Skyline: Interactive In-Editor Computational Performance Profiling for Deep Neural Network Training

Geoffrey X. Yu[*‡], **Tovi Grossman**[*], **Gennady Pekhimenko**[*‡]
[*]University of Toronto
Toronto, Ontario, Canada
[‡]Vector Institute
Toronto, Ontario, Canada
gxyu@cs.toronto.edu, tovi@dgp.toronto.edu, pekhimenko@cs.toronto.edu

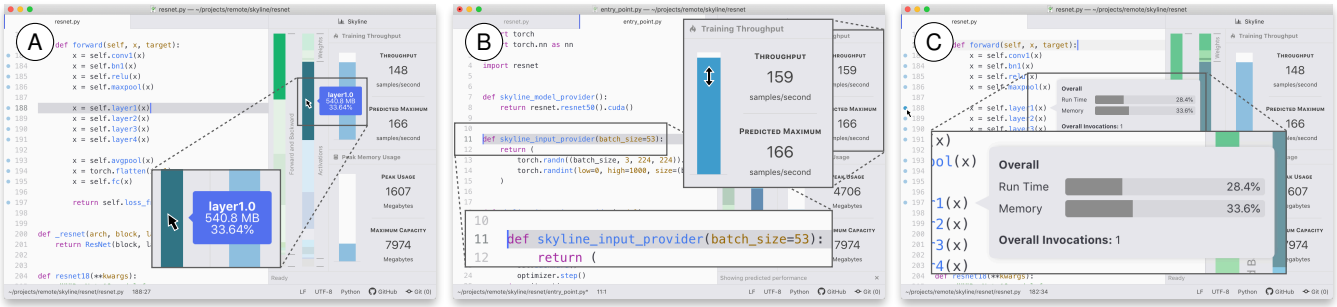

Figure 1. An overview of SKYLINE and its features. (a) Hovering over the visualizations highlights the relevant line(s) of associated code and reveals detailed performance information. (b) Manipulating the throughput and memory usage bar charts results in predictive changes to the batch size that help achieve the desired performance metrics. (c) Hovering over markers for specific lines of code reveals performance details for that line of code.

ABSTRACT

Training a state-of-the-art deep neural network (DNNs) is a computationally-expensive and time-consuming process, which incentivizes deep learning developers to debug their DNNs for computational performance. However, effectively performing this debugging requires intimate knowledge about the underlying software and hardware systems—something that the typical deep learning developer may not have. To help bridge this gap, we present SKYLINE: a new interactive tool for DNN training that supports in-editor computational performance profiling, visualization, and debugging. SKYLINE's key contribution is that it leverages special computational properties of DNN training to provide (i) interactive performance predictions and visualizations, and (ii) directly manipulatable visualizations that, when dragged, mutate the batch size in the code. As an in-editor tool, SKYLINE allows users to leverage these diagnostic features to debug the performance of their DNNs during development. An exploratory qualitative user study of SKYLINE produced promising results; all the participants found SKYLINE to be useful and easy to use.

Author Keywords

Skyline; interactive performance profiling; debugging; visualization; deep neural networks; machine learning.

UIST '20, October 20–23, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-7514-6/20/10 ...$15.00.
http://dx.doi.org/10.1145/3379337.3415890

CCS Concepts

•**Human-centered computing** → **Interactive systems and tools;**

INTRODUCTION

In recent years, deep neural networks (DNNs[1]) have led to breakthroughs in classical machine learning tasks such as image classification [25, 27, 34, 50], object detection [24], machine translation [53, 54], speech recognition [6], and recommendations [15]. As a result, there has been an exciting effort by researchers and practitioners to apply DNNs in practice.

However, a central concern when using DNNs in practice is that they can be computationally-expensive, memory-intensive, and time-consuming to *train*—ranging from hours to *weeks* [14, 17, 37, 52, 58]. Not only can this process cost users time, but it can also translate to substantial monetary costs (e.g., cloud compute costs) and, indirectly, non-trivial amounts of carbon emissions. For example, Strubell et al. found that training BERT [17], a state-of-the-art DNN used for language modeling, can result in carbon emissions roughly equivalent to a trans-American flight and would incur over $12,000 in cloud compute costs [52]. To make matters worse, the computational cost of DNN training is only projected to continue increasing, doubling every 3.4 months [7].

As a result, these costs incentivize deep learning developers to debug and tune their DNNs for *computational performance*.[2] This tuning involves experimenting with different DNN designs, as well as adjusting parameters such as the *batch size*:

[1]Note that in this paper, we also refer to DNNs as *models* and use the two terms interchangeably.
[2]In this paper, when we refer to a DNN's performance we mean its *computational* training performance—*not* its prediction accuracy.

the number of data samples processed during a single training step. For example, prior work found that larger batch sizes could lead to a higher training throughput for certain models [14, 37, 58], which helped motivate later efforts to enable training with larger batch sizes on the same hardware platforms [28, 57].

Today, effectively performing this kind of debugging and tuning requires intimate knowledge of and visibility through the *entire* software and hardware stack—something the typical deep learning developer may not have. This is because deep learning developers primarily work at the top of this stack, implementing their models using high-level software frameworks such as PyTorch [46] or TensorFlow [1]. By design, these frameworks abstract away the underlying hardware (e.g., graphics processing units (GPUs) [42] or tensor processing units (TPUs) [29]) and software systems used during training, which are critical for tuning, understanding, and debugging performance. This abstraction results in a *loss of information*, where developers may face challenges connecting the high-level code they write to the performance characteristics of their models (e.g., training throughput and memory usage).

Unfortunately, much of the existing work on machine learning performance debugging has not addressed computational performance, and instead has focused on *model quality* (i.e. helping improve a model's prediction accuracy) [5, 30, 33, 47, 51]. To help bridge this gap, we make the key observation that DNN training has a set of favorable *properties* that enable the use of interactive features that can help with performance understanding and debugging. DNN training (i) consists of *short repetitive iterations*, enabling rapid profiling; (ii) has *predictable* performance with respect to batch size changes, enabling performance suggestions; and (iii) is implemented using *structured software frameworks*, enabling the linking of performance details to specific lines of code.

In this work, we leverage these properties to develop SKYLINE: a new interactive computational performance profiling and debugging tool for DNN training. SKYLINE visualizes domain-specific performance metrics *in the editor* where developers write their code, and also links these visualizations to their associated lines of code (Figure 1(a) and 1(c)). Some of these features are inspired by prior work on in-editor tooling [8, 13, 26, 47] and live programming environments [31, 36]. SKYLINE extends these ideas to DNN training performance and introduces directly manipulatable performance visualizations that mutate the batch size in the code when dragged (Figure 1(b)). As a *computational* performance profiling tool, SKYLINE does not make predictions about a model's accuracy (see our Discussion section) nor does it provide model accuracy diagnostics, which has been explored in prior work [30, 33, 51]. However, we see SKYLINE as a step toward unified deep learning development tools that include diagnostics and predictions for both performance and model accuracy.

Finally, as an initial evaluation, we conducted an exploratory qualitative user study that examined SKYLINE's diagnostic capabilities. In our study, seven deep learning researchers (i) used SKYLINE to carry out five performance investigation tasks across three state-of-the-art DNNs: GNMT [55], the Transformer [54], and ResNet-50 [25], and (ii) completed a qualitative questionnaire about the usefulness and ease of use of SKYLINE and its features. Our study results were promising, with all participants either agreeing or strongly agreeing that SKYLINE is useful and easy to use—paving the way for a more comprehensive future study about how SKYLINE might help influence model design.

In summary, the contributions of this work are:

- Empirically-derived analytical performance models that predict a DNN's training throughput and memory usage given a batch size.
- Directly manipulatable visualizations that, when dragged, use these performance models to suggest batch sizes in the code that may achieve a desired training throughput or memory usage.
- The design, implementation, and exploratory qualitative evaluation of SKYLINE: an interactive in-editor computational performance profiling tool for DNN training.

RELATED WORK

SKYLINE builds upon bodies of prior work on (i) interactive model quality debugging tools [5, 30, 33, 47, 51], (ii) integrated development environment (IDE)-based performance tools [8, 13, 26], (iii) live programming environments for code understanding [23, 31, 36], and (iv) other similar commercial tools [3, 22, 43, 44]. The key fundamental differences between SKYLINE and these prior works are that SKYLINE (i) focuses on DNN computational performance understanding and debugging, an increasingly important problem as we described previously; and (ii) leverages the special properties of DNN training computation to implement new interactive and manipulatable visualizations linked to the batch size in the code.

