure 3). Generators define an iterator by a series of `yield` statements, and allow us to control global program state between iterations of the main loop; namely, toggle the SkipBlocks from a *skip* state to a *step-into* state between epochs (lines 3, 6 in Figure 4). We implement parallel replay by having every parallel worker (`NPARTS` in total) execute the same instrumented code (as in Figure 4), and `flor` sets `PID` to a different value for each worker so they work on distinct segments of the main loop.

3.3.3 ***Deferred Checks for Correctness.*** As we have discussed, Python's dynamic features and extensive reliance on (compiled) C extensions, make an exact and efficient side-effect analysis intractable. `flor`'s approach to detecting side-effects is efficient but

```python
def flor.generator(*args):
  pseudoresume_sgmnt, work_sgmnt = partition(*args)
  skipblock.set_state('replay', 'skip')
  for element in pseudoresume_sgmnt:
    yield element
  skipblock.set_state('replay', 'step_into')
  for element in work_sgmnt:
    yield element

for epoch in flor.generator(range(N), PID, NPARTS):
  ...
```

Figure 4: `flor` instrumentation nests the main loop's iterator inside a generator to parallelize replay (with *pseudoresume*). A generator defines an iterator, and enables us to control global state between iterations of the main loop.

unsafe: it may misdetect side-effects and thus fail to checkpoint sufficient state for correct replay. To mitigate risk, we automatically check that common user-observable state between record and replay matches [4]. The standard training metrics that get logged by default (e.g. the loss and accuracy) form a fairly unique fingerprint of a model's training characteristics, so it's hard to perturb state or data that the model depends on without this being reflected in one of the model's metrics. Consequently, at the end of replay, we run `diff`, and warn the user if the replay logs differ from the record logs in any way other than the statements added for hindsight logging.

4 EVALUATION

To assess `flor`'s ability to meet the goals of Section 1 in practice, we evaluated eight diverse machine learning workloads, taken from three separate benchmarks: classic computer vision, the General Language Understanding and Evaluation (GLUE) [59], and ML Perf [34] (Table 3). These workloads vary in their tasks, model architectures, execution time scales, and software engineering patterns; they are jointly representative of a large class of model training workloads. Every experiment was run on P3.8xLarge EC2 instances with 4 Tesla V100 GPUs, 64 GB of GPU memory in aggregate, 32 vCPUs, 244 GB of RAM, and an EBS bandwidth (IO throughput) of 7Gbps. The checkpoints generated by `flor` record were spooled from EBS to an S3 bucket by a background process.

4.1 Flor Record Overhead is Low

We compared the overhead added by manual periodic checkpointing, at a rate of once per epoch, against the overhead added by automatic `flor` record (Figure 5). We did not manually set the period for any of the `flor` record experiments. The checkpointing period was automatically calibrated by the mechanism in Section 3.2. **`flor` record does not add significant overhead to training**, so it may be enabled by default. Moreover, the **`flor` instrumentation library achieves a comparable outcome as hand-tuned periodic checkpointing**, with competitive performance, and without intervention from the user. For manual periodic checkpointing, we assume that each checkpoint is complete with respect to training. For `flor` record we only assume partial checkpoints: each checkpoint corresponds to the side-effects of the code block it memoizes (Section 2.1), but it may be incomplete with respect to training.

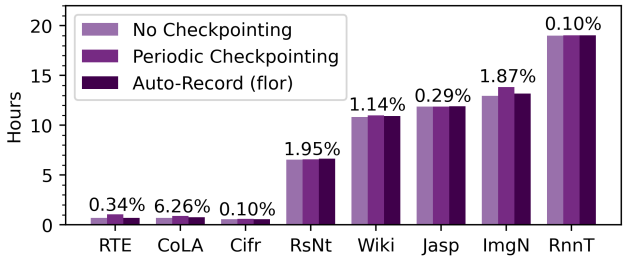

Figure 5: Comparison of model training times, with and without checkpointing, in hours. "Periodic Checkpointing" measures the time achievable when a model developer judiciously selects the contents of a checkpoint, at a frequency of once per epoch. The overhead added by `flor` Record is denoted by the text labels over each group of bars.

