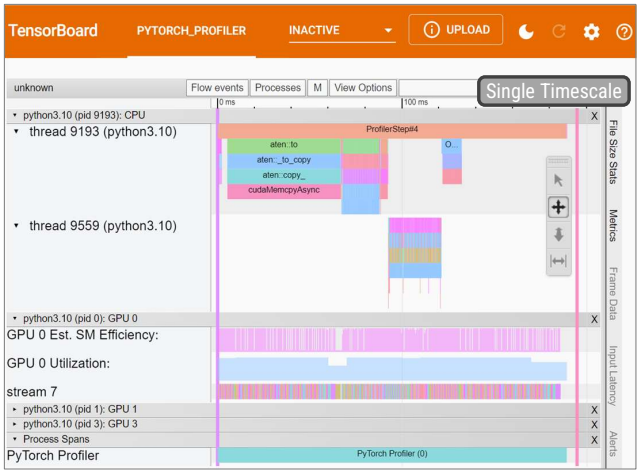
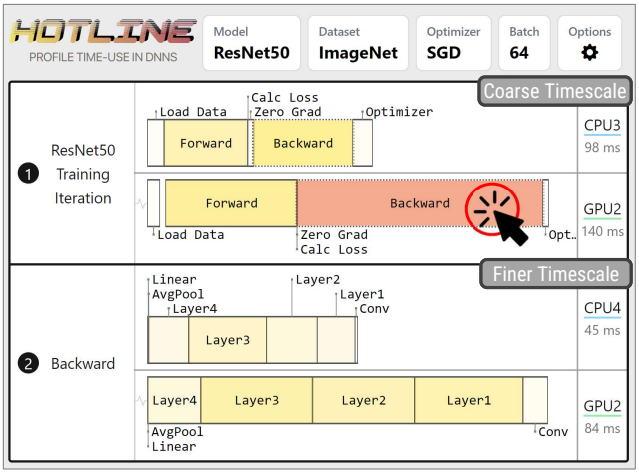
HOTLINE PROFILER: AUTOMATIC ANNOTATION AND A MULTI-SCALE TIMELINE FOR VISUALIZING TIME-USE IN DNN TRAINING

Daniel Snider [1 2] **Fanny Chevalier** [1 3] **Gennady Pekhimenko** [1 2]



(a) Existing: TensorBoard



(b) Ours: HOTLINE

Figure 1. Hotline generates annotations of DNN concepts that existing runtime tracing profilers do not currently offer and visualizes them in a novel Multi-Scale Timeline. HOTLINE uses color and size of operations to highlight bottlenecks across the stack. Users can interactively drill down to progressively smaller timescales and view kernel runtimes at the lowest level (see Appendix B for examples).

ABSTRACT

Profiling is a standard practice used to investigate the efficiency of software and hardware operation at runtime and is a crucial part of proving new concepts, debugging problems, and optimizing performance. However, most machine learning (ML) developers find profiling secondary to their goal of improving model accuracy or just too difficult (especially with existing ML tools). As a result, profiling is frequently an afterthought, and so many ML developers rely on opaque metrics such as iteration time and GPU utilization which give little insight into why ML training may be slow. This leads developers to spend excessive time investigating performance issues. In this work, we aim to provide better tools to the large group of ML developers who currently do not profile their deep neural network (DNN) training workloads or are not happy with existing tools.

To help ML developers investigate and understand time-use in DNN training, we propose HOTLINE, a novel profiler designed specifically for *runtime bottleneck identification*. HOTLINE is the first profiler to automatically annotate a standard data format for program runtime traces with DNN concepts that most ML developers are familiar with, i.e. the DNN training loop and model architecture. HOTLINE does so *without* modifying DNN libraries or making use of vendor-specific tools and introduces no additional overhead on measurements. We further introduce noise reduction techniques and a multi-scale timeline visualization to make the presentation of DNN runtime data more *insightful, familiar, and easy to navigate*. We demonstrate HOTLINE's utility through in-depth case studies of finding bottlenecks in real-world DNN applications and we report on a user study with 17 software developers in which most participants were able to perform common performance investigation tasks in under 30 seconds (avg = 26 sec) and further commented that HOTLINE's visualization *"takes less time to find insights compared to existing approaches"*. Source code: `https://github.com/UofT-EcoSystem/hotline`.

[1]Department of Computer Science, University of Toronto, Toronto, Ontario, Canada [2]Vector Institute, Toronto, Ontario, Canada [3]Department of Statistics, University of Toronto, Toronto, Ontario, Canada. Correspondence to: Daniel Snider <dans@cs.toronto.edu>.

Proceedings of the 6^{th} MLSys Conference, Miami Beach, FL, USA, 2023.

1 INTRODUCTION

The recent success of deep learning (DL) (He et al., 2016; Devlin et al., 2018; Hannun et al., 2014) has resulted in an increasing amount of compute and energy dedicated to *training* deep neural network (DNN) models (Amodei & Hernandez, 2018; Bianco et al., 2018; Horowitz, 2014; Strubell et al., 2019). Much of the success of DL is credited to increasing algorithm and system efficiency (Sutton, 2019; Hernandez & Brown, 2020; Eassa & Burc Eryilma, 2022). However, inefficient *bottlenecks* in DNN training are likely still a widespread problem, because DNN development (1) is often performed by ML developers and data scientists with limited background in system-level optimization, (2) is developed through ad-hoc prototyping, and (3) makes use of rapidly evolving DNN libraries. Recent works (Chen et al., 2018b; Zheng et al., 2020b; Vasilache et al., 2018) and DL libraries (Abadi et al., 2016; Frostig et al., 2018; Chen et al., 2015; Chetlur et al., 2014) try to make performance optimizations automatic for developers, but there is evidence that those works do not cover all major optimization possibilities (Ivanov et al., 2021; Shacklett et al., 2021; Petrenko et al., 2020; Dalton et al., 2020; Andoorveedu et al., 2022). There remains room for developers to recognize and guide optimizations; but who does this, how, and how can we do it better?

The desire to find optimization opportunities motivates the need for *profiling tools* to investigate the operation of software and hardware at runtime (Joukov et al., 2006) and *visualizations* to help users analyze profiler data, support reasoning, and communicate findings (Munzner, 2014). There exists a limited number of ML system performance experts and they typically use custom (Hu et al., 2022; Zhu et al., 2020; 2018; Gleeson et al., 2021) or expert-level profiling tools (NVIDIA, 2018b; Intel, 2020; Google, 2022) to find optimization opportunities. We strongly believe that we cannot bank on scaling up the number of ML system performance experts. Instead, a more sustainable and valuable approach is to equip the larger community of ML/DNN algorithm and model developers[1] with better tools. Even the most basic questions *"What is slow?"* and *"Why is it slow?"* are challenging to answer with existing DNN profiling tools (Google, 2015; NVIDIA, 2018b). While these tools generate the necessary data to investigate these questions, processing and understanding this data is difficult.

Existing approaches fail to distill the overwhelming amount of information generated into insights that are clearly expressed and easily interpreted (Bohnet et al., 2009; Weber et al., 2015). We observe that interpreting DNN training runtime traces, which include the runtime of every executed operation, is challenged by several factors including *information overload* (e.g., the backward pass of an RNN-T model (MLCommons, 2021) is comprised of 363,095 operations), *disparate granularities* (e.g., a single timeline view of very short and very long operations is hard to see, navigate, and compare), and *discontiguous execution* (e.g., relationships between operations may be hard to follow due to asynchronous and distributed processing). As a result, profiling is often an afterthought and so ML developers mainly rely on basic metrics such as training iteration time, or coarse system efficiency metrics such as GPU utilization. These metrics are high-level, aggregate, and opaque. They give little insight into the system operation of DNN training and so the developers may spend excessive time investigating performance issues (they *"search for a needle in a haystack"*) or write code without understanding the problem (they *"try stuff until it works"*).

Our research aims to address these challenges from the perspective of **identifying bottlenecks**, rather than follow-up questions of *"Why is it a bottleneck?"*, and *"What should be done about the bottleneck?"*. In our work, we identify bottlenecks using detailed measurements of how time is spent during DNN training, rather than resource utilization where high utilization could be a good or bad sign. Our approach differs from prior profiling techniques because we leverage two key observations based on the unique characteristics of common ML/DNN training workloads. First, we observe that DNN operations in runtime traces are serialized in an order that is *highly predictable*. Second, these operations appear in runtime traces with names that are *recognizable* and can be attributed to known portions of the DNN training. These observations allowed us to develop an *automatic offline annotation algorithm* for minimal/lightweight tracing profilers to generate new annotations which shed light on where bottlenecks lie without incurring additional overhead on timing measurements and minimal instrumentation effort required in DL source code.

In this paper, we propose HOTLINE, a novel profiler designed specifically for *runtime bottleneck identification* in DNN training. HOTLINE generates three types of annotations that existing profilers do not offer: (1) the steps of the DNN training loop, (2) the architecture of the DNN model, and (3) arbitrary but interesting sections not defined in the user's DL source code, such as inter-GPU communication. HOTLINE supports users who manually add custom annotations into traces at runtime. Those annotations will not interfere, they will be treated as arbitrary sections, and they will be displayed by HOTLINE. Furthermore, HOTLINE applies a suite of noise reduction heuristics to annotations that summarize, rename for brevity, or hide redundancies to trade information completeness for improved interpretation.

A key contribution of HOTLINE is the introduction of the *Multi-Scale Timeline* (Richter et al., 1999) to system profil-

[1] In the rest of the paper, we will simply refer to our target audience as "developers" for short.

ing of DNN model training. A multi-scale timeline allows users to see multiple timescales in one view, i.e. low-level details and high-level context (see Figure 1), which eases understanding. HOTLINE initially shows the runtime of only the highest-level operations in the DNN training loop to present a simple and familiar view. Users can drill down using *Semantic Zooming* (Aigner, 2014), an interaction technique that is faster, simpler, and more exact than using pan-and-zoom which is found in existing runtime trace viewers.

We evaluate the usefulness and effectiveness of HOTLINE through a series of case studies. In the first case study, we show how HOTLINE enables ML developers to see how decisions about model architecture affect runtime; how using HOTLINE can be a positive learning experience; and, for example, how quickly we can find an optimization opportunity worth up to 40% in training a popular ResNet-50 model (He et al., 2016). In the second case study, we show how HOTLINE can perform rapid low-level investigations by using HOTLINE's "Open with Perfetto" button to isolate interesting portions of the runtime trace and quickly switch to a more detailed profiling tool (Google, 2022). For example, using HOTLINE to view only the backward pass of an RNN-T model on 4 GPUs, we discovered that the GPU is only executing kernels 30% of the time and is bottlenecked by CUDA API launch overhead. In a third case study, we show how HOTLINE's annotation of arbitrary sections of training enabled us to find unexpected bottlenecks. For example, we find in training ResNet-50 on 4 GPUs that PyTorch's default parallel training mode spends 67% of the forward pass replicating the DNN model to all GPUs and 50% of backward pass reducing gradients onto 1 GPU from the other 3 GPUs. This may be a surprise to users because they may not be aware of these stages or that they are so slow.

Additionally, we collected qualitative feedback from software developers at our institution on our design concept, not the implemented tool itself, to guide and validate features before implementation. We report on evidence that HOTLINE's visual approach is a good match for the user's task and mental model of runtime profiling. For example, 97% of the time participants were able to perform common ML profiling tasks without help when using a non-interactive mockup of HOTLINE.

Our contributions are summarized as follows:

- We observe fundamental challenges in DNN training runtime traces which make finding insights and bottlenecks difficult, and then develop a new strategy to attract developers to profiling.
- We propose HOTLINE, the first DNN profiler to annotate DNN concepts with no impact on timing measurements and minimal instrumentation. HOTLINE denoises runtime traces to make insights clearer and visualizes results in an interactive multi-scale timeline that is easier to navigate and quicker to isolate interesting portions of event traces than with existing tools.
- We present case studies of how HOTLINE can be used to identify and understand time-use bottlenecks in real-world DNN training across a variety of models including ResNet, RNN-T, and Transformer.
- We report on a qualitative user study of software developers at our institution in which 94% (16/17) of participants said that HOTLINE's visualization design concept *"helps them understand program runtime"*, and 82% (14/17) said that HOTLINE's visualization *"takes less time to find insights compared to existing approaches"*.

2 WHY MANY ML DEVELOPERS DO NOT PROFILE, EVEN THOUGH THEY SHOULD

2.1 Background on Profiling

Profiling is a crucial part of proving new concepts, debugging problems, and optimizing performance (Joukov et al., 2006). The concept of profiling can easily be confused with monitoring and tracing. Let us first disambiguate our use of these terms. Monitoring is ongoing and can be used to trigger alerts such as when a monitored resource falls above or below a trigger level (Thor, 2012). While tracing is tracking the flow of a request or data through a network, program, or disturbed system (Kaldor et al., 2017). Whereas profiling is usually done on a particular program to see which code is using the most resources (Graham et al., 1982).

