DLSpec: A Deep Learning Task Exchange Specification

Abdul Dakkak[1*], Cheng Li[1*], Jinjun Xiong[2], Wen-mei Hwu[1]
[1]*University of Illinois Urbana-Champaign,* [2]*IBM T. J. Watson Research Center*
{dakkak, cli99, w-hwu}@illinois.edu, jinjun@us.ibm.com

Abstract

Deep Learning (DL) innovations are being introduced at a rapid pace. However, the current lack of standard specification of DL tasks makes sharing, running, reproducing, and comparing these innovations difficult. To address this problem, we propose DLSpec, a model-, dataset-, software-, and hardware-agnostic DL specification that captures the different aspects of DL tasks. DLSpec has been tested by specifying and running hundreds of DL tasks.

1 Introduction

The past few years have seen a fast growth in Deep Learning (DL) innovations such as datasets, models, frameworks, software, and hardware. Current practice of publishing these DL innovations involves developing ad-hoc scripts and writing textual documentation to describe the execution process of *DL tasks* (e.g. model training or inference). This requires a lot of effort and makes sharing and running DL tasks difficult. Moreover, it is often hard to reproduce reported accuracy or performance results and have a consistent comparison across DL tasks. This is a known [7, 8] "pain point" within the DL community. Having an exchange specification to describe DL tasks would be a first step to remedy this and ease the adoption of DL innovations.

Previous work included curation of DL tasks in framework model zoos [3, 6, 11–13, 17], developing model catalogs that can be used through a cloud provider's API [1, 2, 5], or introducing MLOps specifications [4, 18, 19]. However, these work either use ad-hoc techniques for different DL tasks or are tied to a specific hardware or software stack.

We propose DLSpec, a DL artifact exchange specification with clearly defined model, data, software, and hardware aspects. DLSpec's design is based on a few key principles (Section 2). DLSpec is model-, dataset-, software-, and hardware-agnostic and aims to work with runtimes built using existing MLOp tools. We further develop a DLSpec runtime to support DLSpec's use for DL inference tasks in the context of benchmarking [9].

*The two authors contributed equally to this paper.

2 Design Principles

While the bottom line of a specification design is to ensure the usability and reproducibility of DL tasks, the following principles are considered to increase DLSpec's applicability:

Minimal — To increase the transparency and ease the creation, the specification should contain only the essential information to use a task and reproduce its reported outcome.

Program-/human-readable — To make it possible to develop a runtime that executes DLSpec, the specification should be readable by a program. To allow a user to understand what the task does and repurpose it (e.g. use a different HW/SW stack), the specification should be easy to introspect.

Maximum expressiveness — While DL training and inference tasks can share many common software and hardware setups, there are differences when specifying their resources, parameters, inputs/outputs, metrics, etc. The specification must be general to be used for both training and inference tasks.

Decoupling DL task description — A DL task is described from the aspects of model, data, software, and hardware through their respective manifest files. Such a decoupling increases the reuse of manifests and enables the portability of DL tasks across datasets, software, and hardware stacks. This further enables one to easily compare different DL offerings by varying one of the four aspects.

Splitting the DL task pipeline stages — We demarcate the stages of a DL task into pre-processing, run, and post-processing stages. This enables consistent comparison and simplifies accuracy and performance debugging. For example, to debug accuracy, one can modify the pre- or post-processing step and observe the accuracy; and to debug performance, one can surround the run stage with the measurement code. This demarcation is consistent with existing best practices [10, 16].

Avoiding serializing intermediate data into files — A naive way to transfer data between stages of a DL task is to use files. In this way, each stage reads a file containing input data and writes to a file with its output data. This approach can be impractical since it introduces high serializing/deserializing overhead. Moreover, this technique would not be able to support DL tasks that use streaming data.