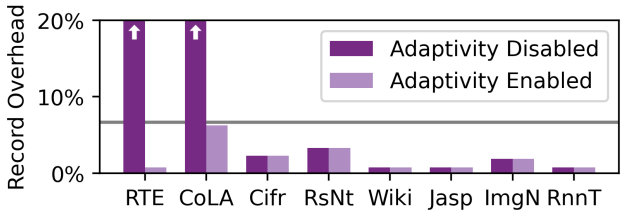

Figure 6: Impact of adaptivity on `flor` record overhead. The two upward arrows denote extreme values: adaptivity-disabled overhead is 91% for RTE and 28% for CoLA. The user-specifiable overhead tolerance (6.67%) is denoted by the gray horizontal line. No workload exceeds the overhead limit with adaptive checkpointing.

4.2 Flor Record Overhead is Adaptive

Different model developers have different sensitivies to overhead. In this section, we measured that `flor` record is able to adjust its checkpointing frequency to stay within the user-specifiable overhead limits (e.g. $\epsilon = 6.67\%$). The nested training loops in most model training workloads are memoized every epoch by `flor`'s adaptive checkpointing mechanism. This is because the time to materialize their checkpoints is negligible compared to the time it takes to execute them. In contrast, the sharp drop in overhead for fine-tuning workloads is due to their less frequent checkpointing (Figure 6). Fine-tuning workloads are checkpointed less frequently because their loops have poor materialization time to computation time ratios: their checkpoints are massive relative to their short execution times. This is the case because the vast majority of weights are frozen in model fine-tuning, so a loop execution quickly updates a small fraction of values in an enormous model [20]. We find that **adaptive checkpointing drastically reduces overhead** on model fine-tuning workloads (RTE & CoLA), and **ensures that no workload exceeds the user's overhead tolerance**.

Table 3: Computer vision and NLP benchmarks used in our evaluation.

Name	Benchmark	Task	Model	Dataset	Train/Tune	Epochs
RTE	GLUE	Recognizing Textual Entailment	RoBERTa	RTE	Fine-Tune	200
CoLA	GLUE	Language Acceptability	RoBERTa	CoLA	Fine-Tune	80
Cifr	Classic CV	Image Classification	Squeezenet	Cifar100	Train	200
RsNt	Classic CV	Image Classification	ResNet-152	Cifar100	Train	200
Wiki	GLUE	Language Modeling	RoBERTa	Wiki	Train	12
Jasp	MLPerf	Speech Recognition	Jasper	LibriSpeech	Train	4
ImgN	Classic CV	Image Classification	Squeezenet	ImageNet	Train	8
RnnT	MLPerf	Language Translation	RNN w/ Attention	WMT16	Train	8

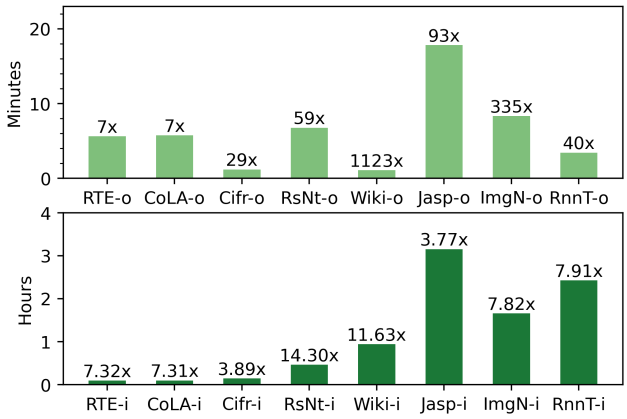

Figure 7: Replay latency, factored by the position of hindsight logging statements. The top plot reports partial and parallel replay speedups when the model developer probes only the outer main loop (as in line 13 of Figure 8). The bottom plot reports parallel-only replay speedups when the model developer probes the inner training loop and a full re-execution is needed (as in line 10 of Figure 8). Each workload uses as many machines, from a pool of four machines, as will result in parallelism gains. Text labels show speedup factors relative to naive re-execution.

4.3 Flor Replay Latency is Low

In this section, we measure the replay speedups achieved by `flor` replay, assuming `flor` record checkpoints were materialized during training. Consequently, we measure the replay speedups when `flor` instruments model developers' code end-to-end for efficient hindsight logging—without intervention from the developer.