Machine Learning Accuracy Debugging Tools

Much of the prior work on machine learning performance debugging focuses on performance from the perspective of *model quality* [5, 30, 33, 47, 51]. ModelTracker [5], a tool for general supervised learning models, helps users investigate a model's prediction accuracy on a dataset both for individual data points and in aggregate. Prospector [33], ActiVis [30] and Seq2Seq-Vis [51] address model interpretability: understanding why models make certain predictions. Gestalt [47], on the other hand, is a development environment for machine learning—similar to SKYLINE in spirit. However, Gestalt focuses on the entire machine learning pipeline: loading and processing data, implementing models, analyzing predictions made by models, and fixing correctness bugs in the pipeline.

In contrast with all these works, SKYLINE tackles the *computational* performance aspect of machine learning, with a focus on DNN training. SKYLINE aims to help users understand and debug the execution performance of their models during training—an increasingly important problem as DNNs have become more widely used in recent years.

IDE-Based Performance Debugging

Prior work has also explored the idea of surfacing performance information inside a text editor (e.g., an IDE) [8, 13], including placing visualizations in situ with the code [26]. For example,

Feature / Functionality	nvprof [43]	DLProf [44]	TF Profiler [22]	PyProf [3]	PyTorch [46]	SKYLINE
Shows GPU kernel info	✓	✓	✓	✓	–	–
Shows thpt. / iter. run time	✓	✓	✓	✓	✓	✓
Iter. run time breakdown	–	✓	✓	✓	✓	✓
Shows total memory usage	–	–	P	–	✓	✓
Memory usage breakdown	–	–	P	–	✓	✓
Hierarchical breakdowns	–	–	–	–	–	✓
In-editor profiling	–	–	–	–	–	✓
Code-linked visualizations	–	–	–	–	–	✓
Interactive thpt. and mem. vs. batch size predictions	–	–	–	–	–	✓

Table 1. A feature comparison between SKYLINE and existing DNN training profiling tools, as of July 2020. P = Planned support in the future. See the SKYLINE section for a description of these features.

PerformanceHat [13] helps software engineers debug performance bottlenecks in general-purpose Java code by surfacing production monitoring information inside the IDE. SKYLINE's approach is inspired by these systems, but is complementary as SKYLINE specifically addresses DNN training performance. Moreover, SKYLINE leverages the special properties of DNN training computation to extend these ideas by introducing manipulatable visualizations linked to the batch size in the code.

Live Programming Environments

SKYLINE's ability to provide live, up-to-date performance information is inspired by prior work on live programming environments [23, 31, 36]. Theseus [36] and Omnicode [31] visualize line-by-line execution results of JavaScript and Python code respectively. These systems help programming novices understand the execution of their code and develop mental models about programming. SKYLINE is different because it takes this idea in the direction of *computational performance* for DNN training. Through interactive visualizations, SKYLINE aims to help deep learning users (who may lack systems-level expertise) understand, debug, and develop mental models about the computational performance of their DNNs.

Existing DNN Profiling Tools

A few existing tools also help with performance debugging. The TensorFlow (TF) Profiler [22] is similar in spirit to SKYLINE, but is instead a standalone tool that lacks SKYLINE's interactive performance predictions. DLProf [44], PyProf [3], and PyTorch's built-in profiler [46] are primarily command-line based DNN performance profilers, and nvprof [43] is a general-purpose GPU performance profiler. Table 1 shows a feature comparison between SKYLINE and these existing tools. In a nutshell, SKYLINE is fundamentally different from these existing tools because it surfaces high-level domain-specific performance information *in-editor* and has manipulatable visualizations linked to the batch size in the code.

THE NEED FOR PERFORMANCE UNDERSTANDING

As previously described, much of the existing work has focused on machine learning performance debugging from the perspective of helping users understand and improve a model's *prediction accuracy* [5, 30, 33, 47, 51]. In this work, we take a complementary approach by focusing on *computational performance*, specifically for DNN training. In this section, we provide some background information about DNN training and discuss some of the common considerations for computational training performance.

Background on DNNs and DNN Training

DNNs, at their heart, are mathematical functions that produce predictions for a specific task using *learned* parameters—often called *weights* [20]. In practice, DNNs are built by assembling together a series of *operations*, often called *layers*, each of which may have weights. To make a prediction for an input, a DNN applies each of its operations in sequence to the input.

A DNN's weights are learned during an iterative process called *training* [20]. In each training iteration, the DNN processes a *batch* of inputs. Based on the prediction errors it makes, the DNN's weights are then updated in an attempt to reduce its prediction error. This process repeats until the DNN reaches an acceptable prediction accuracy [20].

In practice, DNNs are often trained using hardware accelerators such as GPUs [42] or TPUs [29] because they offer significant performance improvements over CPUs [29, 37]. In this work, we focus on DNN training performance on *GPUs* because they are (i) more commonly used by deep learning developers [20], and (ii) are readily available for purchase and rent [4, 21, 40, 42].

Common Performance Considerations

Although training is a conceptually simple procedure, it can be computationally-expensive, memory-intensive, and time-consuming in practice [14, 17, 37, 52, 58]. As a result, there are a number of common considerations for DNN training performance; we outline and discuss them in more detail below.

Training Throughput. The key metric for computational performance is the DNN's *training throughput*; it indicates how well the underlying hardware is being utilized [58]. Throughput measures how many data samples are processed per unit of time, and is therefore calculated by dividing the batch size by the time it takes to run one training iteration.

Batch Size. The *batch size* is one factor that affects the training throughput. Using a larger batch size may lead to higher training throughputs on some models, but usually with diminishing returns [14, 37, 58]. The batch size can also affect the *convergence* (i.e. final attainable prediction accuracy) of the model [32, 35, 37]. As a result, deep learning developers need to empirically adjust the batch size to balance computational performance and convergence. In this work, we focus on guiding the computational performance aspect of this tuning (i.e. selecting batch sizes that lead to increased throughput).

Memory Usage. Memory usage is an important performance consideration because a DNN generally needs to fit in the memory of the underlying hardware system. A DNN's training memory footprint consists primarily of memory allocated for the model's (i) *weights*, and (ii) *activations*. Activations are computed in each training iteration and are used to determine how to update the weights at the end of each iteration [20].

A natural first step in improving a DNN's computational training performance is discovering and understanding the DNN's performance bottlenecks. However, as described previously, this debugging process can still be a difficult endeavor for the typical deep learning developer.

PERFORMANCE UNDERSTANDING: FORMATIVE STUDY

To better understand how deep learning developers think about and approach computational performance, we conducted a formative study. We interviewed five participants who work with DNNs: three deep learning graduate student researchers (P1, 5 years of experience working with DNNs; P2, 4.5 years; P3, 4.5 years) and two deep learning practitioners in industry (P4, 4 years; P5, 4 years). Four interviews were in-person and one was remote. Each interview took up to 30 minutes to complete and each participant received a $25 gift card.

The purpose of our interviews was to learn (i) about the importance of computational training performance to deep learning developers, (ii) how developers currently deal with computational performance, and (iii) whether they face any challenges understanding and or debugging performance. To elicit this information, we asked each participant open-ended questions about computational performance. We began by asking them to describe if and how computational performance influences their models' designs. Then, to learn how they deal with performance, we asked them about the metrics they use to evaluate performance and how they obtain these metrics today. Finally, we asked them to describe any challenges they face when understanding and debugging performance. We summarize the key takeaways from our interviews below.

Performance Constrains Model Design

P1, P3, P4, and P5 all said that computational performance is important to them. P2 mentioned that performance is only a concern if it is noticeable, stating that otherwise the effort needed to find and debug performance issues would not be worth the time saved—alluding to the difficulty of debugging performance in the first place. P1 and P3 noted that performance is important because it *constrains how they design their models*. P3 specifically stated that performance affects which model designs they can experiment with due to the limitations of the computational resources they have available.

Participants Lack Useful Profiling Tools

The participants mentioned that, for performance understanding and debugging, they either (i) do not use any tools at all, (ii) manually instrument their code to measure run time, or (iii) rely on existing profiling tools that are difficult to use or lack features such as memory usage breakdowns.

P1 and P2 said that they do not use *any* tools for understanding performance and rather just *guess* based on what "feels slow." P2 went further to say that PyTorch, the software framework they use, hides performance details through its abstractions—making it difficult to debug performance. P3 noted that it is impossible to see breakdowns of a model's memory usage and training iteration run time in existing tools, which they felt to be useful metrics when investigating performance issues. P5 said that they resort to manually instrumenting parts of the code that they want to debug.