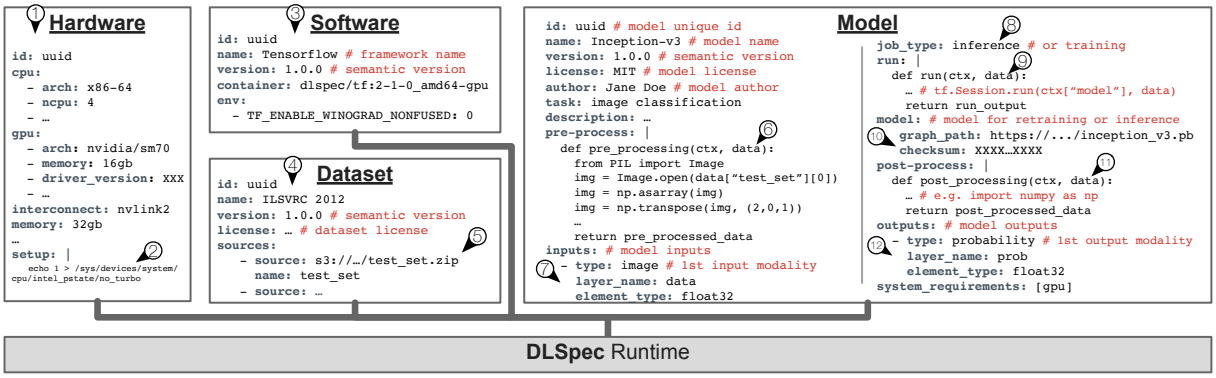

Figure 1: An example DLSpec that consists of a hardware, software, dataset and model manifest.

3 DLSpec Design

A DLSpec consists of four manifest files and a reference log file. All manifests are versioned [14] and have an ID (i.e. can be referenced). The four manifests (shown in Figure 1) are:

- **Hardware**: defines the hardware requirements for a DL task. The parameters form a series of hardware constraints that must be satisfied to run and reproduce a task.
- **Software**: defines the software environment for a DL task. All executions occur within the specified container.
- **Dataset**: defines the training, validation, or test dataset.
- **Model**: defines the logic (or code) to run a DL task and the required artifact sources.

The reference log is provided by the specification author for others to refer to. The reference log contains the IDs of the manifests used to create it, achieved accuracy/performance of DL task, expected outputs, and author-specified information (e.g. hyper-parameters used in training).

We highlight key details of DLSpec:

Containers — A software manifest specifies the container to use for a DL task, as well as any configuration parameters that must be set (e.g. framework environment variables). The framework or other libraries information are listed for ease of inspection and management.

Hardware configuration — While containers provide a standard way of specifying the software stack, a user cannot specify some hardware settings within a container. E.g., it is not possible to turn off Intel's turbo-boosting (Figure 1②) within a container. Thus, DLSpec specifies hardware configurations in the hardware manifest to allow the runtime to set them outside the container environment.

Pre-processing, run, and post-processing stages — The pre-/post-processing and run stages are defined via Python functions embedded within the manifest. We do this because a DLSpec runtime can use the Python sub-interpreter [15] to execute the Python code within the process thus avoiding using intermediate files (see Section 2). Using Python functions also allows for great flexibility; e.g. the Python function can download and run Bash and R scripts or download, compile, and some C++ code. The signature of the DLSpec Python functions is `fun(ctx, data)` where `ctx` is a hash table that includes manifest information (such as the types of inputs) accepted by the model. The second argument, `data`, is the output of the previous step in the dataset→pre-processing→run→post-processing pipeline. In Figure 1, for example, the pre-processing stage's ⑥ `data` is the list of file paths of the input dataset (ImageNet test set in this case).

Artifact resources — DL artifacts used in a DL task are specified as remote resources within DLSpec. The remote resource can be hosted on an FTP, HTTP, or file server (e.g. AWS S3, Zenodo) and have a checksum which is used to verify the download.

4 DLSpec Runtime

While a DL practitioner can run a DL task by manually following the setup described in the manifests, here we describe how a runtime (i.e. an MLOps tool) can use the DLSpec manifests shown in Figure 1.

A DLSpec runtime consumes the four manifests and selects the ① hardware to use and runs any ② setup code specified (outside the container). A ③ container is launched using the image specified, and the ④ dataset is downloaded into the container using the ⑤ URLs provided. The ⑥ dataset file paths are passed to the pre-processing function and its outputs are then processed to match the ⑦ model's input parameters. The ⑨ DL task is run. In the case of ⑧ inference, this causes the ⑩ model to be downloaded into the container. The result from the run are then ⑪ post-processed using the ⑫ data specified by the model outputs.