Replay latencies are query dependent: they depend on the position of hindsight logging statements in the code. In cases when the model developer probes only the outer loop of training (as in line 13 of Figure 8), **partial replay can provide latencies on the order of minutes, even when model training takes many hours to execute**. This is achieved by skipping unnecessary recomputation with loop memoization (e.g. skipping the nested training loop). The top subplot in Figure 7 shows outer-loop probe latencies for each of our models. Note the improvements range from 7× to 1123×—with the more significant improvements favoring the longer experiments (recall Figure 5). When the model developer logs data post-hoc from the inner training loop (as in line 10 in Figure 8), then that loop must be re-executed on replay, and it will not contribute to savings from loop memoization. For these workloads, we will need to rely on parallelism to reduce latencies. We measured the hindsight logging latencies when a full re-execution of model training was necessary by running replay on multiple machines—this is shown in the bottom subplot in Figure 7. Assuming no work or guidance from the model developer, beyond the insertion of a couple hindsight logging statements, **`flor` automatically parallelizes and conditionally skips computation on the re-execution of model training** (Subsection 3.3.2).

The parallel replay workloads used as many machines from the pool of 4 machines as would provide further parallelism gains. Each machine has 4 GPUs. In the limit, every epoch may re-execute in parallel, but the degree of parallelism may be increased even further by checkpointing additional state, which we leave as future work.

4.4 Ideal Parallelism and Scale-out

Next, we compare the performance of parallel replay with checkpoint resume against parallel replay with checkpoint pseudoresume (refer to Subsection 2.2.1). An expert model developer, such as Judy, who does periodic checkpointing by-hand can ensure that the checkpoints are complete with respect to training. Thus, they can achieve the *checkpoint resume* performance in Figures 10 and 11. On the other hand, when `flor` instruments training code on behalf of the developer, it will rely on *checkpoint pseudoresume*, because as we discussed earlier, `flor` cannot automatically ensure that its checkpoints are complete with respect to training, and it assumes that the checkpoints are partial. Our results show that, although pseudoresuming training adds initialization overhead, this overhead is amortized through the course of parallel replay, such that **there is a negligible difference between checkpoint resume and checkpoint pseudoresume**.

In Figure 10, we measured parallel replay performance, and observe that **`flor` replay achieves near-ideal parallelism**. Ideal parallelism is denoted by the gray horizontal line in each subplot (Figure 10). These results are possible because model training *replay* is embarrassingly parallel given (complete or partial) checkpoints.

```
1  init_globals()
2  checkpoint_resume(args, (net, optimizer))
3  for epoch in range(args.start, args.stop):
4    if skipblock.step_into(...):
5      for batch in training_data:
6        predictions = net(batch.X)
7        avg_loss = loss(predictions, batch.y)
8        avg_loss.backward()
9        optimizer.step()
10       tensorboard.add_histogram(net.params())
11   skipblock.end(net, optimizer)
12   evaluate(net, test_data)
13   tensorboard.add_overlays(net, test_data)
```

Figure 8: Model training example with checkpoint resume. Lines 10 and 13 correspond to hindsight logging statements, or logging statements added after training.

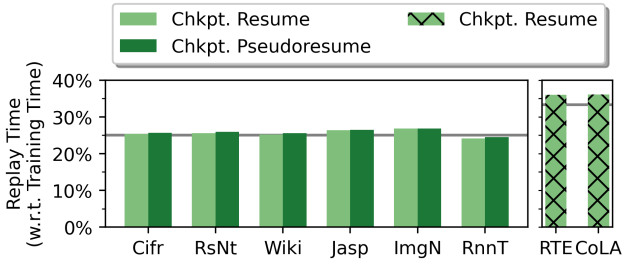

Figure 10: Parallel replay time of model training jobs ($4x$ parallelism), as fraction of a serial re-execution. RTE & CoLA only have 6 work partitions each, so parallelism on 4 GPUs leads to at best $2/6 = 33\%$ replay time.

