DataStates: Towards Lightweight Data Models for Deep Learning

Bogdan Nicolae

DataStates: Towards Lightweight Data Models for Deep Learning

Bogdan Nicolae

Argonne National Laboratory, USA
bogdan.nicolae@acm.org

Abstract. A key emerging pattern in deep learning applications is the need to capture intermediate DNN model snapshots and preserve or clone them to explore a large number of alternative training and/or inference paths. However, with increasing model complexity and new training approaches that mix data, model, pipeline and layer-wise parallelism, this pattern is challenging to address in a scalable and efficient manner. To this end, this position paper advocates for rethinking how to represent and manipulate DNN learning models. It relies on a broader notion of data states, a collection of annotated, potentially distributed data sets (tensors in the case of DNN models) that AI applications can capture at key moments during the runtime and revisit/reuse later. Instead explicitly interacting with the storage layer (e.g., write to a file), users can "tag" DNN models at key moments during runtime with metadata that expresses attributes and persistency/movement semantics. A high-performance runtime is the responsible to interpret the metadata and perform the necessary actions in the background, while offering a rich interface to find data states of interest. Using this approach has benefits at several levels: new capabilities, performance portability, high performance and scalability.

Keywords: deep learning, state preservation, clone, model reuse

1 Introduction

Deep learning applications are rapidly gaining traction both in industry and scientific computing. A key driver for this trend has been the unprecedented accumulation of big data, which exposes plentiful learning opportunities thanks to its massive size and variety. Unsurprisingly, there has been significant interest to adopt deep learning at very large scale on supercomputing infrastructures in a wide range of scientific areas: fusion energy science, computational fluid dynamics, lattice quantum chromodynamics, virtual drug response prediction, cancer research, etc.

Initially, scientific applications have gradually adopted deep learning more or less in an ad-hoc fashion: searching for the best deep neural network (DNN) model configuration and hyperparameters through trial-and-error, studying the tolerance to outliers by training with and without certain datasets, etc. Often,

the lack of *explainability*, i.e., being able to understand why a DNN model learned certain patterns and what correlations can be made between these patterns and the training datasets was overlooked if the results were satisfactory. However, with increasing complexity of the DNN models and the explosion of the training datasets, such a trend is not sustainable. Scientific applications are particularly affected by this because they are often mission-critical (e.g., a patient misdiagnosis can have severe consequences), unlike many industrial applications (e.g., a misclassification of a picture as a dog instead of a cat is mostly harmless).

In a quest to solve this challenge, systematic approaches are beginning to emerge: guided *model discovery* where the DNN architecture [26] and hyperparameters [3] are automatically identified, *sensitivity analysis* [29], which is used to identify what parts/layers of the DNN model and/or training samples are the most influential the learning process and how robust the DNN model is regarding tolerance to outliers or transfer learning (i.e., ability to reuse the learned patterns to solve related problems), etc.

All these approaches rely on several fundamental data management abilities: (1) *capture* intermediate snapshots of the DNN model in order to study its evolution in time and potentially *reuse* it later; (2) *clone* a DNN model whose training has progressed up to a point into many parallel alternatives where slight variations are introduced; (3) apply the *FAIR* principles [2] (findable, accessible, interoperable, reusable) to the snapshots, to make it easy to navigate through their evolution and/or search for interesting snapshots that can be reused.

However, with increasing complexity and sizes of DNN models and training data, a mix of data parallel, model parallel, pipeline parallel and layer-wise parallel approaches are emerging to speed-up the training process. In this context, a training instance is not a single process anymore, but an entire group of tightly coupled processes that are distributed across many devices and/or compute nodes of large scale HPC infrastructures. Such groups of processes collaboratively work on a shared, distributed DNN model state, exhibiting specific properties and access patterns. In addition, HPC data centers are increasingly equipped with complex heterogeneous storage stacks (multi-level memory hierarchies on compute nodes, distributed caches and burst buffers, key-value stores, parallel file systems, etc.). Under such circumstances, the fundamental data management abilities mentioned above become highly challenging to implement in a scalable and efficient manner.