P4 was the most vocal about the lack of useful and easy to use DNN profiling tools. Since they use GPUs in their work, they experimented with using nvprof [43]: a general-purpose GPU computation performance debugger. They described their experience with nvprof as a "disaster" because it provided too much low-level GPU information. They had trouble connecting this low-level information to the high-level PyTorch code that they wrote. Their frustration was a result of existing performance tools working at *too low* of an abstraction level for typical deep learning developers.

Understanding Memory Usage is a Shared Concern

P1, P2, P3, and P5 said that understanding how memory is used in their models would be useful. P1 noted that their models tend to use a lot of memory and that it is difficult to determine what part of the model is responsible for the high memory usage. Similarly, P2 and P3 said that having a breakdown of their model's memory usage would be helpful, with P2 noting that a memory breakdown would help them decide how to shape their model's layers.

P1, P2, and P3 all mentioned that they experiment with different batch sizes during training to adjust their model's memory usage. P1 went further to say that they tune the size of their model's layers along with the batch size to fit within memory constraints, and that having visibility into their model's memory usage would help with this process.

Overall, these takeaways point to a lack of performance tools that (i) showcase *useful* performance information (e.g., memory usage breakdowns), (ii) are *easy to use*, and (iii) work at the *right abstraction level*. As a result, our interviewees resorted to either ignoring performance problems or expending significant effort to manually instrument their code.

SKYLINE

To help fill this need for new performance profiling tools, we present SKYLINE: a new interactive in-editor tool for DNN training on GPUs that supports performance profiling, visualization, and debugging. In this section, we begin by outlining SKYLINE's design goals and the special properties of DNN training computation that enable SKYLINE to achieve these goals. Then, we describe SKYLINE's features in detail.

Skyline Design Goals

SKYLINE's design is guided by the following goals, which encompass the insights from our formative study.

Goal 1 (G1): Appropriate Level of Abstraction

SKYLINE needs to present performance information at the right level of abstraction for everyday deep learning developers. These users develop their models in software frameworks (e.g., PyTorch) and are therefore most familiar with the abstractions used in these frameworks.

Goal 2 (G2): Code-Connected Insights

Performance profiling tools should help deep learning developers uncover performance issues in their code. This means that SKYLINE should make it easy for users to identify the relevant lines of code associated with any performance information it displays (e.g., visualizations of operation run times).

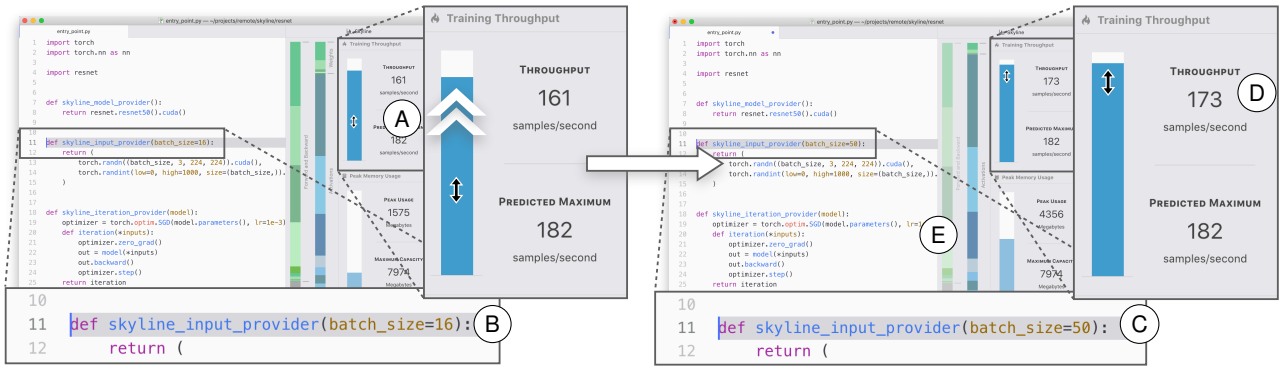

Figure 2. An example of SKYLINE's code-linked manipulatable key performance visualizations. (a) Users can drag the bar charts, e.g., upwards. (b) The initial batch size in the code. (c) Dragging the throughput bar leads to a new batch size that is predicted to result in the dragged throughput. (d) The new throughput, after dragging. (e) Other visualizations (memory breakdown and usage) update when the throughput bar is dragged, and vice versa.

Goal 3 (G3): Relevant Metrics

SKYLINE needs to focus on surfacing relevant metrics that deep learning developers care about for DNN training. These metrics include training throughput, iteration run time, and overall memory usage. This is because these metrics are directly affected by (i) the code that the user writes, and (ii) parameters that the user sets (e.g., the batch size).

What Makes DNN Training Computation Special?

Property 1 (P1): Fast and Repetitive Iterations

Although DNNs can take hours or days to train to an acceptable accuracy [14, 37, 58], the training process consists of repetitions of a *single* training iteration [20], which runs on the order of hundreds of *milliseconds* [58]. Additionally, for many DNNs, the computation in a training iteration depends only on the *size* of the inputs, which is fixed throughout training[3] (i.e. generally, the same "amount" of computation runs regardless of the values of the inputs). As a result, the computational performance of an entire training process can be characterized by the performance of just a few training iterations.

Property 2 (P2): Predictability for Batch Sizes

The performance (throughput, memory footprint) of a training iteration with respect to its batch size is predictable, as we describe and show in more detail in the System Implementation section. This property helps enable the directly manipulatable visualizations in SKYLINE.

Property 3 (P3): Structured Code

DNNs are built using software frameworks (e.g., PyTorch [46] or TensorFlow [1]), which offer a set of abstractions that *constrain* how a model can be implemented (e.g., all models in PyTorch are instances of a specific class). This means DNN code is written in a structured way, which helps SKYLINE connect performance visualizations to specific lines of code.

Overview

SKYLINE runs as a plugin in Atom [19] (a popular open-source editor) and supports DNNs implemented using PyTorch [46]. It runs within the editor, in a sidebar to the right of an opened file (Figure 1). Although we implemented SKYLINE to work with Atom and PyTorch, the ideas behind it are *not* fundamentally limited to this editor and framework combination. We discuss extensibility in more detail in our Discussion section.

To use SKYLINE, users first write an entry point file that contains special *provider functions* that tell SKYLINE how to run their model. We describe these functions in more detail in the Provider Functions section. Then, users start SKYLINE by navigating to the directory containing their entry point file and running a SKYLINE command in their terminal.

One of SKYLINE's key features is that it provides *live* performance feedback. When users make and save changes to their model in the editor, SKYLINE re-profiles their code in the background and updates the visualizations with the latest performance data. SKYLINE can offer live feedback because getting performance data about a model only requires profiling a few short training iterations (*P1*).

Interactive Key Performance Metrics

SKYLINE uses bar charts to visualize a model's training throughput and peak memory usage (*G1, G3*). Respectively, these charts show the current throughput and peak memory usage relative to the maximum achievable throughput (predicted) and memory capacity on the user's GPU (Figure 2(a)).

One of SKYLINE's novel contributions is the ability for the user to *directly manipulate* these bar charts (Figures 1(a) and 2). As the user drags the bars up or down (Figure 2(a)), SKYLINE makes *predictions* about the batch sizes that can be used during training to achieve these manipulated metrics. While the dragging occurs, SKYLINE also (i) *mutates* the user's code to actually set these predicted batch sizes (Figure 2(c)), and (ii) updates the other visualizations to account for the changed batch size (Figure 2(e)). For example, if the user drags the throughput bar upwards, SKYLINE will predict that a larger batch size needs to be used. SKYLINE would then update the memory footprint breakdown (described in the following section) to show that activations will comprise a larger proportion of the memory footprint. This feature is made possible by the predictability of DNN training computational performance and the structured nature of the code (*P2, P3*).

In addition to computational performance, the batch size used during training can also affect a model's final prediction ac-

[3]Language-based models (e.g., GNMT [55]) take variable-sized inputs because they operate on sentences. For these kinds of models, SKYLINE can use the longest sentence in the dataset as input during profiling to provide a lower-bound on the measured performance.