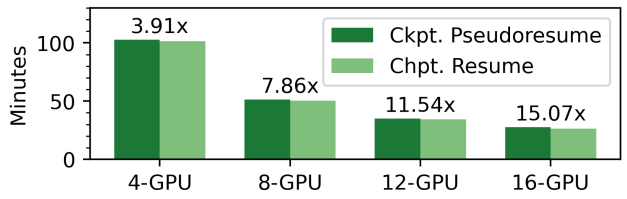

Figure 11: Replay time using GPUs from multiple P3.8xLarge machines, on experiment RsNt. The "checkpoint resume" speedup relative to a sequential execution is denoted by the text labels.

Because parallel workers do not need to communicate or coordinate, `flor` replay is especially well-suited for elastic and horizontally scalable cloud computing, in which **it can scale out to more GPUs at low marginal costs**. To assess our scaling performance, in Figure 11 we illustrate the incremental speedup as we add 4-GPU machines. We choose RsNt as our experiment because it has 200 epochs to parallelize. The modest gap between our results and ideal

```
1  init_globals()
2  for epoch in range(0, args.stop):
3    if skipblock.step_into(...
4        && epoch >= args.start):
5      for batch in training_data:
6        predictions = net(batch.X)
7        avg_loss = loss(predictions, batch.y)
8        avg_loss.backward()
9        optimizer.step()
10       tensorboard.add_histogram(net.params())
11   skipblock.end(net, optimizer)
12   lr_scheduler.step()
13   evaluate(net, test_data)
14   tensorboard.add_overlays(net, test_data)
```

Figure 9: Model training example with checkpoint pseudoresume. Lines 10 and 14 correspond to hindsight logging statements (added after training).

here is due to load balancing limitations: balancing 200 epochs over 16 parallel workers results in each worker doing up to 13 epochs of work. Consequently, the maximum achievable speedup on 16 GPUs is $\frac{200}{13}$: 15.38×.

5 RELATED WORK

ML lifecycle management. The machine learning lifecycle encompasses many tasks, including model design and development, training, validation, deployment, inference, and monitoring [14]. There is a wide range of research and tooling being developed to support these many tasks. ML lifecycle management is especially challenging because it involves many cycles of trial-and-error [25], and its dependencies are hard to scope [47]. When something goes wrong, ML engineers may need to rollback their model to an earlier version [36, 57], inspect old versions of the training data [21, 23, 33], or audit the code that was used for training [35, 46]. Those activities require the proper management, versioning, and provenance tracking of data, models, code, and other context; existing solutions provide some support [9, 17, 26, 27, 63]. Hindsight logging is a novel contribution in the lifecycle, and its minimalist, low-friction interface makes it complementary to the prior work. `flor` is designed to be compatible with any of the tools in the Python ecosystem. In terms of training libraries, we have focused on PyTorch, but adopting another training library involves only encoding any side-effects in the library's API (Section 3.3.1).

Model Debugging. There are many tools and techniques for helping users understand the behavior of their models [1, 32, 44, 45, 48], and for inspecting model internals [29, 40, 43, 55, 56]. These techniques only inspect models, so they are complementary to our work—which focuses on the execution data generated while training the models. The value of execution data is evidenced by widespread use of domain-specific loggers and visualization tools for that data, including TensorBoard [15], MLflow Tracking [63], and WandB [3]. Hindsight logging allows developers to keep their current logging practices and tools, and use them to "query the past".

Partial Materialization. Inspired by classical work on materialized views [11], a new body of work addresses partial materialization of state in ML workflows, to aid in iterative tasks like debugging. As representative examples, Columbus [64] accelerates the exploration of feature selection by choosing to cache feature columns; Helix [61] focuses on choosing to cache and reuse the outputs of black-box workflow steps; Mistique [56] focuses on techniques for compressing model-related state and deciding whether to materialize or recompute. These systems introduce bespoke languages for pre-declaring what to capture prior to computation; they also provide custom query APIs to interrogate the results. Hindsight logging is complementary: it enables post-hoc materialization in cases when it was *not* prespecified. Precisely because `flor` does not dictate a new API, it is compatible with this prior work: users of these systems (or any library with pre-declared annotations) can benefit from `flor` to add annotations in hindsight, and benefit from `flor`'s efficient replay to add materialized state. At a more mechanistic level, some of the policies and mechanisms from this work (e.g., the model compression of Mistique) could be adapted into hindsight logging context to further improve upon our results.