What profiling do people do today in DNN training? Aside from performance engineers and researchers who develop custom tools (Hu et al., 2022; Zhu et al., 2020; Gleeson et al., 2021; Yu et al., 2021) or use expert-level profiling tools (NVIDIA, 2018b; Intel, 2020; Google, 2022), there are two types of ML developers: (1) those who are not particularly interested in system efficiency because it is secondary to their goal of DNN model accuracy, and (2) those who see the value but find it demands too much systems knowledge or engineering effort to profile (Yu et al., 2020). Both types of developers commonly push the limits of their hardware and subsequently ask seemingly simple questions like, "*Why is training slower than expected?*" and "*What is the slowest part of training?*". Finding answers to these questions requires *runtime profiling*.

2.2 Background on Runtime Profiling

Runtime is an important performance metric because *time* is an absolute metric that sums up program behavior enabling users to identify slow sections and where to target optimizations. The naïve approach is to manually insert print statements with timestamps, or vendor-specific annotations such as NVTX (NVIDIA,

Figure 2. Runtime profilers record a timeline of events. Here we see a portion of a single training iteration of an RNN-T model visualized with the Perfetto trace viewer. We have drawn red annotations to show a zoomed-in portion of events.

2020b), `pytorch.record_function` (Facebook, 2021), or `jax.named_scope` (Google, 2022), but this has poor code coverage and introduces significant developer effort. Alternatively, *tracing profilers* instrument the program to record every function call during program execution and produce a detailed event trace of exactly what operations took place and when (Evans, 2017). Figure 2 shows an example runtime event trace produced by the PyTorch tracing profiler (Facebook, 2022). The content of tracing profiler data produced by PyTorch consists of an array of events each with a timestamp, duration, operation name, and other data such as function arguments (e.g., input data dimensions).

2.3 Why is Interpreting Runtime Data Challenging?

Despite the importance of using runtime profiling tools to find optimization opportunities, we observe several *challenges when interpreting runtime data*, which impede adoption by non-experts. Runtime data is often complex, multi-dimensional, and noisy, and therefore it becomes difficult for humans to see patterns, trends, and outliers by looking at it in its raw form. Figure 2 illustrates these challenges and is representative of existing trace viewers including TensorBoard (Google, 2015), Nsight Systems (NVIDIA, 2018b), and Perfetto (Google, 2022).

Challenge 1: Information Overload
Existing visualizations fail to convey a high degree of useful information due to *information overload* caused by the large number of operations typically found in DNN training. For example, we find that a single training iteration on 4 GPUs for an RNN-T model produces almost one million operations as seen in Figure 2. This vast quantity puts a large burden on users, requiring excessive effort to navigate and interpret large runtime traces, and makes it non-trivial to summarize or hide some details.

Challenge 2: Disparate Granularities
Disparate granularities refers to operations whose runtimes span significantly different timescales. For example, we find that the high-level backward pass of an RNN-T model (runtime 1.2 sec) is comprised of 363,095 low-level operations (avg. runtime 42 μs). Existing visualizations are not easily interpretable because they display all operations at all granularities on a single timeline view. A single view of very short and very long operations is hard to see, navigate, and compare. Zooming in and out between granularities is tedious and only clearly displays one granularity at a time.

Challenge 3: Discontiguous Execution
Relationships between operations may be hard to follow due to *discontiguous execution* patterns in which related operations are not next to each other, such as in asynchronous and distributed processing. For example, asynchronous GPU kernels can have their CPU and GPU portions very temporally distant, and the two operations will always be found in different timelines, one for a CPU thread, and one for a GPU stream. Furthermore, operations can be parallelized across multiple GPUs or servers. These challenges have yet to be addressed by existing works and result in *fewer ML developers performing or benefiting from profiling*.

2.4 How to Attract Developers Who Do Not Profile?

To attract developers to profiling we can draw inspiration from the qualities of past success stories of getting people to do something new and complex. For example, how did the iPhone get millions of people to use *smart*phones? The iPhone was radically *simpler to navigate* by touch, it brought together many utilities, and it was colorful, fast, and fun to use. From this we draw our first design goal:

Goal 1: Easy to Navigate for everyone in a way that is efficient and easy to remember when returning to the tool.

Similarly, consider the Linux monitoring tool `htop` (Muhammad, 2004), it improved upon `top` (LeFebvre, 1984) by (again) improving interactivity, adding meaningful *color* and *graphs*, and reducing visual *noise* by displaying less text. Displaying less text reduces the cognitive burden on users, is *inviting to non-experts*, and supports *finding insights* more easily:

Goal 2: Reduce Noise so that insights are easier to find, data is easier to navigate, and novices are not overwhelmed.

We also observe that the iPhone and `htop` did not do anything radically new. They repackaged information and applications that users were already familiar with:

Goal 3: Familiar to everyone and only branch out when the user wishes to expand their knowledge.

In this work, we set out to apply these successful qualities which embody usability heuristics (Nielsen, 2005) to the DNN profiling problem: our strategy to get more people to measure is to make a profiler that is *insightful, familiar, and easy to navigate*.

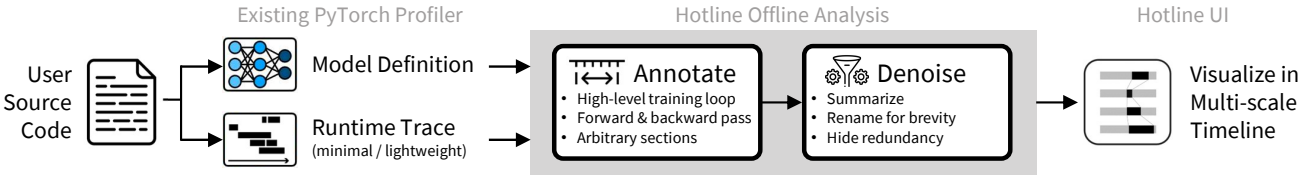


Figure 3. HOTLINE system overview. A DNN model definition is used to guide automatic annotations of a DNN training runtime trace. After applying denoising heuristics to simplify the runtime annotations they are displayed in a multi-scale timeline for users to investigate.

```python
import torch.profiler
import torchvision.models
import hotline

model = torchvision.models.resnet50()
# ... other setup...

with torch.profiler.profile(
  on_trace_ready=hotline.analyze(model)):
    # ... training loop ...
```

```
$ ./my_training_script.py  # then open 'results.json' with UI
```

Figure 4. How to instrument PyTorch code with HOTLINE.

3 HOTLINE: OVERVIEW

When investigating optimization opportunities, interpreting raw DNN training runtime traces can be challenging for ML researchers and developers. To greatly simplify the effort required we propose HOTLINE[2], a novel profiler designed specifically for *runtime bottleneck identification*.

Figure 3 illustrates the workflow of HOTLINE. HOTLINE parses the user's DNN model definition including any user-defined variable names for operations, or groups of operations such as layers, so that HOTLINE can summarize runtime traces at any of these granularities (*Goal 2: Reduce Noise*) and with names familiar to the user (*Goal 3: Familiar*). A runtime trace of a single representative iteration of DNN training is collected after a user-chosen number of warmup iterations. Then HOTLINE performs offline annotation of the trace with DNN concepts that most ML developers are familiar with, i.e. the DNN training loop and model architecture (Section 4). Next, these annotations are summarized where possible to simplify or eliminate less-interesting parts (i.e. *"noise"*) of the runtime trace (Appendix C.2). Finally, HOTLINE displays the runtimes of the resulting annotations in a multi-scale timeline (Section 5), a technique well suited to navigating complex timelines (*Goal 1: Easy to Navigate*).

We choose PyTorch (Paszke et al., 2019) as our prototyping DL framework as it is one of the most widely used frameworks. HOTLINE is designed in a general way that supports a wide variety of models (e.g., ResNet, RNN-T, Transformer) without modifying PyTorch or relying on hardware or software-specific knowledge. This means that HOTLINE introduces no new overhead on timing measurements, does not create vendor lock-in, and is easy to maintain and adapt to new applications. The same ideas can be implemented on top of other DL frameworks including TensorFlow (Abadi et al., 2016), JAX (Frostig et al., 2018), and MXNet (Chen et al., 2015) because they produce runtime traces in the same standard Trace Event Format (Sinclair, 2016). Also, HOTLINE is carefully designed to accommodate "novice" developers. It can be used seamlessly with PyTorch training scripts, and only requires adding one argument when using PyTorch's built-in profiler. To illustrate, Figure 4 shows how to enable HOTLINE for ResNet-50.

[2]The name HOTLINE name is a concatenation of the word "HOT" which is the colormap we use to visually emphasize runtime bottlenecks, and "LINE" which is taken from the multi-scale time*line* design that we introduce to ML system profiling.

4 HOTLINE: AUTOMATIC ANNOTATION

An *annotation* is a group of operations that share a common purpose. Annotations have a runtime that spans from the first start time to the last stop time of the operations in that group. Annotations can cover various granularities of a program. For example, in DNN training, annotations may include the forward pass, layer 1, or an individual convolution. To the best of our knowledge, HOTLINE is the first profiler to provide annotations of DNN training concepts across these granularities.

Annotations are fewer in number than raw events which makes them easier to navigate (*Goal 1*) and less noisy (*Goal 2*). Furthermore, annotations of DNN concepts allow ML developers to begin their profiling investigation at a level of granularity they are familiar with (*Goal 3*). HOTLINE processes a standard runtime trace data format (Sinclair, 2016) to generate three types of annotations automatically: the steps of the DNN training loop (Section 4.2), the architecture of the DNN model (Section 4.3), and arbitrary subsections of the DNN training loop and model (Section 4.5).

4.1 Key Observations to Enable Automatic Annotation

In this section, we highlight the key observations behind HOTLINE's automatic annotation design so that it does not incur overhead on timing measurements and can utilize PyTorch's existing profiling instrumentation. Our major observations are based on the characteristics of the workloads

of most developers. While some DL researchers experiment with novel approaches to DNN training, the majority of developers in supervised DL use very similar training loops (Ruder, 2016) and canonical model architectures (He et al., 2016; Iandola et al., 2016).

In our evaluation of these common workloads, **we observe that DNN operations in runtime traces are serialized in an order that is static, repeatable, and predictable.** We believe this observation holds for three reasons. Firstly, most DNN models do not exhibit data-dependent control flow so that they can be effectively accelerated by compilers (Chen et al., 2018b; Zheng et al., 2022) and specialized hardware (NVIDIA, 2018a; Barham et al., 2022). Secondly, to achieve high accuracy, DL optimization algorithms (Dozat, 2016; Kingma & Ba, 2014) are most commonly used with synchronous model updates and this effectively serializes the training loop (Ruder, 2016). Thirdly, the most common parallelization technique in DL is data parallelism (Chen et al., 2018a), which performs serialized single instructions on multiple data (SIMD) and the order of instructions remains serialized when data is distributed across systems.

Additionally, **we observe that descriptive, recognizable operation names exist in runtime traces** such as "conv" or "gemm" and thus can be matched to operation names in DNN model definitions to build richer annotations after trace collection. While DL systems that produce ambiguous operation names are emerging, as seen in JAX's aggressive kernel fusion (Frostig et al., 2018), this is not a new problem. Solutions have been proposed (Gregg, 2017a), and descriptive operation names will continue to be vital to effective software development and debugging.

By relying on these observations, HOTLINE is trading off support for several emerging applications including data-dependent control flow, aggressive kernel fusion or ambiguous operation names, and less common forms of parallelism. However, by targeting the majority of DNN training use cases, including cases of limited kernel fusion (see Appendix C.1), the following unique advantages are obtained. HOTLINE's offline analysis can generate and enrich annotations (e.g., denoise) with no impact on timing measurements and minimal instrumentation.

4.2 DNN Training Loop Annotation

The typical DNN training loop consists of the following high-level stages: (i) data loading, (ii) the forward pass, (iii) calculating loss, (iv) the backward pass, and (v) updating the optimizer. It can be challenging for ML developers to manually identify any given stage in a runtime trace because existing trace viewers do not highlight them amongst the many thousands of events displayed. Furthermore, traces may not even include events demarcating these stages, as is the case for the forward and backward passes when PyTorch's low profiling overhead mode is enabled[3].

To solve this problem, we build on our observations that the execution behavior of typical DNN training loops is serial and detectable. Our annotation algorithm searches the event trace for recognizable names (e.g.,"dataload", "forward", "loss"), and creates an annotation for that stage including any nested events. Should any one stage not be detected, an annotation can be made that spans in-between detected stages or spans to the start or end of the trace. This approach is generic enough to continue working if PyTorch changes its trace output and flexible enough to adapt to differences in traces produced by different DL frameworks.