In this position paper we advocate for *DataStates*, a new data model that addresses the aforementioned challenges by rethinking how to represent and manipulate scientific datasets. At its core is the notion of a *data state*, which is a collection of annotated, potentially distributed data structures that applications can capture at key moments during the runtime and revisit/reuse later. Instead explicitly interacting with the storage layer (e.g., to save the dataset into a file), users define such coupled datasets and "tag" them at key moments during runtime with metadata that expresses attributes and persistency/movement semantics. Tagging triggers asynchronous, high performance I/O strategies that run in the background and capture a consistent snapshot of the datasets and

associated metadata into the lineage, a history that records the evolution of the snapshots. Using dedicated primitives, users can easily navigate the lineage to identify and revisit snapshots of interest (based on attributes and/or content) and roll back or evolve the lineage in a different parallel direction.

Using this approach, clone and revisit of DNN model states become lightweight primitives focused on high performance, scalability and FAIR capabilities, which not only accelerates existing approaches for model exploration, sensitivity analysis and explainability, but also encourages new algorithms and techniques that can take advantage of frequent reuse of intermediate DNN models. We summarize our contributions as follows:

- We discuss a series of challenges and opportunities that arise in the context of deep learning, where a mix of data parallel, model parallel, pipeline parallel and layer-wise parallel approaches are increasingly applied to improve the performance and scalability of the training (Section 2).
- We introduce an overview of *DataStates*, the data model and runtime we advocate in this paper. We insist both on how the notion of data states can be used as a fundamental abstraction to capture, search for and reuse intermediate datasets, as well as the advantages of such an abstraction (Section 3).
- We position *DataStates* in the context of state-of-art, insisting both on the gaps filled by our approach and the complementarity that can be achieved by using DataStates in conjunction in other approaches (Section 4).

2 Background

Deep learning approaches have evolved from independent training and inference into complex workflows (Figure 1): they involve training sample pre-processing and augmentation (e.g., create more training samples by stretching or rotating images), model discovery (both DNN architecture and hyperparameters), training and validation of the inference, sensitivity analysis used to explain the model and/or influence the data pre-processing and model discovery.

In this context, there is a need to explore a large number of alternatives, which applies for each step of the workflow. For example, model discovery strategies based on evolutionary techniques [26] (such as genetic algorithms) need to maintain a large population of promising DNN model individuals, which are combined and/or mutated in the hope of obtaining better individuals. Training a DNN model may also involve alternatives, especially in the case of reinforcement learning [35], where there are multiple variations of environments and alternative actions possible. DNN models with early exits [30] are becoming increasingly popular: in this case, the inference can take alternative shorter (and thus faster) paths through the model layers when they provide sufficient accuracy (e.g., non-ambiguous regions in a classification problem). Sensitivity analysis [29] needs to explore many alternative training paths that include/exclude certain training samples and/or layers in order to understand their impact. For example, CANDLE [32] (Cancer Deep Learning Environment) employs an approach where the