Recovery and Replay Systems. Our techniques are inspired by literature on both database recovery and program replay. Hindsight logging is a redo-only workload, and we use a "physiological" approach [16]: in our view, a model training script is a complete logical log (in the WAL sense) of a model training execution, and occasional physical checkpoints serve solely to speed up redo processing. Parallel and selective redo recovery was studied as early as ARIES [37, 60]. Parallelism in those techniques is data-partitioned and recovers the most recent consistent state; we are in essence time-partitioned and recover all prior states. In that sense our work bears a resemblance to multiversion storage schemes from POSTGRES [52] onward to more recent efforts (e.g., [30, 38]). These systems focus on storing complete physical versions, which is infeasible in our setting due to constraints on runtime overhead.

Numerous program record-replay systems have been used in the past for less data-oriented problems. Jalangi is a system for dynamic program analysis that automatically records the required state during normal processing, and enables high-fidelity selective replay [49]. This is achieved by identifying and storing memory loads that may not be available at replay time, using a "shadow memory" technique. Unlike `flor`, Jalangi replay has strict correctness guarantees. `flor` uses side-effect analysis rather than shadow memory because the former is lighter on overhead: in this sense, we risk replay anomalies to reduce record overhead and replay latency.

Prior work on Output Deterministic Replay [4] makes a similar trade-off as we do. However that work pays for higher latencies to enable reproduction of nondeterministic bugs; we can avoid that overhead in Python model-training scenarios because sources of non-determinism may be captured, and model-training frameworks are increasingly designed for reproducibility. An interesting line of work enables reverse replay with relatively high fidelity and without overhead by using memory dumps on a crash [12]—this impressive result is made possible by the spatial locality of bugs in the vicinity of execution crashes; one complication with model debugging is that training errors, such as over-fitting, may not crash the program. We borrow the SkipBlock language construct from Chasins and Bodik's record-replay system for web scraping [10].

6 CONCLUSION

At every step of exploration, model developers routinely track and visualize time series data to assess learning. Most model developers log training metrics such as the loss and accuracy by default, but there soon arise important differences between what additional training data model developers log—with major implications for data management. In contrast to conservative logging, optimistic logging is an agile and lazy practice especially well-suited to early and unstructured stages of exploratory model development. In optimistic logging, model developers log training metrics such as the loss and accuracy by default, and defer collection of more expensive data until analysis time, when they may restore it selectively with hindsight logging. In the common case, or fast path, model developers get all the relevant information from the training loss, and move on. In exceptional cases, however, training replay may be necessary for post-hoc data restoration. In this paper, we documented a system of method for efficient record-replay of model training. Methodical hindsight logging consists of: (i) periodic checkpointing, (ii) block memoization, and (iii) checkpoint resume. To extend the benefits of methodical hindsight logging to novices and experts alike, we open source the `flor` suite for hindsight logging, which includes tools for: (i) low-overhead background logging, (ii) adaptive periodic checkpointing, and (iii) end-to-end instrumentation for efficient and fully automatic record-replay. We evaluated methodical hindsight logging to show that it achieves the goal of efficient record-replay, and then compared the instrumentation library provided by `flor` against the methodical expert-tuned approach, and find that the performance is comparable.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their patient, thoughtful, and thorough feedback. Their recommendations substantially improved the quality of the paper. Malhar Patel, Sona Jeswani, and Karina Uppal provided valuable help with the development of earlier versions of `flor`. We would also like to thank Eric Baldeschwieler, Paul Barham, Paras Jain, Chris Severs, Anirudh Srinivasan, Marvin Theimer, Sindhuja Venkatesh, and Doris Xin for helpful suggestions and discussion. In addition to NSF CISE Expeditions Award CCF-1730628, NSF Grant No. 1564351, DOE Grant No. DE-SC0016934, and an NSF Graduate Research Fellowship, this research is supported by gifts from Alibaba, Amazon Web Services, Ant Financial, CapitalOne, Ericsson, Facebook, Futurewei, Google, Intel, Microsoft, Nvidia, Scotiabank, Splunk and VMware.