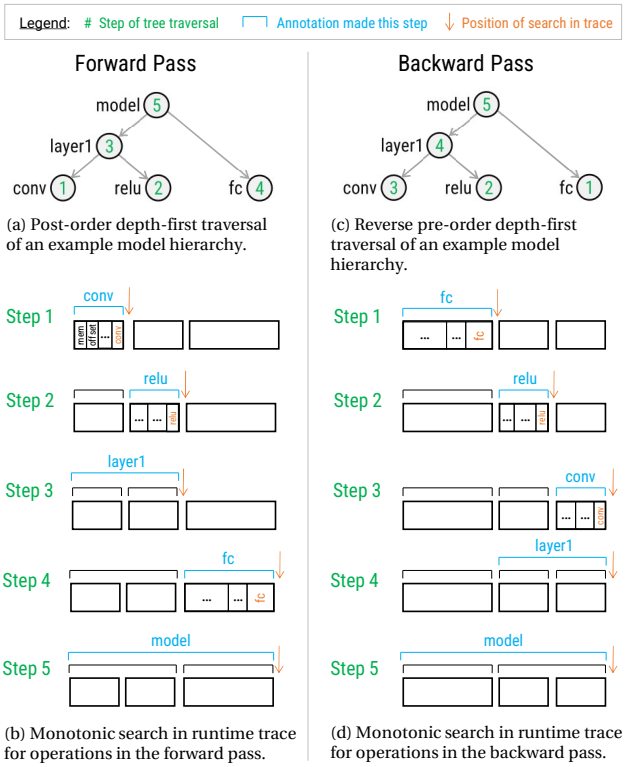
4.3 DNN Model Annotation

Reconciling operations defined in DNN models to operations in the DNN training traces is non-trivial and tedious to do manually so we have developed an algorithm to automatically annotate the entire hierarchical structure of DNN models. Figure 5 illustrates the operation of our annotation algorithm as it traverses a graph representation of the user-defined DNN model obtained from PyTorch. Leafs in the graph represent DNN operations and branch nodes represent groups of operations. For leaf nodes, the operation name is searched for in a single pass by stepping monotonically through time in the runtime trace and performing substring matching on the names of CPU operations and connected GPU operations whenever a kernel is launched. When a match is found, an annotation is generated that includes the operations in the runtime trace starting after the last annotation and ending after the matched operation. When multiple consecutive DNN operations have the same name, a single annotation is produced to avoid incorrectly dividing up the trace in this situation of ambiguity. For branch nodes, all the runtime operations of descendent nodes in the graph are included in a new annotation.

The forward and backward passes are detected in runtime traces using the same algorithm, except the order of operations is reversed for the backward pass. For the forward pass, a post-order depth-first tree traversal is used to iterate over the hierarchical DNN model. For the backward pass, a reverse pre-order depth-first tree traversal is used. In a data-parallel training setup, each GPU will execute the same sequence of operations allowing us to repeat the same automatic annotation algorithm for each GPU.

A small number (n=14) of special rules were hard-coded to assist in name matching to support automatic DNN annota-

[3]By low overhead mode of PyTorch's profiler, we mean with the `stacks` option disabled. This reduces the number of events in the trace (by up to 3× for ResNet) and disables the inclusion of a stack trace in each event. HOTLINE works better in this mode because it doesn't rely on stack traces so it can process events more quickly and timing measurements will be more accurate.



(a) Post-order depth-first traversal of an example model hierarchy.

(c) Reverse pre-order depth-first traversal of an example model hierarchy.

(b) Monotonic search in runtime trace for operations in the forward pass.

(d) Monotonic search in runtime trace for operations in the backward pass.

Figure 5. HOTLINE's offline DNN model annotation algorithm traverses the model definition and matches operation names to event names in the runtime trace.

tion. For example, one rule looks for the name "`cudnn::bn`" in the runtime trace when the searched model operation name is "`batchnorm`". Another example is matching "`linear`" to any of "`addmm`", "`mmbackward`", or "`sgemm`". The effort of developing these rules was extremely small. We believe we have covered the majority of cases, and that future addition will be trivial and not overly burdensome.

4.4 Annotation Accuracy

In practice, we were surprised by the robustness of our prototype tool for automatic DNN annotation. We attribute this robustness to our observations (Section 4.1) holding true and to the synthetic test cases that we wrote to test functionality and edge cases when developing our algorithms. To help prove the correctness of HOTLINE's automatic annotation method we developed an accuracy evaluation system. For ResNet, RNN, and Transformer models we systematically compared our automatic annotations to hand-implemented annotations of the training loop and full model architecture. We found 97-99% accuracy of matching raw events into the correct ML operation. We did find ambiguity at the boundaries of annotations, but only for insignificant, short-lived operations (e.g., "Memset") and had no appreciable impact on insights. The accuracy we observed is an improvement over a human guessing which low-level event belongs to which high-level operation or a human expending effort to manually insert annotations into source code and likely making a few mistakes or omissions.

4.5 Arbitrary Annotation

Thus far we showed how HOTLINE annotates DNN concepts. However, there remain interesting regions of runtime traces that are not defined in DNN models or training loops. For example, interesting sub-steps arise when using more than one GPU because communicating between GPUs becomes necessary (e.g., replicating a model's learned parameters across GPUs). ML developers may not be aware of these stages or that they can be bottlenecks. To help users spot important sections of DNN training without overwhelming users with a tremendous number of raw runtime events, HOTLINE contains a heuristic-based algorithm to detect and annotate arbitrary sections without the need for user input or software or hardware-specific knowledge.

HOTLINE will demarcate an annotation in the runtime trace when (1) a series of consecutive operations are all tiny (i.e. their runtime is below a configurable threshold relative to its parent operation; 5% is our default), or (2) a series of consecutive operations have the same name. This procedure is applied recursively because we find these patterns occur at multiple timescales within nested operations. This enables HOTLINE to display far fewer annotations than existing trace viewers which display all raw runtime events. Therefore it becomes easier for developers to interpret runtime traces, learn about the inner workings of DNN training, and identify unexpected bottlenecks. In Appendix B.3 we show how this feature reveals bottlenecks in data-parallel training.

5 HOTLINE: MULTI-SCALE TIMELINE

Visualization is an underappreciated part of systems research. When no automated optimization solution exists or is trusted, systems visualization assists in making subtle determinations about what is good or bad performance. Visualization (i.e. plots, charts, and diagrams) is used throughout the lifecycle of systems research including discovering, validating, and communicating optimization opportunities (Guo et al., 2015; Adhianto et al., 2010; Li et al., 2020).

Existing timeline visualizations show the entire raw trace in one large timeline and this tends to overwhelm ML developers due to information overload, disparate granularities, and discontiguous execution (*Challenges 1-3*). HOTLINE addresses these challenges by introducing the Multi-Scale Timeline (Richter et al., 1999), a promising visualization technique that is well suited for navigating and interpreting the complex timelines found in DNN training and which may have broad utility in the field of systems profiling.

We have developed an interactive web UI, seen in Figure 1b, that is generic enough to display any hierarchical set of

Table 1. Size and analysis time of the workloads evaluated on 2080Ti GPU.

Model	Dataset	# of GPUs	Total # of Operations Per Training Iter.	Raw Trace File Size	Iteration Time	Iteration Time with PyTorch Profiling	HOTLINE Analysis Time	HOTLINE Generated Annotations
ResNet-50	ImageNet (batch=96)	1	7,406	3.8 MB	0.354 sec ±0.0%	0.356 sec, 1.01× slower	4.4 sec	324
ResNet-50	ImageNet (batch=384)	4	23,384	10.8 MB	0.506 sec ±0.5%	0.512 sec, 1.01× slower	8.5 sec	1,073
Transformer	WMT (batch=8)	1	22,755	9.7 MB	0.355 sec ±0.1%	0.362 sec, 1.02× slower	7.3 sec	180
Transformer	WMT (batch=16)	4	72,005	27.8 MB	1.68 sec ±0.1%	1.72 sec, 1.02× slower	16 sec	636
RNN-T	LibriSpeech (batch=32)	4	941,489	266 MB	1.48 sec ±2%	2.12 sec, 1.43× slower	3 m 44 sec	602

timeline events in a multi-scale timeline. The multi-scale timeline allows users to see multiple timescales in one view, greatly simplifying the embedded complexity found in software stacks. HOTLINE's visualization uses both color and size of annotations to make runtime bottlenecks pop out (*Goal 2: Noise Reduction*). We initially show the runtime of only the highest level operations in the DNN training loop (*Goal 3: Familiar*). Users can interactively drill down by clicking on an operation to view the runtime of the sub-operations contained within any part of the training loop.

Existing tools use pan and zoom interaction to navigate, which is overly burdensome on users because it requires switching back and forth between two types of interactions (i.e. pan and zoom separately) and make multiple adjustments to view a region of interest. Instead, HOTLINE uses a single click to perform a technique called Semantic Zooming (Aigner, 2014) to reveal details at a finer granularity and fit them on screen. Semantic Zooming is faster, simpler, and more exact for users (*Goal 1: Easy to Navigate*).

Figure 1a shows that the existing trace visualizations found in TensorBoard can be characterized by an excessively large number of operations and most operations names are hidden or cut off. By contrast, HOTLINE's multi-scale timeline seen in Figure 1b, shows HOTLINE's annotations without cluttering (*Goal 2: Noise Reduction*) and importantly, in the context of what happens before, after, and at multiple levels of granularity. To communicate this much information without HOTLINE, users of TensorBoard must compile multiple screenshots at different time scales or record a video of zooming in.

6 EVALUATION

Next, we evaluate the utility of HOTLINE's automatic annotation and multi-scale timeline visualization. Section 6.1 gives a summary of profiler overhead and annotation analysis time for our evaluated workloads. Section 6.2 presents in-depth case studies illustrating HOTLINE's unique advantages when used to investigate time-use bottlenecks. Finally, Section 6.3 reports on software developers' qualitative impressions of HOTLINE's design and shows how rapidly common performance investigation tasks can be performed.

We evaluated HOTLINE with three diverse real-world DNN models: ResNet-50 (He et al., 2016), RNN-T (MLCommons, 2021), and Transformer (Vaswani et al., 2017) with standard batch sizes and optimizers (see Appendix B.5 for details). We use PyTorch's `DataParallel` mode for training with multiple GPUs. We performed our experiments on a server with 4 2080Ti GPUs (NVIDIA, 2018a), an AMD EPYC 7601 CPU (AMD, 2017), and 128 GB of RAM.

6.1 Size and Analysis Time of Workloads

Before using HOTLINE's visualization to find time-use bottlenecks, a user must first collect a runtime trace of DNN training and analyze it using HOTLINE to generate and denoise annotations. Table 1 shows that the number of raw operations collected by PyTorch in a single iteration of training can be as high as 941,489 or 266 MB in `JSON` format for the RNN-T model, an inefficient model comprised of many, short, sequential operations. HOTLINE summarizes this trace into 602 annotations represented on disk in a 524 KB `JSON` results file. We found that PyTorch's built-in profiler can slow down iteration time by as much as 43% for RNN-T, but for more efficient ResNet and Transformer models by 1-2%. This is a behavior of PyTorch's profiler regardless of whether HOTLINE is used. The analysis time required by HOTLINE to process these traces was under a minute for the efficient models, but for RNN-T, a large amount of trace data needs to be processed by our Python analysis code, so several minutes are required. We believe this can be sped up in future work and that it is a small amount of time compared to the time required to observe convergence of model accuracy in real-world DNN training.

6.2 Bottleneck Identification Case Studies

We illustrate the usefulness of HOTLINE's annotations and multi-scale timeline visualization through in-depth case

studies of ResNet-50 and RNN-T models and find opportunities to speed up training by 40% and 51% respectively. The process by which we conduct our case studies is as follows. First, we use HOTLINE's multi-scale visualization to drill down on the annotations that have the longest runtime starting at the highest level granularity (the DNN training loop) down to the lowest granularity (the individual GPU kernels). Then we describe what insights can be found and how HOTLINE's unique advantages make DNN performance investigation faster and easier.

6.2.1 *Case Study: ResNet-50 on a Single GPU*

The first case study looks at the ResNet-50 model on 1 GPU, which is a common benchmark in the field of DL (Reddi et al., 2020; Mattson et al., 2020). Appendix B.1 shows HOTLINE's UI after a user has drilled down into the slowest parts of a single ResNet-50 training iteration. The **bottleneck is the most striking visual aspect** which is the large and bright red operation marked `backward` on the `GPU0` track. Compare this experience to the one seen in Appendix D of TensorBoard, where metrics about the DNN training loop are not presented in the summary view or any view, leaving users to guess about the bottleneck.

The second most striking visual aspect seen in this visualization of ResNet-50 is the large, orange operation at the lowest level of detail named `wgrad_alg0_engine`. HOTLINE can be an **educational experience** because the user can learn what kernels are used by DL libraries and learn that "`wgrad`" is an algorithm used for convolution (Coppersmith & Winograd, 1982). When observing the middle three levels of the multi-scale timeline, titled "`2. backward`", "`3. layer2`", and "`4. Bottleneck-0`", one may note the less dramatic colors consisting of shades of light yellow. A DNN developer might be pleased to find this because it means they have balanced runtime similarly across layers and groups of operations. This is an example of how HOTLINE helps developers **understand how their choices about DNN model architecture and input data size effects translate into actual runtime**.

A user can draw on these insights to correctly conclude that the biggest optimization opportunity is to speed up the backward pass GPU kernels. By looking at HOTLINE's runtime breakdown of the training loop, it can be deduced that the maximum possible speedup of optimization to the GPU backward pass would be able to reduce the current 80ms runtime to 30ms, at which point the CPU portion of the backward pass would prevent further speedup. If there was a way to speed up the GPU kernels of the forward and backward pass, such as with a faster GPU, end-to-end training could be sped up by as much as 40%. We tested a faster Nvidia 3090 GPU and found the forward pass had within 1% number of kernels and ran them 42% faster.

