Data Management and Visualization for Benchmarking Deep Learning Training Systems

Ties Robroek, Aaron Duane, Ehsan Yousefzadeh-Asl-Miandoab, Pınar Tözün
IT University of Copenhagen, Copenhagen, Denmark
titr,aadu,ehyo,pito@itu.dk

ABSTRACT

Evaluating hardware for deep learning is challenging. The models can take days or more to run, the datasets are generally larger than what fits into memory, and the models are sensitive to interference. Scaling this up to a large amount of experiments and keeping track of both software and hardware metrics thus poses real difficulties as these problems are exacerbated by sheer experimental data volume. This paper explores some of the data management and exploration difficulties when working on machine learning systems research. We introduce our solution in the form of an open-source framework built on top of a machine learning lifecycle platform. Additionally, we introduce a web environment for visualizing and exploring experimental data.

ACM Reference Format:
Ties Robroek, Aaron Duane, Ehsan Yousefzadeh-Asl-Miandoab, Pınar Tözün. 2023. Data Management and Visualization for Benchmarking Deep Learning Training Systems. In *Data Management for End-to-End Machine Learning (DEEM '23), June 18, 2023, Seattle, WA, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3595360.3595851

1 INTRODUCTION

Deep learning has become a staple in data science. Large deep learning models provide state-of-the-art functionality solving many problems not solvable by conventional algorithms [1, 2, 3]. Models need to be trained before being deployed in production. This training is an expensive iterative process in which the model iterates over a dataset multiple times. The growth in deep learning has been paired with an exponential growth in model and dataset size. More powerful and optimized hardware is required to facilitate the training of such models. This, in addition to the increase in training times, has inflated the resource requirements of deep learning training to a level where it can no longer be ignored.

GPUs are the de facto commodity hardware for meeting the resource requirements of deep learning. Today's GPUs are significantly more powerful than those of ten years ago. In order to improve the utilization of hardware resources it is paramount that we use GPUs to their maximum potential. This requires tuning the training process to properly fit the hardware. On the other hand, a problem may not have a large enough dataset to warrant a large model, or the ideal training setup for a model might not be able to utilize all of the GPU resources [4, 5, 6]. This poses an issue when training neural networks as this process usually takes exclusive access to a GPU. This may lead to resource wastage as the model may not be large enough to saturate the GPU. As the scale of the hardware increases, underutilization of hardware resources becomes a serious consideration for data scientists training deep learning models. All these issues underline the need for performing systematic experiments that evaluate the impact of certain configurations on deep learning training and hardware utilization.

In the machine learning training space there has been considerable work to provide insights into the training process. Techniques to improve model selection are focused on e.g. model accuracy instead of hardware utilization [7]. Platforms such as WandB [8] and MLFlow [9] provide extensive tracking and management functionality, but their hardware monitoring is limited. MLOps tools like Polyaxon [10] and Kubeflow [11] provide a solution for deploying training tasks on clusters and may log hardware metrics if the user wants them to, but are not specifically designed for keeping and exploring detailed benchmarking data with hardware metrics. Umlaut [12] provides accessible and flexible metric collection, but does not offer GPU metrics. Finally, automated machine learning offers a variety of exploration tools [13, 14], though again focusing on model accuracy.

In this paper, our goal is to build and demonstrate a framework that aids data scientists and machine learning systems researchers when performing systematic experiments that also takes hardware into account. We have identified six requirements and challenges for such a framework. Firstly, in order to provide a rigorous analysis of model training performance, a large amount of configurations has to be examined. This requires a large system and is made more challenging by the time required for training a single workload. Even when using the aggressive limiting measure of capping training to 5 epochs (training iterations) will not guarantee that workloads take less than a day to train. Secondly, a combination of software and hardware metrics, such as training accuracy and power consumption, have to continuously be collected during this training process. This requires both integration with the training script and a variety of hardware profiling and monitoring tools. Thirdly, the data, in the form of time series, quickly grows as training goes on. This results in gigabytes of numeric data which then needs to be sifted through using a flexible yet efficient process. Fourthly, multiple different data sources must be compared with each other, yielding a range of different data visualization use cases. Fifthly, most of the data timeline may be inconsequential and repetitive, and interesting parts must be identified and investigated. Lastly, the solution needs to be as convenient to use as possible. Training a model via