REFERENCES

[1] 2020. Google PAIR. https://research.google/teams/brain/pair/.
[2] 2020. PyTorch Documentation. pytorch.org/docs.
[3] 2020. Weights & Biases. wandb.com.
[4] Gautam Altekar and Ion Stoica. 2009. ODR: output-deterministic replay for multicore debugging. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 193–206.
[5] Dario Amodei, Chris Olah, Jacob Steinhardt, Paul Christiano, John Schulman, and Dan Mané. 2016. Concrete problems in AI safety. *arXiv preprint arXiv:1606.06565* (2016).
[6] Davide Ancona, Massimo Ancona, Antonio Cuni, and Nicholas D Matsakis. 2007. RPython: a step towards reconciling dynamically and statically typed OO

languages. In *Proceedings of the 2007 symposium on Dynamic languages*. 53–64.
[7] Sotiris Apostolakis, Ziyang Xu, Greg Chan, Simone Campanoni, and David I August. 2020. Perspective: A sensible approach to speculative automatic parallelization. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 351–367.
[8] John Aycock. 2000. Aggressive type inference. *language* 1050 (2000), 18.
[9] Denis Baylor, Eric Breck, Heng-Tze Cheng, Noah Fiedel, Chuan Yu Foo, Zakaria Haque, Salem Haykal, Mustafa Ispir, Vihan Jain, Levent Koc, et al. 2017. Tfx: A tensorflow-based production-scale machine learning platform. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 1387–1395.
[10] Sarah Chasins and Rastislav Bodik. 2017. Skip blocks: reusing execution history to accelerate web scripts. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–28.
[11] Rada Chirkova and Jun Yang. 2011. Materialized views. *Databases* 4, 4 (2011), 295–405.
[12] Weidong Cui, Xinyang Ge, Baris Kasikci, Ben Niu, Upamanyu Sharma, Ruoyu Wang, and Insu Yun. 2018. REPT: Reverse Debugging of Failures in Deployed Software. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 17–32.
[13] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc'aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, et al. 2012. Large scale distributed deep networks. In *Advances in neural information processing systems*. 1223–1231.
[14] Rolando Garcia, Vikram Sreekanti, Neeraja Yadwadkar, Daniel Crankshaw, Joseph E Gonzalez, and Joseph M Hellerstein. 2018. Context: The missing piece in the machine learning lifecycle. In *KDD CMI Workshop*, Vol. 114.
[15] Google. 2020. TensorBoard. tensorflow.org/tensorboard.
[16] Jim Gray and Andreas Reuter. 1992. *Transaction processing: concepts and techniques*. Elsevier.
[17] Joseph M Hellerstein, Vikram Sreekanti, Joseph E Gonzalez, James Dalton, Akon Dey, Sreyashi Nag, Krishna Ramachandran, Sudhanshu Arora, Arka Bhattacharyya, Shirshanka Das, et al. 2017. Ground: A Data Context Service.. In *CIDR*.
[18] Sepp Hochreiter. 1998. The vanishing gradient problem during learning recurrent neural nets and problem solutions. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems* 6, 02 (1998), 107–116.
[19] Alex Holkner and James Harland. 2009. Evaluating the dynamic behaviour of Python applications. In *ACSC*.
[20] Jeremy Howard and Sebastian Ruder. 2018. Universal language model fine-tuning for text classification. *arXiv preprint arXiv:1801.06146* (2018).
[21] Silu Huang, Liqi Xu, Jialin Liu, Aaron Elmore, and Aditya Parameswaran. 2017. OrpheusDB: bolt-on versioning for relational databases. *arXiv preprint arXiv:1703.02475* (2017).
[22] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. 2016. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and< 0.5 MB model size. *arXiv preprint arXiv:1602.07360* (2016).
[23] Christian S Jensen and Richard T Snodgrass. 1999. Temporal data management. *IEEE Transactions on knowledge and data engineering* 11, 1 (1999), 36–44.
[24] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. 2015. Numba: A llvm-based python jit compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*. 1–6.
[25] Angela Lee, Doris Xin, Doris Lee, and Aditya Parameswaran. 2020. Demystifying a dark art: Understanding real-world machine learning model development. *arXiv preprint arXiv:2005.01520* (2020).
[26] Yunseong Lee, Alberto Scolari, Byung-Gon Chun, Marco Domenico Santambrogio, Markus Weimer, and Matteo Interlandi. 2018. PRETZEL: Opening the black box of machine learning prediction serving systems. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 611–626.
[27] Edo Liberty, Zohar Karnin, Bing Xiang, Laurence Rouesnel, Baris Coskun, Ramesh Nallapati, Julio Delgado, Amir Sadoughi, Yury Astashonok, Piali Das, et al. 2020. Elastic Machine Learning Algorithms in Amazon SageMaker. *SIGMOD* (2020), 14–19.
[28] Eric Liu. 2020. *Low Overhead Materialization with Flor*. Master's thesis. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2020/EECS-2020-79.html
[29] Mengchen Liu, Jiaxin Shi, Zhen Li, Chongxuan Li, Jun Zhu, and Shixia Liu. 2016. Towards better analysis of deep convolutional neural networks. *IEEE transactions on visualization and computer graphics* 23, 1 (2016), 91–100.
[30] David B Lomet and Feifei Li. 2009. Improving transaction-time DBMS performance and functionality. In *2009 IEEE 25th International Conference on Data Engineering*. IEEE, 581–591.