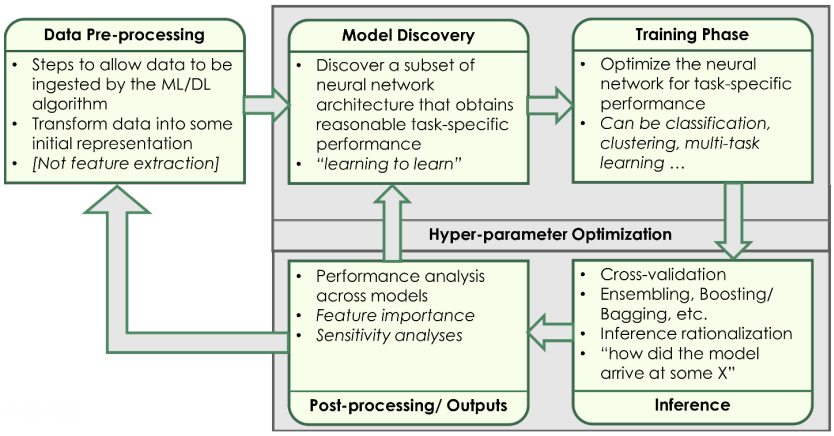


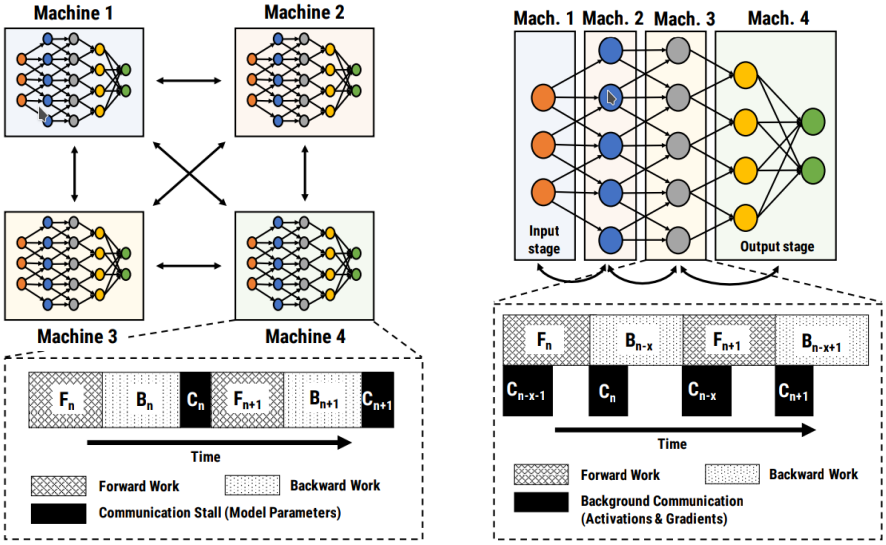
Fig. 1. Structure of a modern deep learning workflow

input data is split into regions and the training process is forked into alternative directions, each of which excludes one of the regions. This process continues recursively for each excluded region until a desired granularity for the excluded training samples is reached, enabling the study of their impact in the training process.

Such alternatives introduce the need for advanced data management approaches: capture intermediate DNN model/layer snapshots as the training (or inference) progresses and then either preserve them for later study/revisiting, or clone them for the purpose of forking the training (or inference) into different parallel directions. To make these snapshots usable, several capabilities related to the FAIR principles (findable, accessible, interoperable, reusable) are needed: automatically capture the evolution of the snapshots, expose their properties, enable search based on such properties, reshape the snapshots on-the-fly to adapt to a new context where it needs to be used.

However, providing such advanced data management capabilities is challenging, because DNN training approaches are constantly being adapted to take advantage of large-scale infrastructures. In this context, the most widely used technique is *synchronous data-parallel* training. It creates replicas of the DNN model on multiple workers, each of which is placed on a different device and/or compute node. We denote such workers as *ranks*, which is the terminology typically used in high performance computing (HPC). The idea is to train each replica in parallel with a different mini-batch, which can be done in an embarrassingly parallel fashion during the forward pass on all ranks. Then, during back-propagation, the weights are not updated based on the local gradients, but using global average gradients computed across all ranks using all-reduce operations. This effectively results in all ranks learning the same pattern, to which each individual rank has contributed. The process is illustrated in Figure 2(a).

Model parallelism [11] is another complementary approach (Figure 3). It works by partitioning the DNN model across multiple ranks, each of which is



(a) Data parallelism: DNN model is replicated, local gradients are averaged.

(b) Pipeline parallelism: DNN model partitioned and distributed as stages (full layers).

Fig. 2. Data parallelism vs. pipeline parallelism (adapted from [19])

running on a different device and/or compute node. This solves the problem of large DNN models that do not fit in the memory of a rank, but requires data transfers between operations and disallows parallelism within an operation.

Pipeline parallelism [19] combines model parallelism with data parallelism. The idea is to partition the DNN model into stages, each of which is made of one or more layers (and can be replicated like in the case of data-parallelism). Each stage is assigned to a different rank, which effectively form a pipeline (Figure 2(b)). Unlike data and model parallelism, where only one mini-batch is active at a given moment for the whole duration of the training step, pipeline parallelism injects multiple mini-batches into the stages one after the other: during the forward pass, each stage sends the output activations to the next stage, while simultaneously starting to process another mini-batch. Similarly, after completing the backward-propagation for a mini-batch, each stage sends the gradients to the previous stage and begins to process another mini-batch.