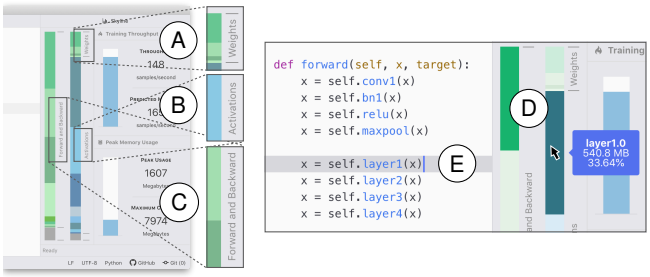

Figure 3. An overview of SKYLINE's interactive breakdowns. (a) Memory used by weights. (b) Memory used by activations. (c) Operation run times. (d) Breakdowns bars are linked; hovering over a memory bar will highlight the corresponding run time bar and vice versa. (e) Relevant line(s) of code are highlighted when hovering over visualizations.

curacy [32]. While the effects of the batch size on a model's quality are generally non-trivial to predict [49] (see our Discussion section), small changes to the batch size empirically do not seem to have a significant effect on a model's final prediction accuracy. To err on the side of caution, SKYLINE does not make model accuracy predictions and it does not automatically recommend batch sizes to use based on its performance predictions. SKYLINE instead offers these manipulatable visualizations to allow users to explore the computational performance trade-offs of different batch sizes. We envision this feature as a way for developers to become more informed about the performance implications of their batch size selections.

Interactive Breakdowns

SKYLINE also displays detailed breakdowns of the *training iteration run time* and *memory footprint*, visualizing them using stacked bar charts (Figure 3). The bars in each bar chart are sorted in descending order based on their size. These breakdowns indicate how each operation contributes to the iteration run time (Figure 3(c)) and memory footprint. The memory footprint breakdown is further split between the memory used for the activations and weights (Figures 3(a) and 3(b)).

Interactivity

When a user hovers over one of these bars, SKYLINE (i) highlights the relevant line of code associated with that bar (Figure 3(e)) (*G2, P3*), and (ii) reveals details about the bar: its run time or memory usage as a concrete number and as a percentage of the total iteration run time or memory usage (Figure 1(a)). If the user clicks the bar, SKYLINE will place their cursor on the bar's associated line of code and will also open the file containing that line of code in the editor.

Navigating the Breakdown Hierarchy

A DNN may contain hundreds of operations, which can be overwhelming when visualized all at once. To manage this complexity, SKYLINE organizes a DNN's operations into a *hierarchy of modules* where the top-level module represents the entire DNN. Each module is a "container" that can hold operations as well as other modules (analogous to a directory in a file system). SKYLINE builds this hierarchy by leveraging the observation that deep learning developers actually implement DNNs using modules; they are an abstraction used in PyTorch (*P3*). SKYLINE uses stack traces, recorded during profiling, to extract the hierarchy from the user's code; we discuss this process in more detail in the System Implementation section.

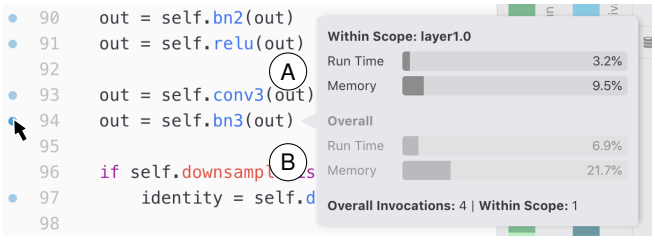

Figure 4. Inline markers stay in-sync with the current breakdown module being explored. (a) Run time and memory usage specific to module "`layer1`." (b) Performance details for all invocations of this line of code.

SKYLINE's breakdowns show the contents of one module at a time, starting with the top-level module representing the entire DNN (*G1*). Because of this design, some bars in the breakdowns will represent a module and will visualize the aggregate of all the operations inside it (e.g., the sum of all the run times of the operations in that module). When the user *double clicks* these bars, SKYLINE will "expand" the bar and visualize the operations *inside* that module. SKYLINE has buttons in its user interface that allow users to (i) go back one level up, and (ii) return to the top of the hierarchy.

These hierarchical breakdowns allow the user to *navigate* the performance profile of their model by drilling down to the specific parts of the model that they are interested in. For example, if a user wanted to investigate run time bottlenecks, they could double click the largest run time breakdown bar to drill down to the operations responsible for the large run time.

Inline Performance Markers

SKYLINE places inline markers in the editor gutter to indicate the lines of code (which correspond to operations in the model) for which performance information is available (Figure 1(a)). When hovering over one of these markers, SKYLINE displays detailed information about that line of code: the amount it contributes to the iteration run time and memory footprint. These visualizations are also linked to the state of the breakdowns. For example, if the user double clicks a breakdown bar to view the contents of a module, the inline markers will show performance details about that specific module as well (Figure 4). This linking is useful because modules are defined once in the code but may be used multiple times in a DNN.

Having both inline markers and breakdowns allows for two different approaches to performance investigation. If the user does not know what their model's bottlenecks are, they can use the breakdowns to explore and drill down to specific parts of their model. But, if they already have concerns about specific lines of code, they can navigate directly to that code and inspect the inline markers for detailed performance information.

Provider Functions

SKYLINE gathers performance data about the user's model by actually *running* the model on a set of chosen inputs. For this to work, users need to implement three SKYLINE *provider functions*. In a nutshell, these providers are

- *Model Provider:* Returns an instance of the user's model.
- *Input Provider:* Returns inputs for the user's model.
- *Iteration Provider:* Returns a function that, when invoked, runs a single training iteration.

SKYLINE uses these providers, which are implemented in Python, to run the user's model during profiling. We designed the providers to be easy to implement for a user who is familiar with the model's code. We also evaluated the difficulty of implementing these providers in our user study (see the User Evaluation section).

SYSTEM IMPLEMENTATION

Software Architecture

SKYLINE consists of two components: (i) a *plugin* that runs in Atom that displays all of SKYLINE's visual and interactive features, and (ii) a *daemon process* that performs the actual performance profiling. The two components communicate over a network socket using a SKYLINE-specific protocol. This architecture enables users to do their development on one machine (e.g., a laptop running Atom with the SKYLINE plugin) while profiling their code on a different machine (e.g., a virtual machine in the cloud).[4]

Gathering Performance Data

Each time the user modifies and saves their code, SKYLINE runs their model in the background to profile it and gather performance data (throughput, memory usage, operation run times). SKYLINE "monkey patches" PyTorch operations with wrappers that allow it to intercept and keep track of all the operations that run in one training iteration, as they are executed.

Profiling Time. The time it takes for this profiling process to complete depends on the DNN, batch size, and underlying GPU. For example, with the experimental setup described in the Evaluation of Prediction Accuracy section, profiling a feed-forward DNN with three hidden layers would take less than one second whereas profiling the Transformer would take up to 29 seconds. SKYLINE's profiling runs in the background and does *not* interfere with the user's ability to write code in their editor. If desired, users can also disable this "re-profile on save" feature and instead manually trigger profiling by clicking a button on SKYLINE's user interface.

Measuring Throughput. To measure the training throughput, SKYLINE measures the time it takes to run three training iterations, to account for any variance in the run times. SKYLINE then computes the throughput by dividing three times the input batch size by the measured run time. SKYLINE uses the user-written provider functions to extract the user's desired batch size and to run the user's training iteration code.

Measuring Operation Run Times. When SKYLINE intercepts an operation, it runs the operation independently to measure its run time. This approach allows SKYLINE to (i) decompose the overall iteration run time into individual operations, and (ii) amortize any profiling overhead by running each operation multiple times. These measured run times are used by the run time breakdown stacked bar chart visualization.

Measuring Memory. SKYLINE explicitly tracks memory allocated for the model's (i) *weights*, and (ii) computed *activations*. SKYLINE tracks the memory allocated for the weights by intercepting all weight creations when a model is instantiated. The memory used for activations is allocated as each DNN operation runs during a training iteration. To track this memory, SKYLINE records the additional memory consumed by each operation after it runs. These measurements are used by the memory breakdown stacked bar chart visualization.

Untracked Run Time and Memory. SKYLINE only explicitly tracks run times and memory allocations that are attributed to weights or operations in a DNN. For completeness, the remaining run time and memory usage are displayed as "untracked" bars in the run time and memory breakdown. These untracked bars represent auxiliary tasks that run during a training iteration (e.g., weight updates, memory allocated by the underlying framework for bookkeeping). They typically do not comprise a significant proportion of the overall iteration run time and memory usage (less than 20% in our use).

Code References. When an operation is executed, the state of the call stack contains all the relevant lines of code leading up to running the operation. Thus, when an operation is intercepted, SKYLINE also records the file and line number for each frame in the call stack. This allows SKYLINE to connect the operation's performance data to the relevant line(s) of code.