6.2.2 *Case Study: RNN-T on 4 GPUs*

In our second case study, we look at the RNN-T model (Graves, 2012) that is planned to be part of future MLCommons benchmarks (Reddi et al., 2020; MLCommons, 2022). Appendix B.2 shows HOTLINE after a user has drilled down into the slowest parts of a single RNN-T training iteration. Here we immediately see a major bottleneck at all levels of the multi-scale timeline, which quickly points us to a low-level cause. A benefit of using a multi-scale timeline is that **one picture explains the low-level cause of slowdown**, that 73% of the time is spent on "`cudaLaunchKernel`" and "`sgemm`" kernels, **as well as higher levels of context for this bottleneck**, that these operations take place within LSTMs, within RNNs, within the backward pass of the DNN training loop. Contrast this to TensorBoard, seen in Appendix D, where the "Kernel View" would say that 46% of GPU kernel runtime is due to "`sgemm`" kernels without any of the aforementioned context and without the ability to click on the kernel to investigate the problem deeper.

Next, we describe how HOTLINE can further assist with rapid low-level investigations by using HOTLINE's "Open with Perfetto" button to **isolate interesting portions of the runtime trace and quickly switch to a more detailed profiling tool**. Upon inspection of the detailed runtime trace for an LSTM operation, we see many kernel launches on the CPU, but large idle times on the GPU (Appendix B.2). We found this problem much faster than we would have using Perfetto alone for two reasons. First, navigating to the bottleneck using HOTLINE's semantic zoom interaction is much faster than Perfetto's pan *and* zoom interaction. Secondly, HOTLINE's annotation isolated a single LSTM operation within the raw trace, significantly reducing information overload in Perfetto.

From this, a user can conclude that the biggest optimization opportunity is to increase GPU utilization by reducing the kernel launch overhead. Using HOTLINE to view only the backward pass in Perfetto, we calculated that the GPU is only executing kernels 30% of the time. Therefore, we deduced that if an optimization could eliminate delays between kernel executions, the maximum possible speed-up of the backward pass would be 840ms and this would speed up end-to-end training by 51%. We found with a newer Nvidia 3090 GPU that CUTLASS was employed in the backward RNN blocks which led to 26% fewer kernels launched and training was 32% faster, partially resolving this bottleneck.

6.3 User Evaluation: Usability Study

To evaluate if software developers find HOTLINE useful we conducted a user study with computer science students recruited by convenience sampling who identified as ML algorithm developers (n=4), ML systems developers (n=13) or non-ML systems developers (n=5). Each participant

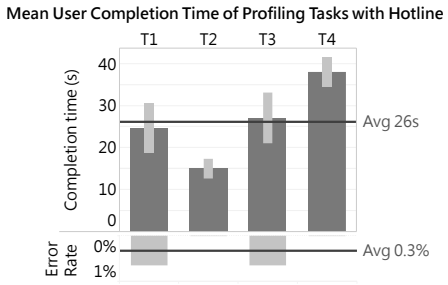


Figure 6. A summary of completion times and error rates for users performing ML profiling tasks using a HOTLINE visualization mock-up. Error bars depict standard error.

was compensated with a $20 gift card. Participants evaluated a non-interactive design mockup of HOTLINE that is very close to what was implemented. Evaluating a mockup allowed us to guide and validate features before implementation. After a 15-minute walkthrough of HOTLINE's features with a realistic ResNet-50 example, we asked skill-testing multiple-choice questions to evaluate how well users could perform the following profiling tasks without help:

1. What is the slowest operation on the CPU?
2. What is the fastest operation on the GPU?
3. Is runtime of the backward pass limited by GPU or CPU?
4. Which operation in "Layer 3" is best to try to speed up?

The results in Figure 6 show that, on average, each task could be completed within roughly 30 seconds and 97% of tasks were completed correctly. All of the participants completed the tasks independently. Because we evaluated on a mockup it is hard to draw quantitative conclusions. The more important qualitative finding was that we observed that HOTLINE's design was well suited to the user's task and mental model of runtime profiling.

Next, we asked participants to rate HOTLINE's features on a five-point Likert scale (Likert, 1932) for each question ($1 \rightarrow$ "*Greatly confuses me*", $5 \rightarrow$ "*Greatly helps me*"). Figure 7 summarizes our results, that most participants found all the features to be greatly helpful (mean 4.31, mode 5). Then we asked for the user's qualitative impressions of HOTLINE more generally. 82% (n=14/17) of participants said that the visualization "*takes less time to find insights*", and 59% (n=10/17) of participants said that the visualization "*provides more insights*", and 65% (n=11/17) were "*more likely to use the new visualization than existing visualizations.*"

We feel these preliminary results are encouraging given that, with just a brief walkthrough, participants of various backgrounds and education levels were able to use the visualization to complete profiling tasks and reported positive impressions. While user studies are some of the time considered weak evidence (Greenberg & Buxton, 2008), they are essential tools to understand user's attitude towards and usage of technology. It has been argued that even a small number of participants covers the majority of insights (Nielsen, 2000). Despite the limited sample size, the study allowed us to capture the user's thought processes, workflows, and pain points in usage. We feel the results are promising, as findings indicate that novices can use HOTLINE to make it easier to identify performance bottlenecks.

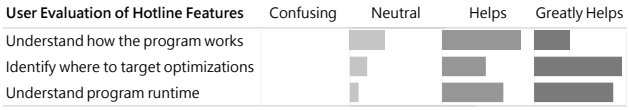


Figure 7. A summary of how useful participants found HOTLINE on a variety of ML profiling tasks.

7 RELATED WORK

HOTLINE builds upon bodies of prior work including (1) developer-friendly ML profiling tools (Yu et al., 2020; Google, 2015) by providing an intuitive runtime breakdown of ML training concepts that most developers are familiar with; (2) analysis of ML training traces which has been applied to predict speedups of known optimizations (Hu et al., 2022; Zhu et al., 2020), instead HOTLINE assists developers in identifying unknown latency bottlenecks; (3) novel visualizations of execution event traces which have been presented in the form of a tree-map (Bockisch et al., 2015), a circular view (Cornelissen et al., 2007), and a trace similarity view (Trümper et al., 2013) but which are different from HOTLINE's presentation of multiple timescales in one view; and (4) event trace simplification techniques such as pruning (Bohnet et al., 2009), folding (Weber et al., 2015), and annotating at the level of program loops (Mohror & Karavanic, 2012) have been demonstrated but these differ from HOTLINE which leverages domain-specific knowledge of ML training to generate simplifying trace annotations.

HOTLINE is intended to be complimentary to (1) existing profilers provided by DL frameworks (Facebook, 2022; Google, 2022b; Chen et al., 2015) because HOTLINE generates additional trace annotations without affecting runtime measurements; (2) existing trace viewers (Google, 2022; NVIDIA, 2018b; Google, 2012a) because HOTLINE provides quick navigation, easy interpretation, and can isolate a less-overwhelming portion of the original trace to view in an existing viewer. HOTLINE has advantages over (1) sampling profilers such as perf (Linux, 2009), Nsight Compute (NVIDIA, 2020a), or VTune (Intel, 2020) because HOTLINE's tracing based approach has better code coverage and timing accuracy; (2) call graph viewers such as pprof (Google, 2022a), Callgrind (Weidendorfer, 2022), or Flame Graph (Gregg, 2017b) which do not present a time ordered view and therefore lacks important context. Compared to expert-level ML profiling tools (Li et al., 2020; Gleeson et al., 2021; Dakkak et al., 2020), HOTLINE has a unique focus on ease of interpretation and is more accessible to a wider variety of skill levels.

8 FUTURE WORK

We have presented a promising approach for visualizing runtime profiler data to help DNN developers find and understand bottlenecks. However, there remain opportunities to extend work in this area.

***Why* is it Slow?**
In addition to identifying bottlenecks, HOTLINE could do more to help users understand bottlenecks. For example, HOTLINE could display resource utilization timelines, link to source code and kernel instructions, or automatically trigger more detailed profiling tools for the most time-consuming operations, for example with Nsight Compute (NVIDIA, 2020a). Aggregating and displaying hardware-level metrics within HOTLINE would help users understand why operations are slow.

Generalizing Beyond PyTorch
Implementing support for multi-scale annotation and visualization of runtime traces in other DNN frameworks such as JAX would ease the task of profiling for more users. Firstly, it would be important to support the ONNX format (Facebook, 2017) for model architecture information rather than only extracting information from PyTorch Modules. ONNX has the advantage of including information about data dependencies and memory operations. However, ONNX may lose important hierarchical information about the structuring or grouping of operations such as within layers or other high-level concepts, and the developer-friendly names of these structures from source code may be lost. Secondly, a framework like JAX which makes use of aggressive kernel fusion may need additional ONNX metadata or a symbol table to correctly associate high-level operations to their respective low-level kernels. Support for CUDA graphs or DNN models with conditional runtime control flow may also require special handling.

Support for Diffs
Making it possible to compare runtimes and behavior between different ML frameworks (Jax, TensorFlow, PyTorch, MXNet), different versions of a model, or different low-level implementations would be incredibly valuable. This is important because users should be able to compare to a baseline to determine if performance has gotten better, worse, or stayed the same. Shades of blue could represent the relative amount of speedup and shades of red for slowdowns. Snider (2022) shows a concept of what this could look like.

Visualization Improvements
To keep the visualization uncluttered, HOTLINE does not depict data dependencies or control flow, but this helpful information could be illustrated using arrows. Additionally, to give a more compact and sophisticated view of runtime behavior it may be beneficial to detect repeating patterns in event traces and provide visual summaries of these.

Study More Models, Hardware, and Applications
In future work, it would be interesting to experiment by fixing the DNN model and varying the accelerator (Nvidia Jetson, Intel CPU, Google TPU) or varying the software setup (CUDA or PyTorch version) to identify interesting runtime bottlenecks that lead to poor performance. Recently we evaluated more models (DLRM, GNN, and Stable-Diffusion) and additional hardware (RTX 3090). We have published these results in a demonstration website of HOTLINE which is linked on our GitHub `https://github.com/UofT-EcoSystem/hotline`. These results include Stable-Diffusion inference which shows that HOTLINE works for DNN inference and training. Inference works in the same way as the forward pass of training. HOTLINE is currently designed for ML applications that use multiple CPU processes/threads and multiple GPU devices/streams. More investigation is needed for applications with deep pipelines and applications that use more types of resources such as storage devices, remote servers, cameras, and other peripherals. We believe HOTLINE's multi-scale timeline ideas may be beneficial for profiling other computer systems such as internet browsers, distributed high-performance computing, and real-time robotics.

9 CONCLUSION

In this work, we explore a simultaneously simple and complex question in DNN training, *"what is slow?"*. To greatly simplify the effort required when investigating ML system runtime traces, we propose HOTLINE. HOTLINE derives a detailed DNN training runtime breakdown from profiler traces in seconds. Previously, such a runtime breakdown would take hours or days to manually investigate by a novice because existing traces contain cryptic names, an overwhelming number of low-level operations, and lack annotations of high-level ML concepts such as stage of training or position in the DNN model. We believe we can change ML profiling from being an afterthought or burden to being a concern as important as hyperparameter tuning; because profiling can have as much impact on the success of DL and can be an enjoyable learning experience for developers too.

ACKNOWLEDGEMENTS

This project was supported by the Canada Foundation for Innovation JELF grant, NSERC Discovery grant, AWS Machine Learning Research Award (MLRA), Facebook Faculty Research Award, Google Scholar Research Award, and VMware Early Career Faculty Grant. We would like to thank the MLSYS reviewers for their valuable feedback and the artifact reviewers for reproducing our experiments. We also thank Vasudev Sharma, Fabian Ulmer, James Gleeson, Mickey Gabel, Nandita Vijaykumar, Gabriela Morgenshtern, Nicole Sultanum, Adamo Young, Geoffrey Yu,

Yaoyao Ding, and Shang Wang for their support, inspiration, and constructive feedback during the development of this work. We also wish to thank all the participants of the user study who helped motivate and validate this work.

REFERENCES

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., and others. TensorFlow: a system for Large-Scale machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, pp. 265–283, 2016.

Adhianto, L., Banerjee, S., Fagan, M., Krentel, M., Marin, G., Mellor-Crummey, J., and Tallent, N. R. HPCToolkit: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience*, 22(6):685–701, 2010. Publisher: Wiley Online Library.

Aigner, W. Semantic Zoom, 2014. URL https://infovis-wiki.net/wiki/Semantic_Zoom.

AMD. EPYC™ 7601 CPU, 2017. URL https://www.amd.com/en/products/cpu/amd-epyc-7601.

Amodei, D. and Hernandez, D. AI and Compute, May 2018. URL https://openai.com/blog/ai-and-compute/.