DEEM '23, June 18, 2023, Seattle, WA, USA

ACM ISBN 979-8-4007-0204-4/23/06…$15.00
https://doi.org/10.1145/3595360.3595851

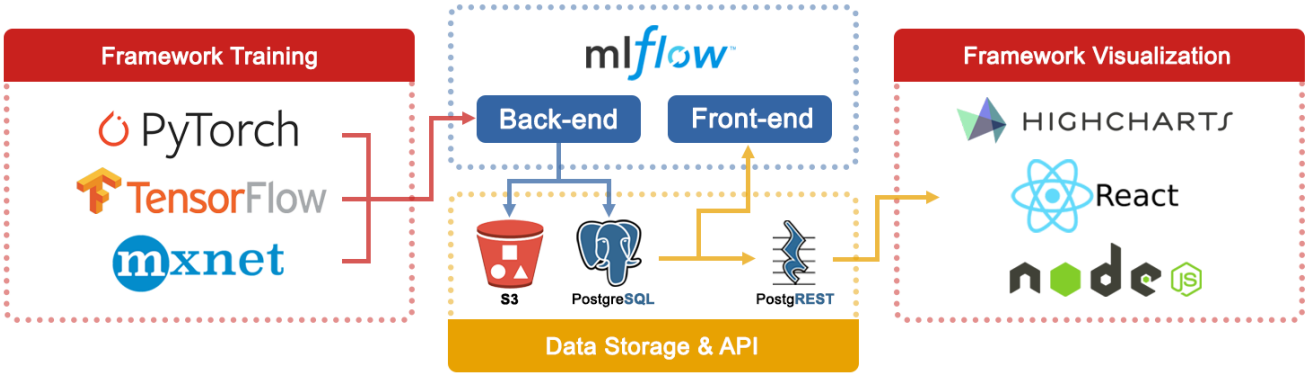

Figure 1: Dataflow architecture of our solution.

the framework should only impose minimal code intrusion and exploring results should be self-explanatory.

This paper presents our framework, which allows for benchmarking and visualizing deep learning model training in a reproducible manner. Our design takes the novel approach of repurposing the well-established machine learning lifecycle platform MLFlow [9] for machine learning systems analysis. Our extensions transform the platform to reach all of our aforementioned requirements in both back- and front-end while ensuring compatibility with pre-existing workflows and models. We describe how multiple combinations of models and datasets can be evaluated, in both isolated and collocated manner. Additionally, we show that there is support for different machine learning environments, and how existing models can be easily retrofitted to be supported by our framework. Researchers in our lab have extensively used our framework using a combination of datasets and machine learning models representative of a variety of deep learning workloads. Lastly, we delve into the visualization front-end and illustrate some of the results from our test runs. The framework is publicly available on GitHub[1].

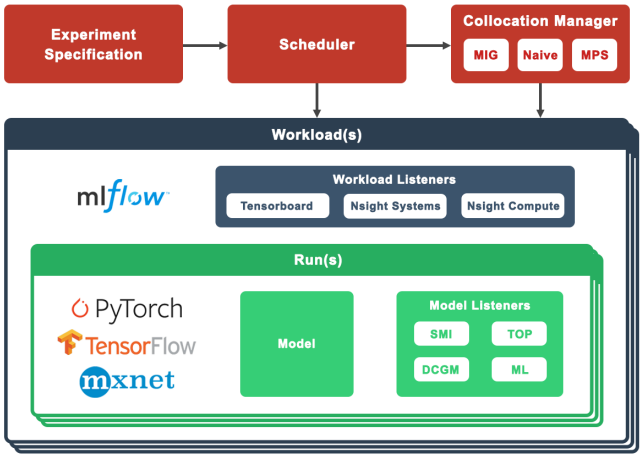

Figure 2: Data collection framework architecture. Experiments consist of workloads that can contain one or multiple model runs.

[1]https://github.com/Resource-Aware-Data-systems-RAD/dl-training-viz

2 FRAMEWORK

Our framework is split into a back-end and a front-end. Our data lifecycle extends that of MLFlow in order to meet our requirements. We illustrate the data architecture in Figure 1. Model training happens as an MLFlow client. This client sends data to the MLFlow server whenever there is a metric to report. The data flow between the client and server contains many events every second due to the large quantity of supported hardware metrics. This connection is frequent for all run-level listeners. Within MLFlow we distinguish the storage of relational data (experiment setup data, collected hardware monitoring metrics, etc.) and the storage of artifacts (stdout logs, profiling tool traces, etc.).

For our general data storage we need a solution that can serve data collection, MLFlow, and our front-end quickly and reliably; the first and last of which are particularly susceptible to performance bottlenecks. Data collection happens continuously throughout model training and requires storage to be available at all times. Data exploration requires sifting through a large amount of data quickly, presenting considerable throughput and responsiveness requirements for the front-end.