Hierarchical Breakdowns. SKYLINE uses the *stack trace* associated with each performance measurement to assemble the measurements into a hierarchy that mirrors the modules used in the code. This approach works because (i) all the operations in the model share a common stack frame, which is where the model is invoked in the code; and (ii) operations in different modules are defined in different functions, meaning any two operations in different modules will always appear in different stack frames. SKYLINE assembles all the stack traces into a tree where nodes represent operations or modules. If a node has children, they represent the operations and modules contained inside that node. SKYLINE's breakdowns visualize the children of one node at a time, starting with the root node (i.e. the operations and modules in the top-level module). Leaf nodes always represent the operations in the model.

Making Performance Predictions

SKYLINE's interactive features link a DNN's throughput and memory usage to its batch size using predictive models. These predictive models help users tune their DNN's batch size when they drag the throughput or memory usage visualizations (Figure 1(a)). To stay up-to-date, SKYLINE builds new predictive models each time the user modifies and saves their code.

Throughput

Let x represent the batch size and let a and b be constants. SKYLINE models the training throughput, $T(x)$, and iteration run time, $R(x)$, using

$$T(x) = \frac{x}{R(x)} \tag{1}$$

$$R(x) = ax + b \tag{2}$$

The intuition behind this model is that, as the underlying hardware is saturated, increasing the number of inputs to process will result in a linear increase in the amount of time it takes process them ($R(x)$). In the limit (i.e. as x approaches infinity),

[4]This remote setup is necessary when users need to train models on remote resources (e.g., a shared GPU compute cluster). SKYLINE also supports developing and profiling on the same machine.

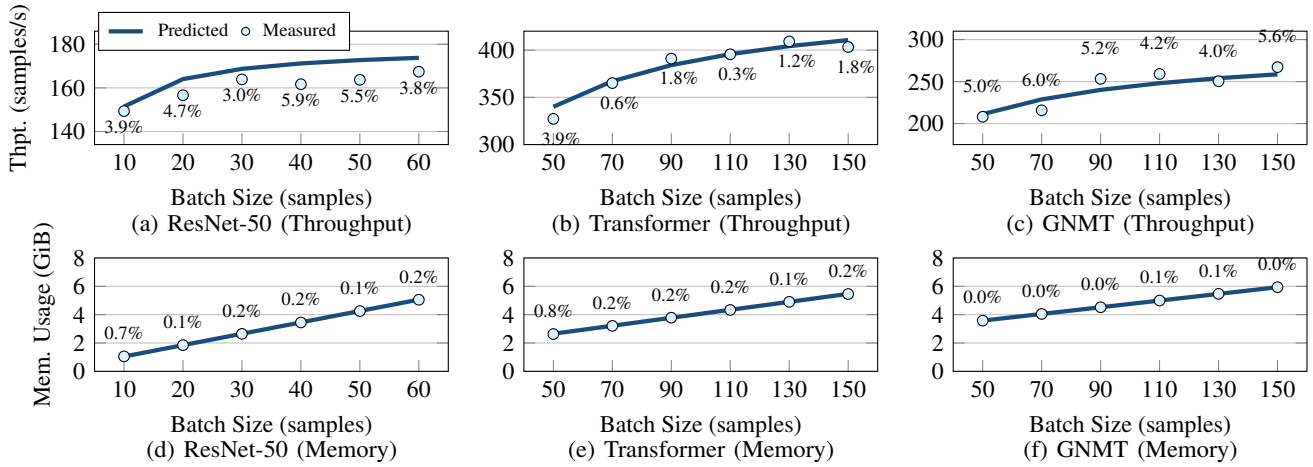

Figure 5. An evaluation of SKYLINE's predictive models for training throughput and memory usage. Each graph depicts SKYLINE's throughput or memory usage prediction with a solid line. The prediction error for specific batch sizes is shown as a percentage next to each measured point.

the throughput approaches $1/a$, which is what SKYLINE uses to predict the maximum attainable throughput.

SKYLINE selects a and b empirically by measuring $R(x)$ for three different values of x (i.e. three different batch sizes) and then applying least squares linear regression [9]. SKYLINE uses the batch size set by the user in the input provider function as the first batch size, and then selects the other two batch sizes by adding a constant to the previously sampled batch size. If a selected batch size results in running out of memory, SKYLINE will adjust and select smaller batch sizes instead.[5]

Memory Usage

Let x represent the batch size and let c and d be constants. SKYLINE models the overall memory usage, $M(x)$, using

$$M(x) = cx + d \tag{3}$$

The intuition behind this model is that the memory used by the computed activations is directly proportional to the batch size because there exist activations for each sample in the batch. Just like for the throughput model, SKYLINE selects c and d empirically by measuring $M(x)$ for three different values of x and then applying least squares linear regression. SKYLINE applies the same strategy for selecting batch sizes to sample as for the throughput model.

Using the Predictive Models

SKYLINE uses these models when the user manipulates the key performance metric bar charts (throughput and memory usage). For example, when the user drags the memory usage bar chart, SKYLINE converts the distance dragged into a new memory usage value. SKYLINE then translates this memory usage into a batch size and throughput prediction by using the memory predictive model followed by the throughput predictive model. The throughput prediction is then used to update the throughput bar chart. SKYLINE takes a similar approach when the user drags the throughput bar chart.

[5]In the rare case where a predictive model cannot be built because the user's DNN is too large to support three different batch sizes, the ability to drag the throughput and memory bar charts is disabled.

To mutate the code, SKYLINE uses Python's built-in abstract syntax tree parser to identify the line of code that contains the input provider function's signature. The user defines the batch size to use by setting a keyword argument in the input provider's signature. SKYLINE mutates this line of code by updating the signature to reflect the new predicted batch size.

Evaluation of Prediction Accuracy

We empirically evaluate the effectiveness of these predictive models by measuring their *prediction errors* (lower is better) for training throughput and memory usage across three different DNNs and six different batch sizes.

Experimental Setup. We run our experiments on a machine equipped with (i) an AMD Ryzen TR 1950X 3.4 GHz 16-core CPU [2], (ii) 16 GB of DDR4 main memory [38], and (iii) an NVIDIA GeForce RTX 2070 GPU [41] with 8 GB of GDDR6 memory [39]. We use PyTorch 1.3.1 [46] and CUDA 10.1 [45], running on Ubuntu 18.04 [11].

Methodology. We use SKYLINE to build predictive models for training throughput and memory usage for ResNet-50 [25], the Transformer [54], and GNMT [55], starting from three different batch sizes for each model (8, 16, and 32 for ResNet-50; 32, 64, and 80 for the Transformer and GNMT). These models are used in well-known DNN benchmarking suites including MLPerf [37] and TBD [58]. The ResNet-50 model is configured for the ImageNet dataset [48] and the Transformer and GNMT are configured for the WMT'16 (EN-DE) dataset [10], with fixed sentences of length 25. We use synthetic training data for all three models because we are evaluating the *computational* performance of the models, which we verified to not depend on the values of the input data.

We then use the predictive models to make training throughput and memory usage predictions for six other batch sizes (shown in Figure 5). We compare the predictions against the measured throughput and memory usage. When measuring the throughput and memory usage, we repeat each measurement five times and use the median measurement. To amortize any measurement overhead when measuring the throughput, we

record the time it takes to run three training iterations and divide three times the batch size by this run time.

Results. Figure 5 shows a summary of SKYLINE's prediction accuracy results. Since we create three predictive models for each DNN (starting from different batch sizes) and make predictions using them for all six different batch sizes, we average the prediction values and prediction errors across all three predictive models when plotting them in Figure 5. Next to each measured point, we show the average prediction error for that batch size. The standard deviations of all our throughput and memory usage measurements are under 1.77 samples/s and 0.00949 GiB respectively.

From this figure we can draw two major conclusions. First, SKYLINE makes *accurate* throughput and memory usage predictions. The average error for throughput and memory usage among the three DNNs we evaluate is 3.7% and 0.19% respectively (maximum 10.% and 1.1%). Second, SKYLINE's predictive models work across a *diverse* set of DNN architectures and tasks. ResNet-50 is a convolutional neural network used for image classification, the Transformer is an attention-based DNN used for translation, and GNMT is a recurrent neural network also used for translation.

USER EVALUATION

To evaluate SKYLINE, we conducted a qualitative user study examining SKYLINE's diagnostic capabilities. SKYLINE is not designed for novice deep learning developers, and so we required the participants to be experienced with PyTorch (the deep learning framework that SKYLINE supports). This requirement made it difficult to recruit a large number of qualified participants. We recruited seven participants (six male), all deep learning graduate student researchers that train DNNs in their day-to-day work. The participants were between the ages of 23 and 30 (median 24), and had one to three years of experience with PyTorch (median 2.5 years). Six of the participants were affiliated with one institution, and the seventh was affiliated with a different institution. Two participants had also taken part in our formative study. We conducted our study sessions remotely using screen sharing software and compensated participants with a $25 gift card.