Andoorveedu, M., Zhu, Z., Zheng, B., and Pekhimenko, G. Tempo: Accelerating transformer-based model training through memory footprint reduction. In Koyejo, S., Mohamed, S., Agarwal, A., Belgrave, D., Cho, K., and Oh, A. (eds.), *Advances in Neural Information Processing Systems*, volume 35, pp. 12267–12282. Curran Associates, Inc., 2022. URL https://proceedings.neurips.cc/paper_files/paper/2022/file/4fc81f4cd2715d995018e0799262176b-Paper-Conference.pdf.

Barham, P., Chowdhery, A., Dean, J., Ghemawat, S., Hand, S., Hurt, D., Isard, M., Lim, H., Pang, R., Roy, S., and others. Pathways: Asynchronous distributed dataflow for ML. *Proceedings of Machine Learning and Systems*, 4: 430–449, 2022.

Bianco, S., Cadene, R., Celona, L., and Napoletano, P. Benchmark analysis of representative deep neural network architectures. *IEEE access*, 6:64270–64277, 2018. Publisher: IEEE.

Bockisch, C., van 't Riet, M., Yin, H., Aksit, M., Lin, Z., Chen, Y., and Zhao, J. Trace-Based Debugging for Advanced-Dispatching Programming Languages. In *Proceedings of the 10th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*, 2015. ISBN 978-1-4503-3657-4. doi: 10.1145/2843915.2843922. URL https://doi.org/10.1145/2843915.2843922.

Bohnet, J., Koeleman, M., and Doellner, J. Visualizing massively pruned execution traces to facilitate trace exploration. In *2009 5th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, pp. 57–64, 2009. doi: 10.1109/VISSOF.2009.5336416.

Chen, C.-C., Yang, C.-L., and Cheng, H.-Y. Efficient and robust parallel DNN training through model parallelism on multi-GPU platform. *arXiv preprint arXiv:1809.02839*, 2018a.

Chen, T., Li, M., Li, Y., Lin, M., Wang, N., Wang, M., Xiao, T., Xu, B., Zhang, C., and Zhang, Z. MXNET: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

Chen, T., Moreau, T., Jiang, Z., Zheng, L., Yan, E., Shen, H., Cowan, M., Wang, L., Hu, Y., Ceze, L., and others. TVM: An automated End-to-End optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pp. 578–594, 2018b.

Chetlur, S., Woolley, C., Vandermersch, P., Cohen, J., Tran, J., Catanzaro, B., and Shelhamer, E. cuDNN: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*, 2014.

Coppersmith, D. and Winograd, S. On the asymptotic complexity of matrix multiplication. *SIAM Journal on Computing*, 11(3):472–492, 1982. Publisher: SIAM.

Cornelissen, B., Holten, D., Zaidman, A., Moonen, L., van Wijk, J. J., and van Deursen, A. Understanding Execution Traces Using Massive Sequence and Circular Bundle Views. In *15th IEEE International Conference on Program Comprehension (ICPC '07)*, pp. 49–58, 2007. doi: 10.1109/ICPC.2007.39.

Dakkak, A., Li, C., Xiong, J., and Hwu, W.-m. MLModelScope: A distributed platform for model evaluation and benchmarking at scale. *arXiv preprint arXiv:2002.08295*, 2020.

Dalton, S., Frosio, I., and Garland, M. Accelerating reinforcement learning through gpu atari emulation. *Advances in Neural Information Processing Systems*, 33: 19773–19782, 2020.

Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pp. 248–255. Ieee, 2009.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. BERT: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

Dozat, T. Incorporating nesterov momentum into adam. 2016.

Eassa, A. and Burc Eryilma, S. The Full Stack Optimization Powering NVIDIA MLPerf Training v2.0 Performance, June 2022. URL https://developer.nvidia.com/blog/boosting-mlperf-training-performance-with-full-stack-optimization/.

Evans, J. How do Ruby & Python profilers work?, 2017. URL https://jvns.ca/blog/2017/12/17/how-do-ruby---python-profilers-work-/.

Facebook. ONNX, 2017. URL https://github.com/onnx/onnx. original-date: 2017-09-07T04:53:45Z.

Facebook. PyTorch record_function Annotation, 2021. URL https://h-huang.github.io/tutorials/recipes/recipes/profiler_recipe.html.

Facebook. PyTorch Profiler — PyTorch Tutorials 1.12.0+cu102 documentation, 2022. URL https://pytorch.org/tutorials/recipes/recipes/profiler_recipe.html.

Frostig, R., Johnson, M. J., and Leary, C. Compiling machine learning programs via high-level tracing. *Systems for Machine Learning*, 4(9), 2018. Publisher: SysML.

Gleeson, J., Gabel, M., Pekhimenko, G., de Lara, E., Krishnan, S., and Janapa Reddi, V. RL-Scope: Cross-stack Profiling for Deep Reinforcement Learning Workloads. *Proceedings of Machine Learning and Systems*, 3:783–799, 2021.

Google. Chrome Trace Viewer, 2012a. URL https://www.chromium.org/developers/how-tos/trace-event-profiling-tool/.

Google. GitHub code history google/trace-viewer, 2012b. URL https://github.com/google/trace-viewer.

Google. TensorBoard: TensorFlow's visualization toolkit, 2015. URL https://www.tensorflow.org/tensorboard.

Google. Google Cloud Profiler, 2022. URL https://cloud.google.com/profiler/docs/about-profiler.

Google. jax.named_scope, 2022. URL https://jax.readthedocs.io/en/latest/_autosummary/jax.named_scope.html.

Google. Perfetto Trace Processor, 2022. URL https://perfetto.dev/docs/analysis/trace-processor.

Google. pprof, October 2022a. URL https://github.com/google/pprof.

Google. TensorFlow Profiler, July 2022b. URL https://github.com/tensorflow/profiler. original-date: 2020-03-10T15:58:33Z.

Graham, S. L., Kessler, P. B., and McKusick, M. K. Gprof: A call graph execution profiler. *ACM Sigplan Notices*, 17(6):120–126, 1982. Publisher: ACM New York, NY, USA.

Graves, A. Sequence transduction with recurrent neural networks. *arXiv preprint arXiv:1211.3711*, 2012.

Greenberg, S. and Buxton, B. Usability evaluation considered harmful (some of the time). In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pp. 111–120. ACM, 2008. doi: 10.1145/1357054.1357074.

Gregg, B. CPU Flame Graphs, 2017a. URL https://www.brendangregg.com/FlameGraphs/cpuflamegraphs.html.

Gregg, B. Visualizing Performance with Flame Graphs. Santa Clara, CA, July 2017b. USENIX Association.

Guo, C., Yuan, L., Xiang, D., Dang, Y., Huang, R., Maltz, D., Liu, Z., Wang, V., Pang, B., Chen, H., and others. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pp. 139–152, 2015.

Hannun, A., Case, C., Casper, J., Catanzaro, B., Diamos, G., Elsen, E., Prenger, R., Satheesh, S., Sengupta, S., Coates, A., and others. Deep speech: Scaling up end-to-end speech recognition. *arXiv preprint arXiv:1412.5567*, 2014.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

Hernandez, D. and Brown, T. B. Measuring the algorithmic efficiency of neural networks. *arXiv preprint arXiv:2005.04305*, 2020.

Horowitz, M. computing's energy problem (and what we can do about it). In *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pp. 10–14. IEEE, 2014.

Hu, H., Jiang, C., Zhong, Y., Peng, Y., Wu, C., Zhu, Y., Lin, H., and Guo, C. dPRO: A Generic Performance Diagnosis and Optimization Toolkit for Expediting Distributed DNN Training. *Proceedings of Machine Learning and Systems*, 4:623–637, 2022.

Iandola, F. N., Han, S., Moskewicz, M. W., Ashraf, K., Dally, W. J., and Keutzer, K. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5 MB model size. *arXiv preprint arXiv:1602.07360*, 2016.

Intel. VTune™ Profiler, 2020. URL https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html.

Ivanov, A., Dryden, N., Ben-Nun, T., Li, S., and Hoefler, T. Data movement is all you need: A case study on optimizing transformers. *Proceedings of Machine Learning and Systems*, 3:711–732, 2021.

Joukov, N., Traeger, A., Iyer, R., Wright, C. P., and Zadok, E. Operating System Profiling via Latency Analysis. In *OSDI*, volume 6, pp. 89–102, 2006.

Kaldor, J., Mace, J., Bejda, M., Gao, E., Kuropatwa, W., O'Neill, J., Ong, K. W., Schaller, B., Shan, P., Viscomi, B., and others. Canopy: An end-to-end performance tracing and analysis system. In *Proceedings of the 26th symposium on operating systems principles*, pp. 34–50, 2017.

Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

LeFebvre, W. top - Linux manual page, 1984. URL https://man7.org/linux/man-pages/man1/top.1.html.

Li, C., Dakkak, A., Xiong, J., Wei, W., Xu, L., and Hwu, W.-m. XSP: Across-Stack Profiling and Analysis of Machine Learning Models on GPUs. pp. 326–327, May 2020. doi: 10.1109/IPDPS47924.2020.00042.

Likert, R. A Technique for the Measurement of Attitudes. In *Archives of Psychology. 140*, pp. 1–55, 1932.

Linux. perf, 2009. URL https://perf.wiki.kernel.org/index.php/Main_Page.

Mattson, P., Cheng, C., Diamos, G., Coleman, C., Micikevicius, P., Patterson, D., Tang, H., Wei, G.-Y., Bailis, P., Bittorf, V., and others. MLPerf training benchmark. *Proceedings of Machine Learning and Systems*, 2:336–349, 2020.

MLCommons. RNN-T Model MLCommons Training, 2021. URL https://github.com/mlcommons/training/tree/d0a86d67e41186835cf3f484f7e3ef00d02b820a/rnn_speech_recognition/pytorch.

MLCommons. MLCommons Algorithmic Efficiency Benchmark RNN-T implementation, 2022. URL https://github.com/mlcommons/algorithmic-efficiency/tree/743fab94ddd64a42133d3542be7a75fe1c405174/algorithmic_efficiency/workloads/librispeech.

Mohror, K. and Karavanic, K. L. Trace profiling: Scalable event tracing on high-end parallel systems. *Parallel Computing*, 38(4):194–225, 2012. ISSN 0167-8191. doi: https://doi.org/10.1016/j.parco.2011.12.003. URL https://www.sciencedirect.com/science/article/pii/S0167819111001852.

Muhammad, H. htop, 2004. URL https://github.com/htop-dev/htop. original-date: 2020-08-17T04:26:40Z.

Munzner, T. Visualization Analysis & Design, June 2014. URL https://www.cs.ubc.ca/~tmm/talks/minicourse14/minicourse14-session1.pdf.

Nielsen, J. Why you only need to test with 5 users. *Alertbox*, 2000. URL https://www.nngroup.com/articles/why-you-only-need-to-test-with-5-users/.

Nielsen, J. Ten usability heuristics, 2005.

NVIDIA. GeForce RTX 2080 Ti Graphics Card, 2018a. URL https://www.nvidia.com/en-me/geforce/graphics-cards/rtx-2080-ti/.

NVIDIA. Nsight Systems, March 2018b. URL https://developer.nvidia.com/nsight-systems.

NVIDIA. Nsight Compute, 2020a. URL https://docs.nvidia.com/nsight-compute/NsightCompute/index.html.

NVIDIA. The NVIDIA Tools Extension Library (NVTX), 2020b. URL http://docs.nvidia.com/nsight-visual-studio-edition/nvtx/index.html.

Panayotov, V., Chen, G., Povey, D., and Khudanpur, S. Librispeech: an asr corpus based on public domain audio books. In *2015 IEEE international conference on acoustics, speech and signal processing (ICASSP)*, pp. 5206–5210. IEEE, 2015.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., and others. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

Petrenko, A., Huang, Z., Kumar, T., Sukhatme, G., and Koltun, V. Sample factory: Egocentric 3d control from pixels at 100000 fps with asynchronous reinforcement learning. In *International Conference on Machine Learning*, pp. 7652–7662. PMLR, 2020.

Reddi, V. J., Cheng, C., Kanter, D., Mattson, P., Schmuelling, G., Wu, C.-J., Anderson, B., Breughe, M., Charlebois, M., Chou, W., and others. MLPerf inference benchmark. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pp. 446–459. IEEE, 2020.

Richter, H., Brotherton, J. A., Abowd, G. D., and Truong, K. N. A Multi-Scale Timeline Slider for Stream Visualization and Control. 1999.

Ruder, S. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

Shacklett, B., Wijmans, E., Petrenko, A., Savva, M., Batra, D., Koltun, V., and Fatahalian, K. Large batch simulation for deep reinforcement learning. *arXiv preprint arXiv:2103.07013*, 2021.

Sinclair, D. Trace Event Format, 2016. URL https://docs.google.com/document/d/1CvAClvFfyA5R-PhYUmn5OOQtYMH4h6I0nSsKchNAySU/preview.

Snider, D. Visualizing time-use in dnn training. *Masters Thesis, University of Toronto*, 2022.

Strubell, E., Ganesh, A., and McCallum, A. Energy and policy considerations for deep learning in NLP. *arXiv preprint arXiv:1906.02243*, 2019.

Sutton, R. The bitter lesson. *Incomplete Ideas (blog)*, 13: 12, 2019.