DL algorithms take advantage of multi-core and hybrid architectures (e.g., CPUs + GPUs) to parallelize the gradient computation and weight updates. Specifically, once a rank has finished computing the local gradients for a layer, it immediately proceeds to compute the local gradients of the previous layer. At the same time, it waits for all other ranks to finish computing their local gradients for the same layer, then updates the weights (based on the average gradients obtained using all-reduce in the case of data-parallelism). This is called *layer-wise parallelism*. Another way of reasoning about this process is by means of a DAG

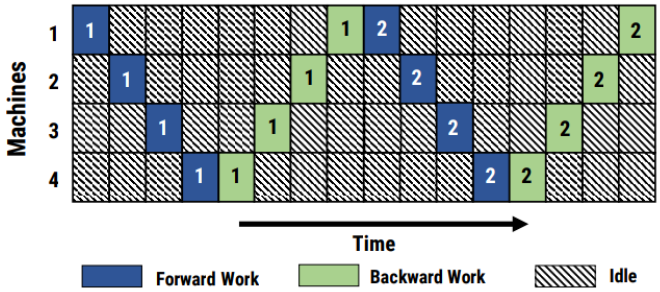


Fig. 3. Model parallelism: DNN model is partitioned and distributed

(directed acyclic graph), where each layer is a pipeline: compute local gradients, average gradients globally, update weights. The local gradient computation of each layer is a dependency for both the local gradient computation of the previous layer and the rest of the pipeline: once it is complete, both paths in the DAG can be executed in parallel.

As a consequence, the distributed nature of DNN model snapshots and the complex multi-level parallelism considerations make the problem of capturing and preserving/cloning intermediate DNN model snapshots non-trivial. This is further augmented by the need to adopt the FAIR principles and the increasingly complex heterogeneous storage stacks [14] that are deployed in modern HPC data centers. Nevertheless, there are also significant opportunities in this space: according to our previous study [13], the combination of data parallelism and layer-wise parallelism leads to subtle delays that can be exploited to to overlap the back-propagation with fine-grain asynchronous data management operations in the background, which can significantly reduce their overhead. We demonstrated the feasibility of this idea both for DNN model checkpointing [23] and DNN model cloning [25], obtaining an overhead reduction of an order of magnitude compared with other state-of-art alternatives.

3 DataStates: An Overview

In this section, we introduce the main ideas and principles behind *DataStates*, the data model we advocate in this paper.

In a nutshell, a data state is a collection of annotated, potentially distributed data structures that applications can capture at key moments during the runtime. For the purpose of this work, we assume such distributed data structures to represent the DNN model state. The application indicates such key moments explicitly (noting that automation of this process opens an interesting research question). More formally, a data state is tuple (C, M_s, M_a) that defines a content C and any associated metadata M_s and M_a. We differentiate between *summary metadata* (denoted M_s), used to label and/or summarize C, and *actionable metadata* (denoted M_a), used to express intents over how C is managed. These intents

take the form of hints (e.g., access pattern) and/or properties (e.g., durability, scope, relationship to other data states). Users do not care how the intent is materialized: it is the job of the DataStates runtime to formulate an appropriate plan through a series of actions, which in our case refer to persisting, caching and fetching a data state. This is a general principle: new intents and action plans can be added as needed.

Many distributed ranks (owners) can share the same data state and mutate it collaboratively by updating its content and/or metadata. We assume the owners are directly responsible for concurrency control and consistency, which in our case is transparently handled by the deep learning frameworks. When the owners reach a key moment during runtime (e.g., an epoch of the training has finished), they *tag* the data state. This triggers a transition into a new data state. From the owner's perspective, nothing changed: they can continue working on C as usual. Meanwhile, in the background, the runtime applies the action plan corresponding to M_a for the original data state as it was at the moment of tagging. The runtime guarantees that any side effects due to the action plan are tied to the original data state and do not affect the new data state (which may trigger internal temporary copies during tagging or copy-on-write). A data state that was tagged is *stable* if its action plan completed successfully and *unstable* otherwise. It is illegal to access unstable states, but the runtime offers support to query about their status and wait for them to become stable.