Study Design

We split our study into three parts: (i) a walkthrough of SKYLINE's features, (ii) a series of timed tasks, and (iii) a qualitative questionnaire about the usefulness and ease of use of SKYLINE. Each study session took up to one hour to complete.

Walkthrough. During the walkthrough, we introduced SKYLINE and demonstrated each of its features. We gave participants a chance to try each feature for themselves and to ask questions during this process. The full walkthrough took up to 20 minutes to complete and we conducted it with a ResNeXt-50 [56] model loaded in SKYLINE.

Timed Tasks. After the walkthrough, we asked the participants to complete five timed tasks:

1. Find the top-level module or operation that contributes the most to the iteration run time. If it is a module, repeat recursively.
2. Repeat the same as above, but for the module or operation that contributes the most to the memory footprint.
3. Determine whether the weights or activations contribute the most to the model's memory footprint.
4. Find the batch size at which the weights and activations take up an equal proportion of the overall memory footprint.
5. Determine whether increasing the batch size leads to increased training throughput but with diminishing returns. If it occurs, after which batch size?

		Mean Completion Time (seconds)				
Model	**Weights**	T1	T2	T3	T4	T5
GNMT	161 M	52.0	36.4	26.0	43.6	63.7
Transformer	94 M	47.4	49.6	6.4	21.7	32.0
ResNet-50	26 M	28.0	29.1	5.0	22.3	23.3

Table 2. Summary of mean completion times for tasks 1 to 5 (T1 – T5). For a sense of their size, we include the number of weights in each model.

We asked the participants to complete these five tasks for three different DNNs: (i) GNMT [55], (ii) the Transformer [54], and (iii) ResNet-50 [25]. We designed these tasks with two goals in mind: (i) to mimic common performance debugging tasks to evaluate the ease of using SKYLINE to accomplish them, and (ii) to evaluate whether the participants could independently determine which SKYLINE features to use for each task.

For the first two models, we provided implementations of the provider functions. To evaluate the difficulty of writing the providers, we asked the *participant* to implement the providers for the third model. We provided them with (i) a copy of the relevant documentation, and (ii) an example of the providers written for a different model. This is the same amount of information made available to all new SKYLINE users.

We did not compare the time it took to complete these tasks in SKYLINE against other existing profiling tools for DNN training. This is because SKYLINE is unique in its design; SKYLINE is an interactive in-editor tool whereas existing tools are either command-line-based [3, 44], not domain-specific [43], or are not interactive and do not offer comparable features (e.g., performance predictions, memory profiling) [3, 22, 43, 44]. As a result, it would be difficult or impossible for participants to accomplish these tasks using any existing profiling tools within the time allocated for the study.

Questionnaire. After completing the timed tasks, the participants filled out a 10-minute qualitative questionnaire. We asked participants to rate whether they found each of SKYLINE's features to be useful and easy to use. We also asked additional questions about SKYLINE as a whole, which we discuss in the Results section. We used a five-point Likert scale for each question ($1 \Rightarrow$ "Strongly Disagree", $5 \Rightarrow$ "Strongly Agree") where "Strongly Agree" was always the most positive option. For example, participants would have to select how strongly they agreed with the statement that *"Skyline's memory breakdown is a useful feature."*

Results

Overall, the results of our exploratory study were positive and encouraging. When qualitatively asked about SKYLINE as a whole, participants unanimously either agreed or strongly

	Useful		Easy to Use	
Feature	Mean	Median	Mean	Median
Throughput Bar Chart	4.4	4	4.3	4
Memory Use Bar Chart	4.4	4	4.4	4
Run Time Breakdown	4.4	4	4.7	5
Memory Breakdown	4.9	5	4.6	5
Inline Markers	4.0	4	4.7	5

Table 3. A summary of how useful and easy to use participants found SKYLINE's features, on a five-point Likert scale (higher is better).

agreed that SKYLINE would be useful (mean 4.7/5; median 5/5) and that it was easy to use (4.7/5; 5/5).

Timed Tasks

Table 2 shows a summary of the mean task completion time, in seconds, among all participants for each model and task. These results show that, on average, each task could be completed within roughly one minute. Our longest recorded time was 2 minutes and 34 seconds for task 4 on GNMT.

During the study, five out of the seven participants completed all the tasks independently. The two other participants knew "what" they needed to do, but needed reminders about how to use SKYLINE's features. For task 1, one participant knew that they needed to "drill down" to different parts of the model using the breakdowns, but forgot that they needed to double click on a breakdown bar to do so. For task 4, both participants knew that they needed to drag something to have SKYLINE make predictions, but needed a reminder to drag the memory usage and throughput bar charts. We feel that these results are encouraging given that, with just a brief 20-minute walk-through, (i) all the participants had some intuition about how to use SKYLINE to complete the debugging tasks, and (ii) most of the participants (5/7) completed the tasks independently.

Writing the Providers

On average, implementing the SKYLINE provider functions took the participants 7 minutes and 41 seconds (median 7 minutes and 52 seconds). The longest time was 12 minutes and 48 seconds; the shortest time was 3 minutes and 47 seconds. Of the seven participants, four implemented the providers independently whereas three needed help with minor syntax or PyTorch API issues (missing commas or keyword arguments, being unsure about the signature of a PyTorch function). When asked qualitatively about whether they agreed that writing the provider functions was easy, two participants strongly agreed, two agreed, and three were neutral. We believe that these results are still encouraging given that this was the first time each of the participants tried to implement the SKYLINE providers. Participant 1 commented that *"[the providers] were very little setup for the benefits of Skyline."* Similarly, participant 3 mentioned that implementing the providers was easier for them than the equivalent setup they had to perform to use nvprof (a general-purpose GPU performance profiler).

Qualitative Impressions

We first asked users to rate SKYLINE's features based on whether they are useful and easy to use; Table 3 summarizes our results. While the participants found all the features to be useful, these results also seem to indicate that the memory breakdown was the most useful. Additionally, the participants generally found each feature to be easy to use.

We then asked participants about the usefulness of the different interactions offered by SKYLINE. These interactions were (i) highlighting the relevant line(s) of code when hovering over a visualization (mean 4.6; median 5), (ii) updating the visualizations when the code is changed (4.6; 5), (iii) double clicking breakdown bars to navigate the hierarchy (4.9; 5), and (iv) making batch size predictions using the draggable visualizations (3.9; 4). We also asked the participants whether they agreed that finding run time and memory bottlenecks would be easy with SKYLINE. These two questions also elicited positive responses, both receiving scores of 4.7 on average (median 5).

We believe that one factor that affected the participants' perceptions of the draggable visualizations was that they were difficult to drag, due to an implementation issue (the user's cursor needed to remain inside the bar chart for any manipulations to take effect). Several participants ran into this issue but were able to perform the dragging after being told to keep their cursor inside the bar chart. We have since fixed this problem.

Finally, we asked the participants how strongly they agreed with two statements about SKYLINE as a whole: (i) *"By being present inside the text editor, Skyline could help me proactively develop my models with computational performance in mind"* (mean 4.1, median 4), and (ii) *"Skyline's interactive features could help me better understand the computational training performance of my models"* (mean 4.7, median 5). We believe that these results are encouraging and pave the way for further research toward improving performance understanding through interactivity.

Overall, the participants made positive comments throughout the study. Participant 1 remarked that *"[Skyline] is very intuitive"* and that getting a memory breakdown *"[...] is usually very hard to do."* Participant 2 mentioned that using SKYLINE was educational because they learned about the run times and memory characteristics of various DNNs. Furthermore, three out of the seven participants also voluntarily commented that they would be interested in using SKYLINE in their own work.

DISCUSSION

While the results of our study are promising, there are also limitations and opportunities for additional research (and extensions of SKYLINE) that warrant further discussion.

Limitations and Future Work

Model Quality. An ideal development tool for DNN training would bring together diagnostics and predictions for both (i) computational performance, *and* (ii) a model's prediction accuracy. This set of features would empower deep learning developers to evaluate the trade-offs between a model's computational performance and accuracy during development. However, SKYLINE focuses only on the former as it does not support making predictions about a DNN's potential prediction accuracy. This is because predicting how a DNN's design (and its batch size) affects its accuracy is a difficult problem; it boils down to understanding the factors that affect a DNN's ability to "learn," which is still an active area of research within the machine learning community [16, 18]. We

see SKYLINE as one step toward achieving this vision for ideal deep learning development tools and we think that exploring ways to integrate SKYLINE with prior work on deep learning accuracy diagnostics [30, 33, 51] would be a great opportunity for future work.