We found that hosting the relational data in a separate PostgreSQL database yields the best results. While MLFlow defaults to local data file storage, we have found this to scale poorly with respect to data size and be unreliable. Furthermore, file storage requires data access to happen through MLFlow, which inhibited the performance of our front-end. We store artifacts in S3 storage on a different server, again forgoing the native file storage. We host our React front-end on a separate server that connects to the database via PostgREST, which is an automatic REST API extension to PostgreSQL databases. We host this REST API on the same server as the database, though it may be more beneficial to run these on separate servers to improve scalability. Similarly, larger setups may benefit from two copies of the database, where one is to write metrics to and the other is for the front-end to read from. This would remove any interference during the training process where metrics are repeatedly written to the database.

3 BACK-END

Figure 2 illustrates the hierarchy presented to the user. Following the structure of MLFlow, individually trained models are called *runs* and are organized in experiments. We introduce a new layer

```
Experiment,Workload,Status,Run,Devices,Collocation,Environment,Model,Data,Listeners,Params
20,1,,,0,-,pytorch,timm.resnet26,/raid/datasets/cifar10/,smi+top+dcgmi,batch-size=128
20,2,,,2,3g.20gb,pytorch,timm.resnet26,/raid/datasets/cifar10/,smi+top+dcgmi,batch-size=128
20,2,,,2,3g.20gb,pytorch,timm.resnet26,/raid/datasets/cifar10/,smi+top+dcgmi,batch-size=128
20,3,,,0,MPS,pytorch,timm.resnet26,/raid/datasets/cifar10/,smi+top+dcgmi,batch-size=128
```

Figure 3: Experiment .csv file. Every row corresponds to a model training.

in-between runs and experiments called *workloads*. This is required as we include testing of multiple models at the same time as a requirement to test impact of workload collocation. A *workload* consists of one or more model runs, and a collection of workloads form an *experiment*. We will now go over the concepts introduced in the back-end pipeline, including the scheduler, environments, collocation, and listeners.

3.1 Scheduling

The execution of training experiments is managed by the workload scheduler. Models can be trained individually by use of a command line interface or be structured into experiment files. These are CSV files that can be edited in any text editor as shown in Figure 3. Every row is a single to-be-trained model. Models can be trained in a collocated fashion by sharing a workload ID. The numbering system of workloads is up to the user. In our case we opt to use three digits for workload identification. All models in a workload are trained concurrently. When model training terminates, either due to success or failure, the row is tagged as such. Once all the model training jobs terminate, the next workload is started. This repeats until all rows have been executed. Rows list all parameters required for training the model with the framework. In particular, the *params* field specifies any pass-through parameters that are sent directly to the model training script.

3.2 Environments

MLFlow as a platform allows machine learning researchers to train and store their models in a controlled fashion. We leverage their environments feature to ensure that our training is reproducible. Models can be trained in either anaconda environments or docker containers. We collectively call these the environments supported by the framework. Whenever a model is added, it is included in one of these environments. Any deep learning library, such as Tensorflow [15], Tensorflow-Keras [16], or Pytorch [17], is supported as long as an environment for it is included. This ensures compatibility and reduces the number of abstractions required to adopt an existing codebase to the framework. Code intrusion is kept to a minimum for the actual model training code. Achieving the basic hardware tracking functionalities only requires two lines of code:

```
from mldgpu import MultiLevelDNNGPUBenchmark
...
with MultiLevelDNNGPUBenchmark() as run:
  ...
```

Support for training the model outside of the framework can be kept by encapsulating these lines in conditional checks.

3.3 Collocation

Collocation allows for multiple models to be trained simultaneously, increasing hardware utilization. Multiple models can be trained on multiple GPUs, but can also share the same GPU. We support multiple technologies for sharing the same GPU resource:

- *MIG*, Multi-Instance GPU, is a hardware mechanism for recent Nvidia workstation graphics cards that allows for hardware partitioning of the GPU [18]. This allows for multiple models to train without interference.
- *MPS*, Multi-Processing Service, is a software mechanism by Nvidia for managing collocated processes on GPUs [19].
- *Naive* refers to simply launching multiple processes to use the same GPU without any other measures taken.