Thor, B. Answer to "What's difference between monitoring, tracing and profiling?", November 2012. URL https://serverfault.com/a/446970/247223.

Trümper, J., Döllner, J., and Telea, A. Multiscale visual comparison of execution traces. In *2013 21st International Conference on Program Comprehension (ICPC)*, pp. 53–62, 2013. doi: 10.1109/ICPC.2013.6613833.

Vasilache, N., Zinenko, O., Theodoridis, T., Goyal, P., DeVito, Z., Moses, W. S., Verdoolaege, S., Adams, A., and Cohen, A. Tensor Comprehensions: Framework-agnostic high-performance machine learning abstractions. *arXiv preprint arXiv:1802.04730*, 2018.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, \., and Polosukhin, I. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Weber, M., Geisler, R., Brunst, H., and Nagel, W. E. Folding Methods for Event Timelines in Performance Analysis. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, pp. 205–214, 2015. doi: 10.1109/IPDPSW.2015.47.

Weidendorfer, J. Callgrind, 2022. URL https://valgrind.org/docs/manual/cl-manual.html.

WMT. Second Conference on Machine Translation. 2017. URL https://www.statmt.org/wmt17/.

Wolf, T. Training Neural Nets on Larger Batches: Practical Tips for 1-GPU, Multi-GPU & Distributed setups, September 2020. URL https://medium.com/huggingface/training-larger-batches-practical-tips-on-1-gpu-multi-gpu-distributed-setups-ec88c3e51255.

Yu, G. X., Grossman, T., and Pekhimenko, G. Skyline: Interactive In-Editor Computational Performance Profiling for Deep Neural Network Training. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*, pp. 126–139, 2020.

Yu, G. X., Gao, Y., Golikov, P., and Pekhimenko, G. Habitat: A runtime-based computational performance predictor for deep neural network training. In Calciu, I. and Kuenning, G. (eds.), *2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021*, pp. 503–521. USENIX Association, 2021. URL https://www.usenix.org/conference/atc21/presentation/yu.

Zhang, Q., Lu, H., Sak, H., Tripathi, A., McDermott, E., Koo, S., and Kumar, S. Transformer transducer: A streamable speech recognition model with transformer encoders and RNN-t loss. In *ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 7829–7833. IEEE, 2020.

Zheng, B., Vijaykumar, N., and Pekhimenko, G. Echo: Compiler-based GPU memory footprint reduction for LSTM RNN training. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pp. 1089–1102. IEEE, 2020a.

Zheng, B., Jiang, Z., Yu, C. H., Shen, H., Fromm, J., Liu, Y., Wang, Y., Ceze, L., Chen, T., and Pekhimenko, G. DietCode: Automatic Optimization for Dynamic Tensor Programs. In Marculescu, D., Chi, Y., and Wu, C. (eds.), *Proceedings of Machine Learning and Systems*, volume 4, pp. 848–863, 2022.

Zheng, L., Jia, C., Sun, M., Wu, Z., Yu, C. H., Haj-Ali, A., Wang, Y., Yang, J., Zhuo, D., Sen, K., and others. Ansor: Generating High-Performance Tensor Programs for Deep Learning. In *14th USENIX symposium on operating systems design and implementation (OSDI 20)*, pp. 863–879, 2020b.

Zhu, H., Akrout, M., Zheng, B., Pelegris, A., Jayarajan, A., Phanishayee, A., Schroeder, B., and Pekhimenko, G. TBD: Benchmarking and analyzing deep neural network training. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 88–100. IEEE, 2018.

Zhu, H., Phanishayee, A., and Pekhimenko, G. Daydream: Accurately Estimating the Efficacy of Optimizations for DNN Training. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pp. 337–352, 2020.

SUMMARY OF APPENDICES

In Appendix A, we provide instructions for reproducing our results. In Appendix B, we provide accompanying visualizations to our evaluated case studies and two additional case studies. In Appendix C, we provide information essential for developers looking to implement profiling tools analogous to HOTLINE. In Appendix D, we describe our perceived shortcomings of TensorBoard, a popular DNN profiling tool.

A ARTIFACT APPENDIX

A.1 Abstract

This artifact appendix includes the source code and scripts to use HOTLINE to functionally reproduce profiling of our evaluated workloads as described in section 6. More specifically, this appendix contains instructions to generate multi-scale timeline visualizations for ResNet50 as seen in Figure 8, RNN-T as seen in Figure 9, Transformer as seen in Figure 12, and the workload summary seen in Table 1. When following our instructions, the results should be similar to the results we present in the paper.

A.2 Artifact check-list (meta-information)

- **Algorithm:** Automatic annotation described in section 4 and multi-scale timeline described in section 5.
- **Data set:** Scripts included to download CIFAR-10 to simulate ImageNet, WMT, and a minimal LibriSpeech dataset.
- **Run-time environment:** Automatically managed by Dockerfiles which install Ubuntu 20.04, PyTorch 1.12. We used CUDA 11.2 but also tested CUDA 12.0.
- **Hardware:** A single machine with 4 Nvidia GPUs. Or with 1 GPU it is possible to reproduce most of the results. We have tested RTX 2080Ti, RTX 3090, and V100 GPUs.
- **Run-time state:** Sensitive to runtime state. No concurrent workloads should be running while running experiments.
- **Execution:** Our scripts run PyTorch in `DataParallel` mode. Other parallelism modes are not yet supported.
- **Output:** Multi-scale timeline visualizations of time-use for each model evaluated (ResNet50 as seen in Figure 8, RNN-T as seen in Figure 9, and Transformer as seen in Figure 12) and the workload summary as seen in Table 1. **You should be able to access HTTP on port 7234 of the machine** to view the Hotline web UI for the multi-scale timeline results.
- **How much disk space required (approximately)?:** 30 GB.
- **How much time is needed to prepare workflow (approximately)?:** 1 hour to build containers and download datasets.
- **How much time is needed to complete experiments (approximately)?:** 30 minutes.
- **Publicly available?:** Yes.
- **Code licenses (if publicly available)?:** Apache 2.0
- **Archived (provide DOI)?:** `https://doi.org/10.5281/zenodo.7791393`

A.3 Description

A.3.1 How delivered

The artifact can be downloaded either from the GitHub link `https://github.com/UofT-EcoSystem/hotline` or from the DOI link `https://doi.org/10.5281/zenodo.7791393`.

A.3.2 Hardware dependencies

We recommend a setup similar to 4 × 2080Ti GPUs, AMD EPYC 7601 CPU, and 128 GB of RAM in order to have the most similar profiling results. If you have only 1 × GPU it is still possible to reproduce most of the results. When using a different GPU the training iteration runtime and the number of traced events will be somewhat different. When using a faster GPU than a 2080Ti the results will be more CPU-bounded. A GPU with less than 11 GB of VRAM will need to have batch sizes reduced in the `run.sh` script provided.

A.3.3 Software dependencies

We recommend using a machine with Ubuntu 20.04 and Docker installed to reproduce the results. We tested on CUDA 11.2 but also confirmed that some newer versions (e.g. CUDA 12.0) are backward compatible with our experiments but produce slightly different results. Other software packages for this artifact are installed automatically by the provided Dockerfiles.

A.3.4 Data sets

For convenience, our scripts automatically download CIFAR-10 to simulate ImageNet, WMT, and a minimal LibriSpeech dataset. The datasets require 10 GB of space and Docker requires 20 GB.

A.4 Installation

Start by downloading the code from the DOI or `git clone https://github.com/UofT-EcoSystem/hotline`. Then navigate to the artifact directory with `cd hotline/artifact` and build the Docker containers with `bash build.sh`.

A.5 Experiment workflow

Once the runtime environment has been installed, execute the experiments with `bash run.sh` to generate the results which are saved to the `hotline/artifact/results` directory.

A.6 Evaluation and expected result

Once the `run.sh` script has finished, a workload summary table similar to Table 1 should be seen at the bottom of the output. Secondly, browse to `http://localhost:7234/` and you should see multi-scale timeline visualizations for each model evaluated. If using a remote machine, you can forward port 7234 to `localhost` with the command: `ssh -L 7234:localhost:7234 remote`. You should be able to navigate the timeline to find results that look similar to Figure 8 for ResNet50, Figure 9 for RNN-T, and Figure 12 for the Transformer model.

A.7 Methodology

Submission, reviewing and badging methodology:

- `http://cTuning.org/ae/submission-20190109.html`
- `http://cTuning.org/ae/reviewing-20190109.html`
- `https://www.acm.org/publications/policies/artifact-review-badging`

B Extended Hotline Visualizations

B.1 Case Study: ResNet-50 on a Single GPU

In the first case study, we show how Hotline enables ML developers to quickly see high-level and low-level bottlenecks. Figure 8 shows drilling down to the lowest level in Hotline, where we find that ResNet-50 on 1 GPU is bottlenecked by GPU kernel runtime and that a faster GPU could speed up training by as much as 40% before the CPU portion limits further speedup. To test this claim we executed the same batch size on a faster Nvidia 3090 GPU and found a single layer in the forward pass had within 1% of the same number of kernels and ran them 42% faster. However, our 3090 GPU machine also had a faster CPU, meaning that the possible speedup can be even higher.

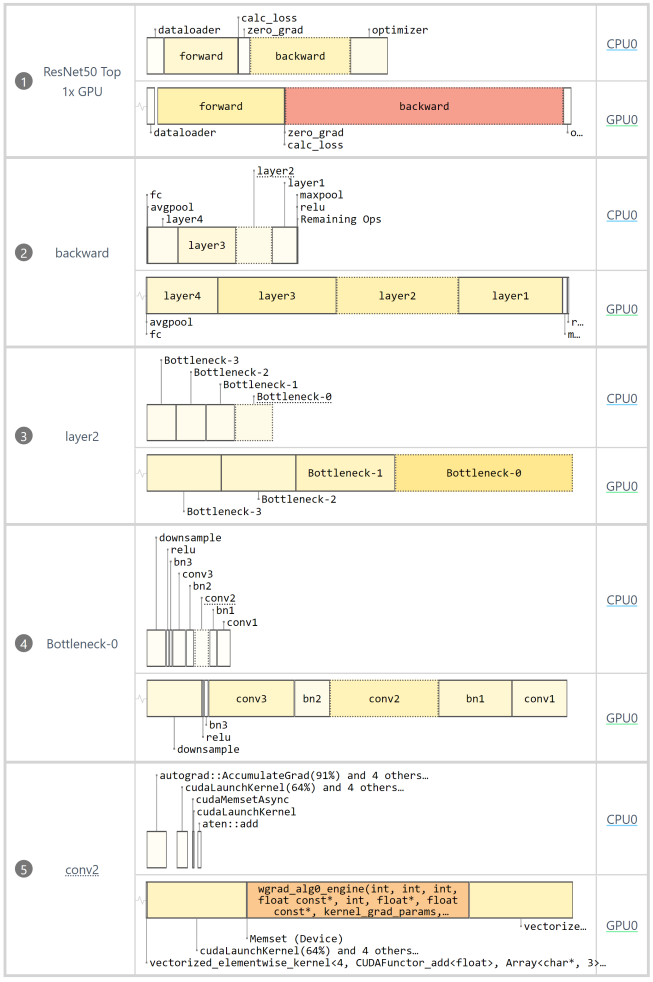


Figure 8. Hotline drill down on the slowest operation at each granularity for ResNet-50 (He et al., 2016) training on one GPU. The longest and brightest segments are the backward pass at a high-level and a GPU convolution kernel at a low-level.

B.2 Case Study: Launch Overhead in RNN-T

In the second case study, we show how Hotline can perform rapid low-level investigations by using Hotline's "Open with Perfetto" button to isolate interesting portions of the runtime trace and open them with a tool that is complimentary to Hotline. In Figure 9, we immediately see a major bottleneck at all levels of the multi-scale timeline, and a low-level cause: that that 73% of the runtime of LSTM units is spent on "`cudaLaunchKernel`" and "`sgemm`" kernels.

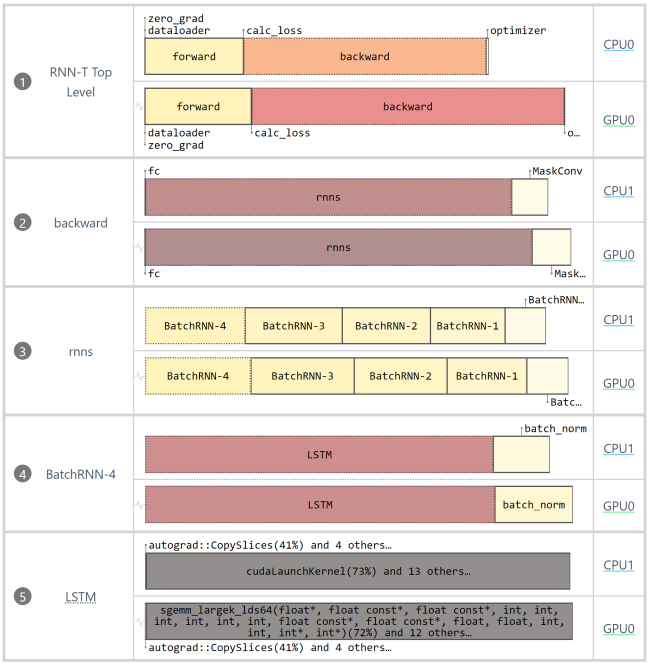


Figure 9. Hotline drill down on the slowest operation at each granularity for RNN-T (Graves, 2012) training on 4 GPUs. The longest and darkest segments indicate bottleneck locations.