User Study. Our user evaluation was an exploratory qualitative study that focused on SKYLINE's diagnostic capabilities. As a result, we do not make any claims about (i) whether SKYLINE is a more effective performance debugging tool when compared to existing commercial profiling tools, nor (ii) how SKYLINE's debugging capabilities might influence a user's DNN design process. The goal of our study was to elicit early qualitative feedback from deep learning developers about the effectiveness of in-editor, interactive performance visualizations for DNN training. Our encouraging results indicate that SKYLINE's interactive features may be a promising way to communicate complex performance information to deep learning developers. As described above, we see SKYLINE as one step toward a comprehensive DNN training development tool and we think that our results open up opportunities for further research studies that can examine how having in-editor diagnostics for both performance and accuracy might influence a user's DNN design process.

Extensibility

Although we implemented SKYLINE to work with Atom and PyTorch and to make mutations to a DNN's batch size, the ideas behind SKYLINE are not fundamentally limited to this editor, framework, and model parameter combination.

Supporting Other Frameworks. To gather and organize performance data, SKYLINE relies on two concepts: (i) *execution graphs* (a representation of the operations that run during a training iteration, along with their dependencies), and (ii) code "modules." SKYLINE uses execution graphs to profile the individual operations in a model and uses modules in the breakdown hierarchy. Both of these concepts exist in other popular software frameworks. For example, both TensorFlow [1] and MXNet [12] use graphs to represent DNNs internally and both frameworks have the notion of "modules" (`tf.keras.Model` in TensorFlow and `gluon.nn.Block` in MXNet).

Supporting Other Editors. Since SKYLINE consists of an editor plugin and a daemon process that communicate over a network socket, supporting additional editors is just a matter of implementing a plugin for that specific editor. Any new editors just need to provide APIs to allow the new SKYLINE plugin to (i) render user interface elements, (ii) open files, and (iii) mutate the contents of opened files.

Mutating Other Model Parameters. Supporting code mutations to other model parameters (e.g., the size of a layer) would consist of three tasks: (i) developing performance models that relate the parameter to the model's training throughput and memory usage, (ii) capturing the relationship between inputs of successive layers (e.g., changing the size of one layer may alter the size of its output, which in turn affects the input size of the next layer in the DNN), and (iii) identifying how the parameter is specified in the code. Task (i) can be approached using regression methods similar to SKYLINE's batch size performance models. Task (ii) is a matter of implementing the well-defined mathematical rules that relate a DNN layer's size to its expected input and output sizes. Task (iii) can be approached using the code's abstract syntax tree because the way DNN parameters are specified is ultimately imposed by the software framework (e.g., PyTorch).

Additional Use Cases

In addition to being a performance profiling and debugging tool, we believe that SKYLINE has the potential to help users in other ways.

Proactive Model Design. SKYLINE shortens the performance feedback loop by displaying computational performance information live during development. We think that this rapid feedback could enable deep learning developers to be more *proactive* when designing their models for performance because they would be able to quickly experiment with and iterate on model designs during development.

Education. SKYLINE's interactive features could also be used to help teach deep learning developers about the performance characteristics of DNN training. For example, users can drag the throughput bar chart to see the relationship between batch size, throughput, and memory usage (e.g., throughput tends to increase with larger batch sizes, but with diminishing returns). They can also use SKYLINE's breakdowns to see the performance costs of different operations instead of guessing about performance—a common approach used by our interviewees.

CONCLUSION

This paper presents SKYLINE: a new interactive tool for DNN training that supports in-editor performance profiling, visualization, and debugging. The key idea behind SKYLINE is that it leverages special properties of DNN training (repetitiveness, predictability, structured code) to offer interactive performance visualizations tied to the code. The main takeaways from this work are that (i) DNN training computational performance debugging is an important problem faced by deep learning developers, and (ii) DNN training computation has useful *properties* that can enable new interactive features (e.g., directly manipulatable visualizations that mutate the batch size in the code). Finally, we have open-sourced SKYLINE at github.com/skylineprof/skyline to help deep learning developers and to hopefully facilitate future research in this area.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their feedback. We also thank the participants in our formative and user studies for providing their insights and time. We are grateful to James Gleeson and Hongyu Zhu for providing feedback on earlier versions of the SKYLINE project, as well as to all members of the EcoSystem research group for the stimulating research environment they provide. This work was supported by a Queen Elizabeth II Graduate Scholarship in Science and Technology, Vector Scholarship in Artificial Intelligence, Snap Research Scholarship, and an NSERC Canada Graduate Scholarship – Master's (CGS M). This work was also supported in part by the NSERC Discovery grant, the Canada Foundation for Innovation JELF grant, the Connaught Fund, and Huawei grants.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*.

[2] Advanced Micro Devices, Inc. 2017. AMD Ryzen Threadripper 1950X Processor. (2017). https://www.amd.com/en/products/cpu/amd-ryzen-threadripper-1950x.

[3] Aditya Agrawal and Marek Kolodziej. 2019. PyProf – PyTorch Profiling Tool. (2019). https://github.com/NVIDIA/PyProf.

[4] Amazon, Inc. 2020. Amazon EC2 P3 Instances. (2020). https://aws.amazon.com/ec2/instance-types/p3.

[5] Saleema Amershi, Max Chickering, Steven Drucker, Bongshin Lee, Patrice Simard, and Jina Suh. 2015. ModelTracker: Redesigning Performance Analysis Tools for Machine Learning. In *Proceedings of the 2015 CHI Conference on Human Factors in Computing Systems (CHI'15)*.

[6] Dario Amodei, Rishita Anubhai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Jingdong Chen, Mike Chrzanowski, Adam Coates, Greg Diamos, Erich Elsen, Jesse Engel, Linxi Fan, Christopher Fougner, Tony Han, Awni Y. Hannun, Billy Jun, Patrick LeGresley, Libby Lin, Sharan Narang, Andrew Y. Ng, Sherjil Ozair, Ryan Prenger, Jonathan Raiman, Sanjeev Satheesh, David Seetapun, Shubho Sengupta, Yi Wang, Zhiqian Wang, Chong Wang, Bo Xiao, Dani Yogatama, Jun Zhan, and Zhenyao Zhu. 2015. Deep Speech 2: End-to-End Speech Recognition in English and Mandarin. *arXiv* abs/1512.02595 (2015). https://arxiv.org/abs/1512.02595

[7] Dario Amodei, Danny Hernandez, Girish Sastry, Jack Clark, Greg Brockman, and Ilya Sutskever. 2018. AI and Compute. Blog Post. (2018). https://openai.com/blog/ai-and-compute

[8] Fabian Beck, Oliver Moseler, Stephan Diehl, and Günter Daniel Rey. 2013. In Situ Understanding of Performance Bottlenecks Through Visually Augmented Code. In *Proceedings of the 2013 IEEE 21st International Conference on Program Comprehension (ICPC'13)*.

[9] Christopher M. Bishop. 2006. *Pattern Recognition and Machine Learning*. Springer.

[10] Ondřej Bojar, Rajen Chatterjee, Christian Federmann, Yvette Graham, Barry Haddow, Matthias Huck, Antonio Jimeno Yepes, Philipp Koehn, Varvara Logacheva, Christof Monz, Matteo Negri, Aurelie Neveol, Mariana Neves, Martin Popel, Matt Post, Raphael Rubino, Carolina Scarton, Lucia Specia, Marco Turchi, Karin Verspoor, and Marcos Zampieri. 2016. Findings of the 2016 Conference on Machine Translation. In *Proceedings of the First Conference on Machine Translation (WMT'16)*.

[11] Canonical Ltd. 2018. Ubuntu 18.04 LTS (Bionic Beaver). (2018). http://releases.ubuntu.com/18.04.

[12] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2016. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. In *Proceedings of the 2016 NeurIPS Workshop on Machine Learning Systems*.

[13] Jürgen Cito, Philipp Leitner, Martin Rinard, and Harald C. Gall. 2019. Interactive Production Performance Feedback in the IDE. In *Proceedings of the 41st International Conference on Software Engineering (ICSE'19)*.

[14] Cody Coleman, Deepak Narayanan, Daniel Kang, Tian Zhao, Jian Zhang, Luigi Nardi, Peter Bailis, Kunle Olukotun, Chris Ré, and Matei Zaharia. 2017. DAWNBench: An End-to-End Deep Learning Benchmark and Competition. In *Proceedings of the 2017 NeurIPS Workshop on Machine Learning Systems*.