[31] Lu Lu, Yeonjong Shin, Yanhui Su, and George Em Karniadakis. 2019. Dying relu and initialization: Theory and numerical examples. *arXiv preprint arXiv:1903.06733* (2019).
[32] Scott M Lundberg and Su-In Lee. 2017. A unified approach to interpreting model predictions. In *Advances in neural information processing systems*. 4765–4774.
[33] Michael Maddox, David Goehring, Aaron J Elmore, Samuel Madden, Aditya Parameswaran, and Amol Deshpande. 2016. Decibel: The relational dataset branching system. In *Proceedings of the VLDB Endowment. International Conference on Very Large Data Bases*, Vol. 9. NIH Public Access, 624.
[34] Peter Mattson, Christine Cheng, Cody Coleman, Greg Diamos, Paulius Micikevicius, David Patterson, Hanlin Tang, Gu-Yeon Wei, Peter Bailis, Victor Bittorf, et al. 2019. Mlperf training benchmark. *arXiv preprint arXiv:1910.01500* (2019).
[35] Hui Miao and Amol Deshpande. 2018. ProvDB: Provenance-enabled Lifecycle Management of Collaborative Data Analysis Workflows. *IEEE Data Eng. Bull.* 41, 4 (2018), 26–38.
[36] Hui Miao, Ang Li, Larry S Davis, and Amol Deshpande. 2017. Modelhub: Deep learning lifecycle management. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. IEEE, 1393–1394.
[37] Chandrasekaran Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. 1992. ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems (TODS)* 17, 1 (1992), 94–162.
[38] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. 2015. Fast serializable multi-version concurrency control for main-memory database systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 677–689.
[39] Jens Nicolay, Carlos Noguera, Coen De Roover, and Wolfgang De Meuter. 2015. Detecting function purity in JavaScript. In *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 101–110.
[40] Chris Olah, Arvind Satyanarayan, Ian Johnson, Shan Carter, Ludwig Schubert, Katherine Ye, and Alexander Mordvintsev. 2018. The building blocks of interpretability. *Distill* 3, 3 (2018), e10.
[41] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. PyTorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*. 8024–8035.
[42] Alexander Ratner, Dan Alistarh, Gustavo Alonso, Peter Bailis, Sarah Bird, Nicholas Carlini, Bryan Catanzaro, Eric Chung, Bill Dally, Jeff Dean, et al. 2019. Sysml: The new frontier of machine learning systems. *arXiv preprint arXiv:1904.03257* (2019).
[43] Paulo E Rauber, Samuel G Fadel, Alexandre X Falcao, and Alexandru C Telea. 2016. Visualizing the hidden activity of artificial neural networks. *IEEE transactions on visualization and computer graphics* 23, 1 (2016), 101–110.
[44] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. 2016. "Why should I trust you?" Explaining the predictions of any classifier. In *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*. 1135–1144.
[45] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. 2018. Anchors: High-precision model-agnostic explanations. In *Thirty-Second AAAI Conference on Artificial Intelligence*.
[46] Sebastian Schelter, Joos-Hendrik Böse, Johannes Kirschnick, Thoralf Klein, and Stephan Seufert. 2017. Automatically tracking metadata and provenance of machine learning experiments. In *Machine Learning Systems Workshop at NIPS*. 27–29.
[47] D. Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, and Michael Young. 2014. Machine Learning: The High Interest Credit Card of Technical Debt. *SE4ML: Software Engineering for Machine Learning (NIPS Workshop)* (2014).
[48] Ramprasaath R Selvaraju, Michael Cogswell, Abhishek Das, Ramakrishna Vedantam, Devi Parikh, and Dhruv Batra. 2017. Grad-cam: Visual explanations from deep networks via gradient-based localization. In *Proceedings of the IEEE international conference on computer vision*. 618–626.
[49] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. 2013. Jalangi: a selective record-replay and dynamic analysis framework for JavaScript. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. 488–498.
[50] David A Spuler and A. Sayed Muhammed Sajeev. 1994. Compiler detection of function call side effects. *Informatica* 18, 2 (1994), 219–227.
[51] Ion Stoica, Dawn Song, Raluca Ada Popa, David Patterson, Michael W Mahoney, Randy Katz, Anthony D Joseph, Michael Jordan, Joseph M Hellerstein, Joseph E Gonzalez, et al. 2017. A Berkeley view of systems challenges for AI. *arXiv preprint arXiv:1712.05855* (2017).
[52] Michael Stonebraker. 1987. *The design of the Postgres storage system*. Morgan Kaufmann Publishers Burlington.
[53] Emma Strubell, Ananya Ganesh, and Andrew McCallum. 2019. Energy and policy considerations for deep learning in NLP. *arXiv preprint arXiv:1906.02243* (2019).
[54] RS Sutton. 2019. The Bitter Lesson. incompleteideas.net/IncIdeas/BitterLesson.html.
[55] F-Y Tzeng and K-L Ma. 2005. *Opening the black box-data driven visualization of neural networks*. IEEE.
[56] Manasi Vartak, Joana M F. da Trindade, Samuel Madden, and Matei Zaharia. 2018. Mistique: A system to store and query model intermediates for model diagnosis. In *Proceedings of the 2018 International Conference on Management of*