Using Hotline to view only one LSTM unit of the backward pass in Perfetto, we discovered that the GPU is only executing kernels 30% of the time and is bottlenecked by CUDA API launch overhead, as seen in Figure 10.

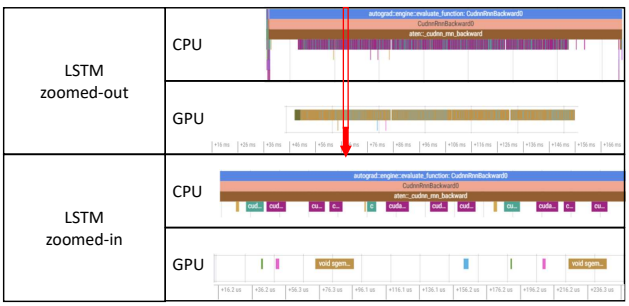
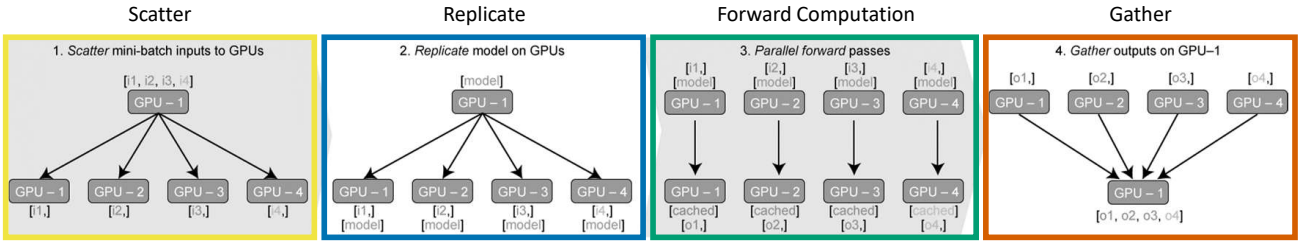
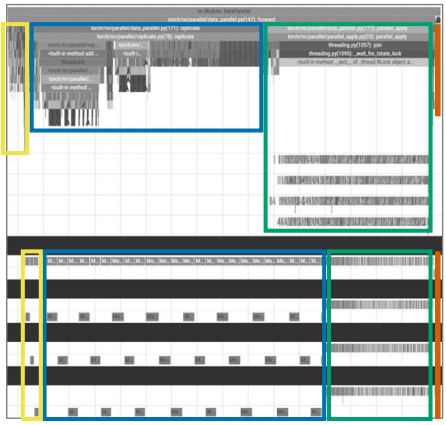


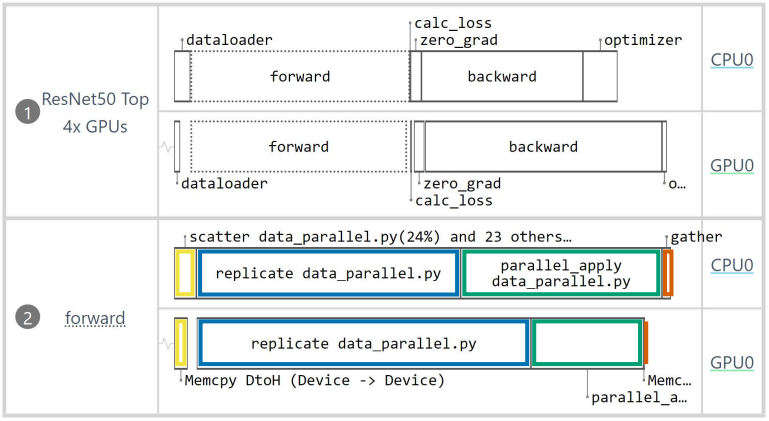
Figure 10. Manually assembled Perfetto screenshots of launch-bound LSTM unit in the backward pass of the RNN-T model.



(a) Abstract stages of forward pass with PyTorch.DataParallel training



(b) Perfetto



(c) Hotline

Figure 11. The major stages of the forward pass in PyTorch's `DataParallel` training mode are seen in three ways: (a) conceptual diagram (Wolf, 2020), (b) runtime trace in Perfetto of ResNet-50 on 4 GPUs, (c) and the same trace visualized with HOTLINE where the sections have been automatically detected. We have muted original colors and drawn colored boxes by hand.

B.3 Case Study: Bottlenecks in Data Parallel Training

In the third case study, we show how HOTLINE's annotation of arbitrary sections of training enabled us to find unexpected bottlenecks. Figure 11a illustrates the stages that PyTorch's default `DataParallel` training mode takes under the hood during the forward pass. In Figure 11b, we manually draw a different colored box for each stage, but without this, these stages are hard to observe when viewing the raw trace in Perfetto. In Figure 11c, HOTLINE has automatically detected these stages and displayed a compact summary. Using HOTLINE, we found that when training ResNet-50 on 4 GPUs, PyTorch spends 67% of the forward pass replicating the DNN model to all GPUs and 50% of the backward pass reducing gradients onto 1 GPU from the other 3 GPUs. Together, this accounts for 51% of end-to-end training time. This may be a surprise to users who are not aware of these stages or that they are so slow.

B.4 Case Study: Transformer Model

In the fourth case study, we show how HOTLINE can help ML developers understand the structure of DNN models and the operation of DNN training. Figure 12 shows the slowest operations of training a Transformer model on 4 GPUs. We observe two unique differences compared to ResNet-50 and RNN-T training: the "`Calc Loss`" step is a more expensive operation, and the "`Loss Gradient`" operation (performed on all GPUs) triggers expensive inter-GPU "`Scatter`" and "`Gather`" operations and we question whether this is optimal.

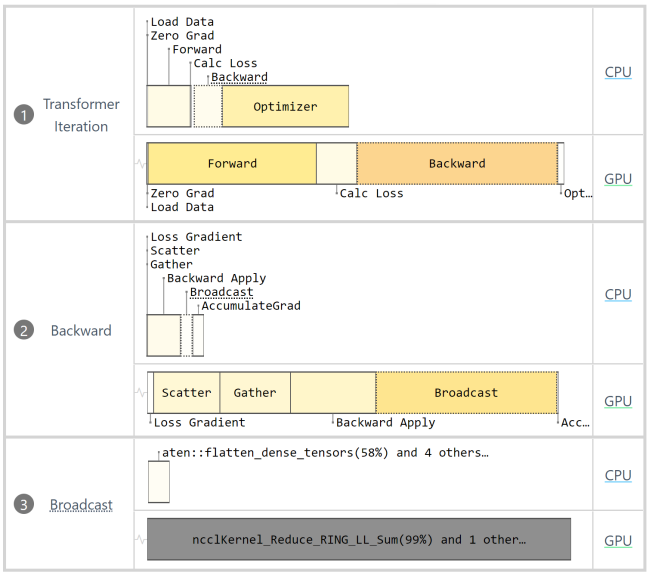


Figure 12. HOTLINE drill down on the slowest operation at each granularity for Transformer (Zhang et al., 2020) training on 4 GPUs. The backward pass is slowed by `NCCL` operations responsible for reducing gradients from all GPUs onto one GPU.

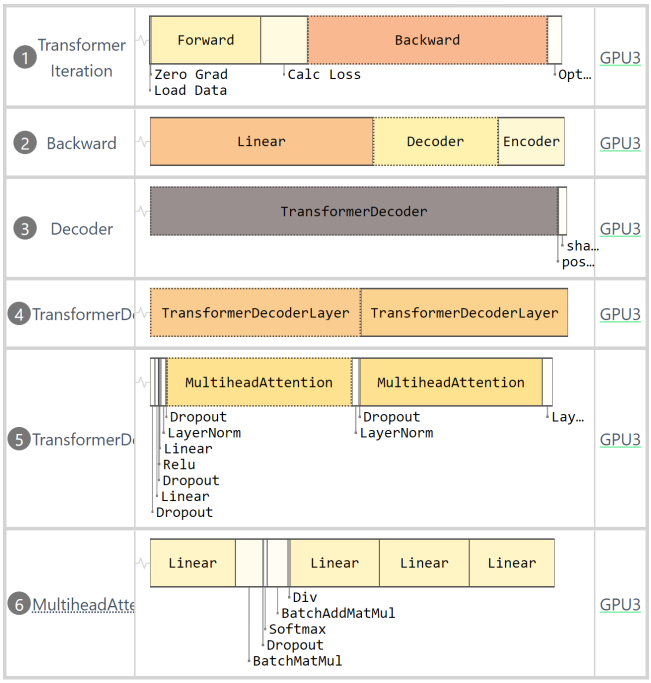


Figure 13. HOTLINE drill down of GPU runtime only on the model architecture of a Transformer-small model training on a single GPU. A Transformer-small is representative of the Transformer model but has been shrunk for demonstration purposes from 6 layers per encoder/decoder to 2 layers, and from 16 attention heads to 2.

Next, we discuss how HOTLINE helps ML developers understand the structure and runtime of the Transformer model on a single GPU. Level "`2. Backward`" in Figure 13 shows that at the highest level the Transformer model is composed of an encoder, decoder, and a last linear layer. Note that in a full-sized Transformer model, the linear layer would have a smaller proportion of runtime because the encoder and decoder would have more layers and heads. Level "`3. Decoder`" is composed of a shared embedding, positional encoder, and the decoder itself which is responsible for the majority of runtime at this level. Level "`4. TransformerDecoder`" is composed of 2 identical decoder layers. Level "`5. TransformerDecoderLayer`" is composed of several layer norm and dropout operations, two multi-head attention units, and a feedforward network made of relu and linear operations. Level "`6. MultiheadAttention`" is composed of three linear operations, a scaled dot product attention (the group of shorter operations), and a final linear operation.

B.5 Extended Methodology

Here we provide additional details about the models and training parameters used in our case studies. While we did not train until model convergence because HOTLINE only needs one sample of training iteration (after a user-defined number of warmup iterations), we did select realistic training parameters unless otherwise specified. We carefully selected a diverse set of models, i.e. ResNet-50, RNN-T, and Transformer, to represent important DL subfields and model architectures.

The ResNet-50 (He et al., 2016) model has 23 million parameters and we applied it to image classification on the ImageNet dataset (Deng et al., 2009). For training ResNet-50 we used a batch size of 256 and stochastic gradient descent (SGD) with momentum and weight decay.

The RNN-T (MLCommons, 2021; Graves, 2012) model has 57 million parameters (Zhang et al., 2020) and is for speech-to-text recognition on the LibriSpeech100 dataset (Panayotov et al., 2015). For training RNN-T we used Adam (Kingma & Ba, 2014) with a batch size of 8, less than the standard 256 batch size to fit on our GPU. RNN-T training typically involves many operations on small vectors (Zheng et al., 2020a) and is memory capacity-bounded (Zhu et al., 2018).

The Transformer (Vaswani et al., 2017) model has 209 million parameters in our configuration of 6 layers in the decoder and encoder, 16 heads per multi-head attention unit, and 4096 hidden units in the feedforward network. We applied it to language translation on the WMT dataset (WMT, 2017) and trained using Adam (Kingma & Ba, 2014) with a batch size of 32, less than the standard 128 batch size to fit on our GPU.

C HOTLINE IMPLEMENTATION DETAILS

C.1 Limited Support for Kernel Fusion

A challenge arises when multiple operations in the DNN model definition, e.g., Conv and Relu, map to the same GPU kernel (i.e. kernel fusion). This is the case for the kernel, "`volta_scudnn_winograd_128x128_ldg1_ldg4_relu_tile148t_nt_v1`", which is found in the forward pass of ResNet-50. To address this challenge, HOTLINE supports detecting arbitrary combinations of DNN model operation names in the same GPU kernel name. Further research is required to support JIT compilation as in JAX (Frostig et al., 2018) where all JIT-compiled kernels have the same name, however, this is not a new problem in the field of profiling and one solution is to use symbol tables (Gregg, 2017a).

C.2 Noise Reduction

A primary motivation of our paper is that existing runtime trace viewers fail to convey useful information due to information overload (Bohnet et al., 2009; Weber et al., 2015). As such, we have designed a suite of heuristics that trade completeness of information for quick and useful interpretation. In Section 4, we introduced techniques to group the majority of runtime trace events into more digestible and often familiar annotations. Before presenting annotations to the user, we apply the following noise reduction heuristics. Users can forgo the following modifications by using HOTLINE UI's "Open with Perfetto" button to view the original trace within any annotation when needed.

Rename. Operation names in runtime traces often include file paths, symbols, or repetitive boilerplate. HOTLINE uses simple rules to rename operations to be more concise (examples in Appendix C.3).

Summarize. When HOTLINE generates an arbitrary annotation for a group of operations as described in Section 4.5, HOTLINE also generates a summarizing name based on the runtime contribution of the constituent operations. For example a name of, "A(60%) and 5 others...", would mean that this annotation contains 6 unique operation names, with "A" dominating runtime the most, at 60% of runtime.