[15] Paul Covington, Jay Adams, and Emre Sargin. 2016. Deep Neural Networks for YouTube Recommendations. In *Proceedings of the 10th ACM Conference on Recommender Systems (RecSys'16)*.

[16] Xiaoliang Dai, Peizhao Zhang, Bichen Wu, Hongxu Yin, Fei Sun, Yanghan Wang, Marat Dukhan, Yunqing Hu, Yiming Wu, Yangqing Jia, Peter Vajda, Matt Uyttendaele, and Niraj K. Jha. 2019. ChamNet: Towards Efficient Network Design through Platform-Aware Model Adaptation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR'19)*.

[17] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics (NAACL'19)*.

[18] Jonathan Frankle and Michael Carbin. 2019. The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks. In *Proceedings of the 7th International Conference on Learning Representations (ICLR'19)*.

[19] GitHub, Inc. 2020. Atom: A Hackable Text Editor for the 21st Century. (2020). https://atom.io.

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

[21] Google, Inc. 2020a. GPUs on Compute Engine. (2020). https://cloud.google.com/compute/docs/gpus.

[22] Google, Inc. 2020b. TensorFlow Profiler. (2020). https://github.com/tensorflow/profiler.

[23] Philip J. Guo. 2013. Online Python Tutor: Embeddable Web-based Program Visualization for CS Education. In *Proceedings of the 44th ACM Technical Symposium on Computer Science Education (SIGCSE'13)*.

[24] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. 2017. Mask R-CNN. *arXiv* abs/1703.06870 (2017). http://arxiv.org/abs/1703.06870

[25] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR'16)*.

[26] Jane Hoffswell, Arvind Satyanarayan, and Jeffrey Heer. 2018. Augmenting Code with In Situ Visualizations to Aid Program Understanding. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems (CHI'18)*.

[27] Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q Weinberger. 2017. Densely Connected Convolutional Networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR'17)*.

[28] Paras Jain, Ajay Jain, Aniruddha Nrusimha, Amir Gholami, Pieter Abbeel, Kurt Keutzer, Ion Stoica, and Joseph E. Gonzalez. 2020. Checkmate: Breaking the Memory Wall with Optimal Tensor Rematerialization. In *Proceedings of Machine Learning and Systems 2020 (MLSys'20)*.

[29] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA'17)*.

[30] Minsuk Kahng, Pierre Y. Andrews, Aditya Kalro, and Duen Horng Chau. 2018. ActiVis: Visual Exploration of Industry-Scale Deep Neural Network Models. *IEEE Transactions on Visualization and Computer Graphics (TVCG'18)* 24, 1 (2018), 88–97.

[31] Hyeonsu Kang and Philip J. Guo. 2017. Omnicode: A Novice-Oriented Live Programming Environment with Always-On Run-Time Value Visualizations. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology (UIST'17)*.

[32] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. 2017. On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima. In *Proceedings of the 5th International Conference on Learning Representations (ICLR'17)*.

[33] Josua Krause, Adam Perer, and Kenney Ng. 2016. Interacting with Predictions: Visual Inspection of Black-box Machine Learning Models. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems (CHI'16)*.

[34] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems 25 (NeurIPS'12)*.

[35] Yann A. LeCun, Léon Bottou, Genevieve B. Orr, and Klaus-Robert Müller. 2012. *Efficient BackProp*. Springer Berlin Heidelberg. 9–48 pages. DOI: http://dx.doi.org/10.1007/978-3-642-35289-8_3

[36] Tom Lieber, Joel R. Brandt, and Rob C. Miller. 2014. Addressing Misconceptions about Code with Always-on Programming Visualizations. In *Proceedings of the 2014 CHI Conference on Human Factors in Computing Systems (CHI'14)*.

[37] Peter Mattson, Christine Cheng, Cody Coleman, Greg Diamos, Paulius Micikevicius, David Patterson, Hanlin Tang, Gu-Yeon Wei, Peter Bailis, Victor Bittorf, David Brooks, Dehao Chen, Debojyoti Dutta, Udit Gupta, Kim Hazelwood, Andrew Hock, Xinyuan Huang, Bill Jia, Daniel Kang, David Kanter, Naveen Kumar, Jeffery Liao, Guokai Ma, Deepak Narayanan, Tayo Oguntebi, Gennady Pekhimenko, Lillian Pentecost, Vijay Janapa Reddi, Taylor Robie, Tom St. John, Carole-Jean Wu, Lingjie Xu, Cliff Young, and Matei Zaharia. 2019. MLPerf Training Benchmark. In *Proceedings of Machine Learning and Systems 2020 (MLSys'20)*.

[38] Micron Technology, Inc. 2014. DDR4. (2014). https://www.micron.com/products/dram/ddr4-sdram.

[39] Micron Technology, Inc. 2017. GDDR6. (2017). https://www.micron.com/products/graphics-memory/gddr6.

[40] Microsoft, Inc. 2020. GPU optimized virtual machine sizes. (2020). https://docs.microsoft.com/en-us/azure/virtual-machines/sizes-gpu.

[41] NVIDIA Corporation. 2018a. NVIDIA GeForce RTX 2070. (2018). https://www.nvidia.com/en-us/geforce/graphics-cards/rtx-2070.

[42] NVIDIA Corporation. 2018b. *NVIDIA Turing Architecture*. Whitepaper. NVIDIA. https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf.

[43] NVIDIA Corporation. 2019. NVIDIA Profiler User's Guide. (2019). https://docs.nvidia.com/cuda/profiler-users-guide/index.html.

[44] NVIDIA Corporation. 2020a. DLProf User Guide. (2020). https://docs.nvidia.com/deeplearning/frameworks/dlprof-user-guide.

[45] NVIDIA Corporation. 2020b. NVIDIA CUDA Toolkit. (2020). https://developer.nvidia.com/cuda-toolkit.

[46] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32 (NeurIPS'19)*.

[47] Kayur Patel, Naomi Bancroft, Steven M. Drucker, James Fogarty, Amy J. Ko, and James Landay. 2010. Gestalt: Integrated Support for Implementation and Analysis in Machine Learning. In *Proceedings of the 23rd Annual ACM Symposium on User Interface Software and Technology (UIST'10)*.

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

[49] Christopher J. Shallue, Jaehoon Lee, Joseph Antognini, Jascha Sohl-Dickstein, Roy Frostig, and George E. Dahl. 2019. Measuring the Effects of Data Parallelism on Neural Network Training. *Journal of Machine Learning Research (JMLR)* 20, 112 (2019), 1–49.

[50] Karen Simonyan and Andrew Zisserman. 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *Proceedings of the 3rd International Conference on Learning Representations (ICLR'15)*.

[51] Hendrik Strobelt, Sebastian Gehrmann, Michael Behrisch, Adam Perer, Hanspeter Pfister, and Alexander M. Rush. 2019. Seq2Seq-Vis: A Visual Debugging Tool for Sequence-to-Sequence Models. *IEEE Transactions on Visualization and Computer Graphics (TVCG'19)* 25, 1 (2019), 353–363.

[52] Emma Strubell, Ananya Ganesh, and Andrew McCallum. 2019. Energy and Policy Considerations for Deep Learning in NLP. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics (ACL'19)*.

[53] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. 2014. Sequence to Sequence Learning with Neural Networks. In *Proceedings of Advances in Neural Information Processing Systems 27 (NeurIPS'14)*.

[54] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is All You Need. In *Advances in Neural Information Processing Systems 30 (NeurIPS'17)*.

[55] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Łukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean. 2016. Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation. *arXiv* abs/1609.08144 (2016). http://arxiv.org/abs/1609.08144

[56] Saining Xie, Ross Girshick, Piotr Dollar, Zhuowen Tu, and Kaiming He. 2017. Aggregated Residual Transformations for Deep Neural Networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR'17)*.

[57] Bojian Zheng, Nandita Vijaykumar, and Gennady Pekhimenko. 2020. Echo: Compiler-Based GPU Memory Footprint Reduction for LSTM RNN Training. In *Proceedings of the 47th Annual International Symposium on Computer Architecture (ISCA'20)*.

[58] Hongyu Zhu, Mohamed Akrout, Bojian Zheng, Andrew Pelegris, Amar Phanishayee, Bianca Schroeder, and Gennady Pekhimenko. 2018. Benchmarking and Analyzing Deep Neural Network Training. In *Proceedings of the 2018 IEEE International Symposium on Workload Characterization (IISWC'18)*.