3.4 Listeners

In addition to the model performance metrics collected by MLFlow we require a host of hardware metrics to evaluate training performance. We introduce a group of additional processes called listeners that automatically record this information. By default we launch a full set of listeners to capture both host system and GPU hardware metrics. This preset can be overwritten by setting the listeners parameter of the run. Additionally, we provide an interface for intuitively adding new listeners. The following run-level listeners are included by default:

- *TOP* is a tool for recording CPU hardware metrics [20].
- *Nvidia-SMI* is a tool by Nvidia for recording GPU metrics. Nvidia-SMI yields GPU-wide metrics such as GPU utilization, memory usage, and power consumption [21].
- *DCGMI*, or Data Center GPU Management Interface, is a Nvidia tool for recording more advanced GPU metrics. Additionally, DCGMI supports metric collection under MIG for individual MIG-partitions [22].

In addition to these run-level listeners we support workload-level listeners. These collect more detailed information but are significantly more likely to impact the performance of the model training. Nvidia Nsight Systems [23] and Compute [24] in particular can have a pronounced effect on training performance [25]. Workload-level listeners are therefore disabled by default, but can be enabled easily inside the framework.

4 FRONT-END

While MLFlow comes with its own set of data exploration tools, they do not fulfil many of our research requirements. Firstly, the tools have not been built around comparing runs to each other and often provide extremely limited functionality. Secondly, they are not designed to handle large amounts of data points which can frequently take over 10 seconds to render a single time series. Lastly, the concept of a *workload* is central in our data architecture but does not exist in the MLFlow workflow.

To address these issues and serve our preferred workflow, we introduce a novel front-end application layer which consists of two primary components: an interface for data selection and an interface for data visualization. Before describing these components in detail, we will first establish the front-end's primary use cases which were identified from dissecting our experimental results:

- *Intra-workload*, where the models trained within a workload are to be compared to each other.

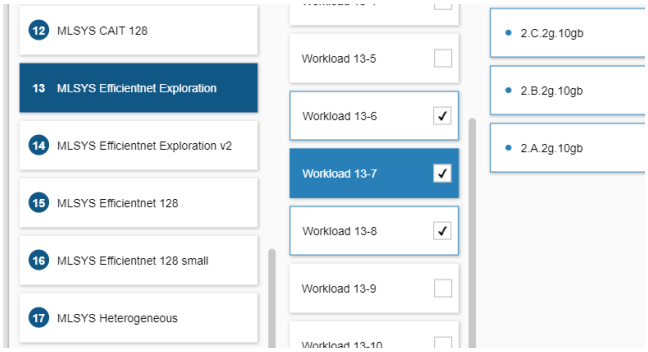

Figure 4: Model runs are organized into workloads and experiments which the user can then explore and visualize.

- *Inter-workload*, where workloads have to be compared to each other by taking the aggregate of their contained runs.
- *Mixed*, where specific runs of different workloads are to be compared to each other.

Figure 4 depicts the front-end's data selection interface where data sources can be browsed in a hierarchical manner. The user first specifies what experiment to explore, after which the workload and corresponding runs can be navigated and optionally selected via checkboxes. As a shortcut, all runs in a workload can be selected by clicking the workload's checkbox. Data from multiple workloads and multiple experiments can be selected at the same time and the system will automatically decide how to deal with the selected data. At any time, all currently selected runs are viewable by the user on the rightmost side of the interface. This section also serves as a shortcut to remove selected runs instead of having to locate their checkboxes again.

Figure 5: Using the Highcharts.js library, generated charts are interactive and responsive, allowing for quick dissection of the data.

After confirming their data selection, the user is directed to the second primary component where the data can be visualized. This interface initially appears blank and simply provides a dropdown list of the available metrics which can be visualized with the currently selected runs. Once the user has chosen a metric, the interface will reload with a corresponding graph for that metric (see Figure 5). There is no limit to the amount of graphs which can be generated and the selected runs can be changed between graphs. This allows for the comparison of different datasets within the same interface. Visualizations are also fully interactive and support toggling, zooming, and clipping, as well as exporting to PDF, PNG, or SVG formats. Finally, an interactive version of any visualization can be shared via a small file which will re-fetch the data from the server.

5 CONCLUSIONS

We have presented our framework for benchmarking and evaluating machine learning training. We have identified the challenges connected to collecting and processing real-time training data in an efficient and accessible manner. Additionally, we have tackled the visualization of this data, allowing for efficient and effective data exploration. We are able to compare data in experiments and between experiments with a unified interface. In addition to our own experimental analysis work with collocated workloads [26], our framework has been actively used in our lab to do experiments for medical imaging research[2]. We invite other researchers interested in both hardware and software metrics to consider it for their own research pipelines.