Hide. When multiple accelerators are used in parallel for DNN training it may be overwhelming to display them all to users. By default, HOTLINE's UI will only display the accelerator with the longest runtime, i.e. the *straggler*, and the user can reveal the others with a checkbox (shown in Appendix C.5).

Name Placement. Existing trace viewers will cut-off operation names that do not fit on-screen. HOTLINE prevents this by taking into account the user's screen size to calculate a non-overlapping placement of operation names (details in Appendix C.4).

Align Timelines. Due to asynchronous GPU execution (*Challenge 3*), the CPU and GPU portions for a single operation are commonly very distant in DNN training timelines, leading to a poor viewing experience. HOTLINE's UI compensates for this by aligning the CPU and GPU portions to make them appear to start at the same time. A ⏦ icon is displayed to indicate this adjustment.

C.3 Renaming Operations To Be More Concise

To simplify the interpretation of runtime traces for developers, one technique in HOTLINE's suite of noise reduction heuristics is renaming operations to be more concise by removing file paths, symbols, or repetitive boilerplate. Figure 14 shows several examples in which the length of CPU operation names has been reduced by 65% and GPU operation names reduced by 22%. Original names can still be viewed using HOTLINE's "Open with Perfetto" button.

Operation Name	Hotline Renamed
torch/utils/data/**dataloader.py**(1173): **_get_data**	_get_data dataloader.py
torch/nn/parallel/**comm.py**(188): <**listcomp**>	listcomp comm.py
typing.py(306): **inner**	inner typing.py
<built-in method **acquire** of **_multiprocessing.SemLock** object at 0x7f86f5bc91f0>	acquire SemLock
<built-in method **_scatter** of **PyCapsule** object at 0x7f8801ea3f90>	_scatter PyCapsule
<**string**>(1): <**lambda**>	lambda string
<built-in function **print**>	print

(a) CPU Operations

Operation Name	Hotline Renamed
void at::native::**vectorized_elementwise_kernel<4, at::native::CUDAFunctor_add<float>, at::detail::Array<char*, 3> >(int, at::native::CUDAFunctor_add<float>, at::detail::Array<char*, 3>)**	vectorized_elementwise_kernel<4, CUDAFunctor_add<float>, Array<char*, 3> >(int, CUDAFunctor_add<float>, Array<char*, 3>)
void **wgrad_alg0_engine<float, 128, 6, 8, 3, 3, 5, false, 512>(int, int, int, float const*, int, float*, float const*, kernel_grad_params, unsigned long long, int, float, int, int, int, int)**	wgrad_alg0_engine<float, 128, 6, 8, 3, 3, 5, false, 512>(int, int, int, float const*, int, float*, float const*, kernel_grad_params, unsigned long long, int, float, int, int, int, int)
void at::native::**vectorized_elementwise_kernel<4, at::native::BinaryFunctor<float, float, float, at::native::threshold_kernel_impl<float>(at::TensorIteratorBase&, float, float)::{lambda(float, float)#1}>, at::detail::Array<char*, 3> >(int, at::native::BinaryFunctor<float, float, float, at::native::threshold_kernel_impl<float>(at::TensorIteratorBase&, float, float)::{lambda(float, float)#1}>, at::detail::Array<char*, 3>)**	vectorized_elementwise_kernel<4, BinaryFunctor<float, float, float, threshold_kernel_impl<float>(TensorIteratorBase&, float, float)::{lambda(float, float)#1}>, Array<char*, 3> >(int, BinaryFunctor<float, float, float, threshold_kernel_impl<float>(TensorIteratorBase&, float, float)::{lambda(float, float)#1}>, Array<char*, 3>)

(b) GPU Operations

Figure 14. Examples of HOTLINE renaming operations to be more concise. Bold text indicates what is retained.

C.4 Non-Overlapping Label Placement

Existing trace viewers will cut-off operation names that do not fit on-screen. We designed an algorithm for HOTLINE to intelligently place names nearby when they cannot fit inside the operation's displayed box as seen in Figure 15. The algorithm takes into account the user's screen size to calculate a non-overlapping placement before drawing and will react dynamically to the user resizing their screen.

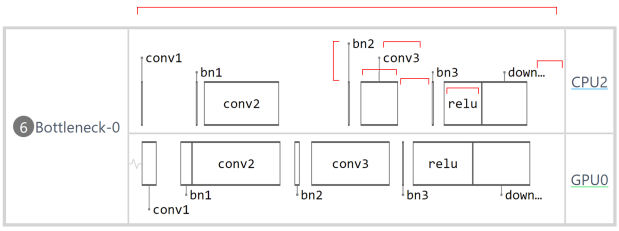
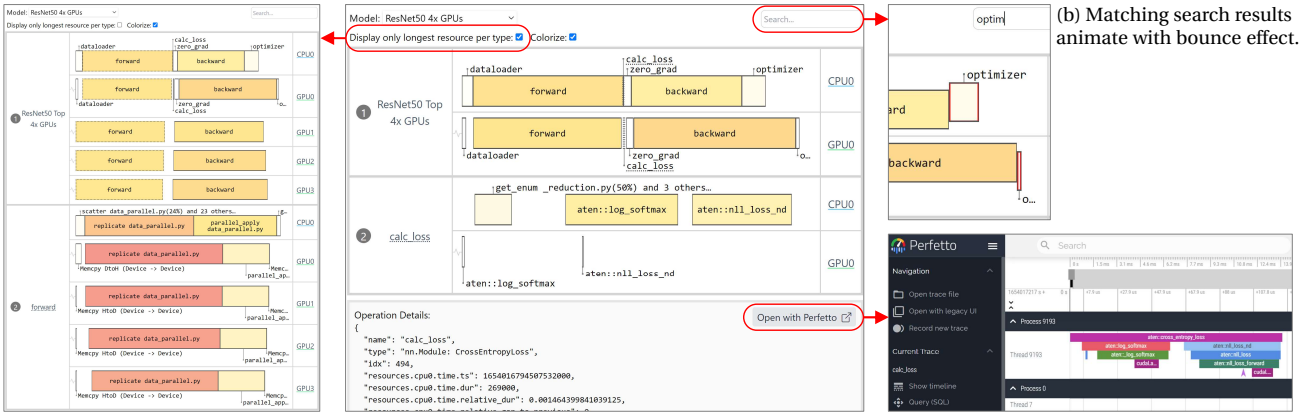


Figure 15. HOTLINE calculates non-overlapping label placement using the dimensions annotated in red.



(a) Operation "forward" with all hardware resources revealed.

(b) Matching search results animate with bounce effect.

(c) Operation "calc_loss" opened with the Perfetto.

Figure 16. Illustration of HOTLINE's interactive features.

C.5 Interactive Features

To assist ML developers in finding and understanding time-use bottlenecks in DNN training, we developed three interactive features shown in Figure 16. Specifically, (a) "Display Only Longest" summarizes complex parallelized timelines for faster understanding, (b) "Search" helps users find operations of interest, and (c) "Open with Perfetto" helps users understand bottlenecks by displaying the most detailed view of raw events. Next, we describe these features in more detail.

Display Only Longest: To address the challenge of information overload in DNN training runtime traces, HOTLINE will only display the longest resource per type by default, i.e. the straggler. That means if there are 4 GPUs, only the GPU with the longest runtime will be displayed. We believe it is a reasonable default for data-parallel training (the most common technique for multi-GPU training) because the same operations repeat on all GPUs and the only difference between them is runtime.

Search: If a user knows what they want to investigate, they can search. Operations will turn red and animate with a bounce effect when the search term partially matches the annotation name or exactly matches any raw event contained within the annotation.

Failing Gracefully: Users can click the "Open with Perfetto" button to display the raw events for any annotation. This is useful when there are too many operations to display in HOTLINE or when users want an exact representation of the runtime trace without summarization or other noise reduction techniques. HOTLINE and Perfetto are good at different things, the former is digestible and the latter exact. This integration makes it possible to quickly switch between tools and get the best of both. Future integration with Nsight Compute for resource utilization is planned.

D LIMITATIONS OF TENSORBOARD

Each DL framework has a different profiler to collect performance data, but the most popular visualization tool is TensorBoard (Google, 2015) because it is officially supported by the three most popular DL frameworks: Tensorflow (Abadi et al., 2016), PyTorch (Paszke et al., 2019), and Jax (Frostig et al., 2018). Here we explain the shortcomings of TensorBoard which make it difficult to use and help to motivate HOTLINE.

What's wrong with TensorBoard? As seen in Figure 17a-e, donut charts and timelines are the most used visualizations in this tool. However, neither of these methods works well when there are **too many data points**, as is common in DNN training, which consists of many thousands of operations. Secondly, the TensorBoard interface is **not meaningful** to ML developers because the aggregated performance metrics **lack DNN training concepts** that are familiar to ML developers. The DNN training loop is not expressed anywhere and the DNN model hierarchy is visualized poorly in the Module View (Figure 17e) for two reasons. First, the model hierarchy shown does not include user-defined names of DNN layers or operations which would be more informative than using PyTorch class names like "Sequential". Secondly, Module View's table and timelines give a poor sense of how time is spent because **colors are not used effectively**. The trace viewer (Figure 17d) used in TensorBoard is cluttered and **hard to navigate**. It was originally developed in 2012 (Google, 2012b) and has not seen much progress since. Finally, TensorBoard's various **pages of profiler data are disconnected from each other**; one cannot click on an operation to see the corresponding information in another view.

These problems in the most popular DNN performance visualization tool help to motivate our research on new performance investigation techniques for ML developers.

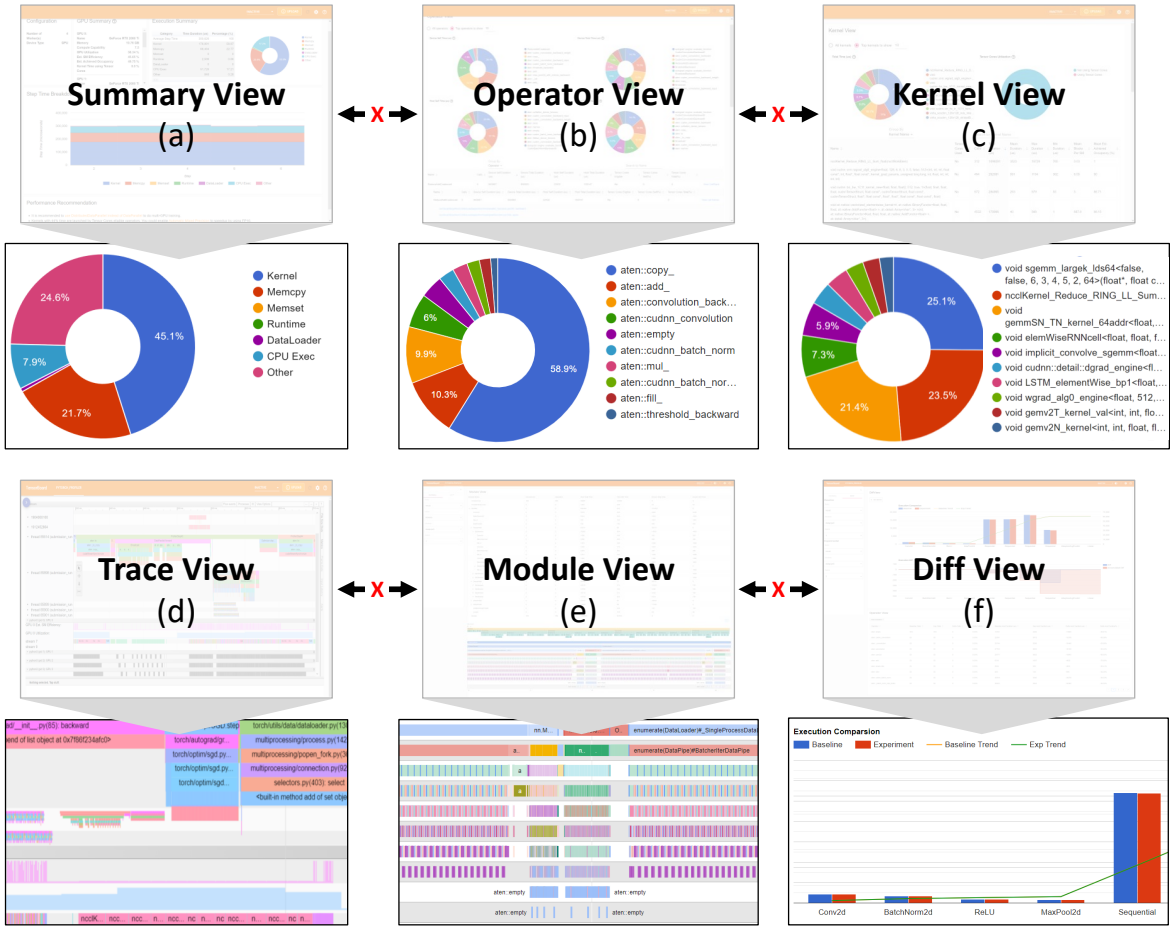


Figure 17. Visualizations in TensorBoard for PyTorch's Profiler.