We tested DLSpect in the context of inference benchmarking and implemented a runtime for it [9]. We collected over 300 popular models and created reusable manifests for each. We created software manifests for major frameworks (Caffe, Caffe2, CNTK, MXNet, PyTorch, TensorFlow, TensorFlow Lite, and TensorRT), dataset manifests (for ImageNet, COCO, Pascal, CIFAR, etc.), and then wrote hardware specs for X86, ARM, and PowerPC. We tested our design and showed that it enables consistent and reproducible evaluation of DL tasks at scale.

5 Conclusion

An exchange specification, such as DLSpec, enables a streamlined way to share, reproduce, and compare DL tasks. DLSpec takes the first step in defining a DL task for both training and inference and captures the different aspects of DL model reproducibility. We are actively working on refining the specifications as new DL tasks are introduced. We maintain an updated published version of DLSpec at `dlspec.netlify.com`.

References

[1] Amazon SageMaker. https://aws.amazon.com/sagemaker, 2020. Accessed: 2020-02-20.

[2] MLOps: Model management, deployment and monitoring with Azure Machine Learning. https://docs.microsoft.com/en-us/azure/machine-learning/concept-model-management-and-deployment, 2020. Accessed: 2020-02-20.

[3] Caffe2 model zoo. https://caffe2.ai/docs/zoo.html, 2020. Accessed: 2020-02-20.

[4] Grigori Fursin, Herve Guillou, and Nicolas Essayan. CodeReef: an open platform for portable MLOps, reusable automation actions and reproducible benchmarking. *arXiv preprint arXiv:2001.07935*, 2020.

[5] Architecture for MLOps using TFX, Kubeflow Pipelines, and Cloud Build. https://bit.ly/39P6JFk, 2020. Accessed: 2020-02-20.

[6] GluonCV. https://gluon-cv.mxnet.io/, 2020. Accessed: 2020-02-20.

[7] Odd Erik Gundersen and Sigbjørn Kjensmo. State of the art: Reproducibility in artificial intelligence. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[8] Matthew Hutson. Artificial intelligence faces reproducibility crisis, 2018.

[9] Cheng Li, Abdul Dakkak, Jinjun Xiong, and Wen-mei Hwu. The design and implementation of a scalable dl benchmarking platform. *arXiv preprint arXiv:1911.08031*, 2019.

[10] Peter Mattson, Christine Cheng, Cody Coleman, Greg Diamos, Paulius Micikevicius, David Patterson, Hanlin Tang, Gu-Yeon Wei, Peter Bailis, Victor Bittorf, et al. Mlperf training benchmark. *arXiv preprint arXiv:1910.01500*, 2019.

[11] Modelhub. http://modelhub.ai, 2020. Accessed: 2020-02-20.

[12] ModelZoo. https://modelzoo.co, 2020. Accessed: 2020-02-20.

[13] ONNX Model Zoo. https://github.com/onnx/models, 2020. Accessed: 2020-02-20.

[14] Tom Preston-Werner. Semantic versioning 2.0.0. https://www.semver.org, 2019.

[15] Initialization, Finalization, and Threads. https://docs.python.org/3.6/c-api/init.html#sub-interpreter-support, 2019. Accessed: 2020-02-20.

[16] Vijay Janapa Reddi, Christine Cheng, David Kanter, Peter Mattson, Guenther Schmuelling, Carole-Jean Wu, Brian Anderson, Maximilien Breughe, Mark Charlebois, William Chou, et al. Mlperf inference benchmark. *arXiv preprint arXiv:1911.02549*, 2019.

[17] TensorFlow Hub. https://www.tensorflow.org/hub, 2020. Accessed: 2020-02-20.

[18] Manasi Vartak, Harihar Subramanyam, Wei-En Lee, Srinidhi Viswanathan, Saadiyah Husnoo, Samuel Madden, and Matei Zaharia. ModelDB: a system for machine learning model management. In *Proceedings of the Workshop on Human-In-the-Loop Data Analytics*, pages 1–3, 2016.

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