ACKNOWLEDGEMENTS

This work is funded by the Independent Research Fund Denmark's (Danmarks Frie Forskningsfond; DFF) Sapere Aude and Inge Lehmann programs under grant agreement number 0171-00061B and 0171-00062B, respectively. We also thank DASYA lab members at IT University of Copenhagen for their support, and the reviewers of DEEM for their constructive feedback.

REFERENCES

[1] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *NIPS*, 1097–1105.
[2] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Identity mappings in Deep Residual Networks. In *ECCV*, 630–645.
[3] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention Is All You Need. *NIPS*, 30, 6000–6010.
[4] Sebastian Baunsgaard, Sebastian Benjamin Wrede, and Pınar Tözün. 2020. Training for Speech Recognition on Coprocessors. In *ADMS*, 1–10.
[5] Alexandros Koliousis, Pijika Watcharapichat, Matthias Weidlich, Luo Mai, Paolo Costa, and Peter Pietzuch. 2019. Crossbow: Scaling Deep Learning with Small Batch Sizes on Multi-GPU Servers. *PVLDB*, 12, 11, 1399–1412.
[6] Shang Wang, Peiming Yang, Yuxuan Zheng, Xin Li, and Gennady Pekhimenko. 2021. Horizontally Fused Training Array: An Effective Hardware Utilization Squeezer for Training Novel Deep Learning Models. *MLSys*, 3, 599–623.
[7] Jun Yuan, Changjian Chen, Weikai Yang, Mengchen Liu, Jiazhi Xia, and Shixia Liu. 2021. A survey of visual analytics techniques for machine learning. *Computational Visual Media*, 7, 3–36.
[8] Lukas Biewald. 2020. Experiment Tracking with Weights and Biases. Software available from wandb.com. (2020). https://www.wandb.com/.
[9] Matei Zaharia et al. 2018. Accelerating the Machine Learning lifecycle with MLflow. *IEEE Data Eng. Bull.*, 41, 4, 39–45.

[2] https://purrlab.github.io/

[10] Mourad Mourafiq. 2018. Polyaxon: cloud native machine learning platform. Web page. Software available from polyaxon.com. (2018). https://github.com/polyaxon/polyaxon.
[11] [n. d.] Kubeflow. Web page. (). https://www.kubeflow.org/.
[12] 2023. Umlaut. (2023). https://github.com/hpides/End-to-end-ML-System-Benchmark/.
[13] Zeyuan Shang et al. 2019. Democratizing data science through interactive curation of ML pipelines. In *SIGMOD*, 1171–1188.
[14] Jorge Piazentin Ono et al. 2020. Pipelineprofiler: A visual analytics tool for the exploration of automl pipelines. *IEEE Transactions on Visualization and Computer Graphics*, 27, 2, 390–400.
[15] Martín Abadi et al. 2015. TensorFlow: large-scale machine learning on heterogeneous systems. (2015). https://www.tensorflow.org/.
[16] Keras Team. 2020. Keras: the python deep learning API. *Keras. io*.
[17] Adam Paszke et al. 2019. Pytorch: an imperative style, high-performance deep learning library. *NIPS*, 32, 8026–8037.
[18] 2020. NVIDIA MIG User Guide. http://docs.nvidia.com/datacenter/tesla/mig-user-guide/index.html.
[19] 2013. NVIDIA MPS. https://docs.nvidia.com/deploy/mps/index.html.
[20] 1984. top(1). https://man7.org/linux/man-pages/man1/top.1.html.
[21] 2012. NVIDIA System Management Interface (SMI). (June 28, 2012). https://developer.nvidia.com/nvidia-system-management-interface.
[22] 2015. NVIDIA DCGM. (Nov. 10, 2015). https://developer.nvidia.com/dcgm.
[23] 2018. NVIDIA Nsight Systems. (Mar. 12, 2018). https://developer.nvidia.com/nsight-systems.
[24] 2019. NVIDIA Nsight Compute. (Aug. 28, 2019). https://developer.nvidia.com/nsight-compute.
[25] Ehsan Yousefzadeh-Asl-Miandoab, Ties Robroek, and Pinar Tozun. 2023. Profiling and Monitoring Deep Learning Training Tasks. In *EuroMLSys*, 18–25.
[26] Ties Robroek, Ehsan Yousefzadeh-Asl-Miandoab, and Pınar Tözün. 2023. An Analysis of Collocation on GPUs for Deep Learning Training. (2023). arXiv: 2209.06018 [cs.LG].