Data. 1285–1300.

[57] Manasi Vartak, Harihar Subramanyam, Wei-En Lee, Srinidhi Viswanathan, Saadiyah Husnoo, Samuel Madden, and Matei Zaharia. 2016. ModelDB: a system for machine learning model management. In *Proceedings of the Workshop on Human-In-the-Loop Data Analytics*. ACM, 14.

[58] Michael M Vitousek, Andrew M Kent, Jeremy G Siek, and Jim Baker. 2014. Design and evaluation of gradual typing for Python. In *Proceedings of the 10th ACM Symposium on Dynamic languages*. 45–56.

[59] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R Bowman. 2018. GLUE: A multi-task benchmark and analysis platform for natural language understanding. *arXiv preprint arXiv:1804.07461* (2018).

[60] Gerhard Weikum and Gottfried Vossen. 2001. *Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery*. Elsevier.

[61] Doris Xin, Stephen Macke, Litian Ma, Jialin Liu, Shuchen Song, and Aditya Parameswaran. 2018. Helix: Holistic optimization for accelerating iterative machine learning. *Proceedings of the VLDB Endowment* 12, 4 (2018), 446–460.

[62] Ding Yuan, Soyeon Park, Peng Huang, Yang Liu, Michael M Lee, Xiaoming Tang, Yuanyuan Zhou, and Stefan Savage. 2012. Be conservative: enhancing failure diagnosis with proactive logging. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. 293–306.

[63] Matei Zaharia, Andrew Chen, Aaron Davidson, Ali Ghodsi, Sue Ann Hong, Andy Konwinski, Siddharth Murching, Tomas Nykodym, Paul Ogilvie, Mani Parkhe, et al. 2018. Accelerating the machine learning lifecycle with MLflow. *Data Engineering* (2018), 39.

[64] Ce Zhang, Arun Kumar, and Christopher Ré. 2016. Materialization optimizations for feature selection workloads. *ACM Transactions on Database Systems (TODS)* 41, 1 (2016), 1–32.

[65] Wenting Zheng, Stephen Tu, Eddie Kohler, and Barbara Liskov. 2014. Fast databases with fast durability and recovery through multicore parallelism. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. 465–477.