Both the data states and the transitions between them are recorded into the *lineage*, which keeps the evolution of the data states. The lineage exposes primitives to *navigate* (i.e., move to a successor or predecessor) and to *search* (i.e., find data states that satisfy given properties) the lineage. Applications can use such primitives to discover and visit interesting data states. For example, this can be used to follow the evolution of tagged DNN model states during training or to search for previously tagged intermediate DNN models based on their accuracy and/or other attributes. Furthermore, each data state can be part of one or more *scopes*, which are explicitly specified in M_a. To avoid the explosion of storage space utilization, non-critical data states that have gone out of scope (e.g., non-critical or locally relevant intermediate DNN models) and their transitions can be *pruned* from the lineage as needed. Pruning is subject to garbage collection algorithms, but can also be triggered explicitly through a dedicated primitive.

The lineage can be combined with two additional powerful primitives: *fork* and *reshape*. Both of them are similar to tagging (i.e., they trigger a transition to a new data state and the execution of an asynchronous action plan) but with important differences. Fork creates a clone of the data state on an entirely different set of processes and "splits" the lineage into two independent directions that can evolve separately. For example, fork can be used to explore an alternative direction for training a DNN model (e.g., using different hyper-parameters and/or training samples). Reshape enables the processes to change the layout and/or distribution of C, by specifying appropriate attributes in M_a. Specifically, this refers to operations such as *migrate* (to different processes) and *shuffle* (i.e., ex-

change pieces of C between processes, which is a common pattern in distributed training of DNN models). Combined with tagging and search/navigation, these two primitives allow flexible strategies to explore multiple parallel evolutions and revisit/reuse previous data states. Note the versatility of reshape, which can be extended with multiple other patterns. For example, data states could be used to record a lineage for Tensorflow [1] by introducing support for tensor operations: slice, rebalance, stack, etc.

This approach has several advantages. First, it introduces *native constructs* that addresses the FAIR (findable, accessible, inter-operable, reusable) principles [2]: (1) findability is directly enabled by the lineage through navigation and search capabilities; (2) accessibility is enabled in a declarative fashion by specifying the desired intent (thus freeing applications from having to worry where their data is and how to bring it where it is needed); (3) inter-operability hiding the implementation of the I/O strategies from the user (thus eliminating differences in the interpretation of actionable metadata); (4) reusability is naturally facilitated by a single, unified view of all data states and the relationship between them, which can be revisited as desired.

Second, the separation of the intents from the actual implementation of the constraints and desired effects they represent is an important step towards *performance portability*, i.e., avoiding the need to customize the application codes on each machine to account for differences in performance characteristics, custom vendor APIs, etc. Specifically, since data states capture the intent only, action plans can be customized for a dedicated supercomputing infrastructure, potentially taking advantage of differences in architecture, performance characteristics of heterogeneous storage and vendor-specific features in order to introduce specific optimizations.

Third, the design of DataStates is *lightweight and data-centric*. DataStates is focused on the evolution of data and metadata alone, leaving other components to worry about computational and synchronization aspects. The data states are wrapping in-memory user data structures directly and are close to their intended life-cycle, therefore minimizing overheads related to data movements (which is not the case when using external repositories). Furthermore, DataStates masks the data management overhead asynchronously in the background, therefore minimizing the interruption of the application. Combined with clever interleaving of such asynchronous operations at fine-granularity during the backpropagation, this approach becomes crucial in facilitating the goal of achieving high performance and scalability.

4 Related Work and Positioning

Checkpoint-restart is a well researched HPC pattern relevant in the context of clone and revisit. In this regard, *multi-level checkpointing*, as adopted by frameworks such as SCR [18] and FTI [4], is a popular approach that leverages complementary strategies adapted for HPC storage hierarchies. VELOC [24, 31] takes this approach further by introducing asynchronous techniques to apply

such complementary strategies in the background. When the checkpoints of different processes have similar content, techniques such as [20, 21] can be applied to complement multi-level checkpointing. However, redundancy is detected on-the-fly, which can be an unnecessary overhead for clone and revisit (e.g., model replicas are known to be identical for data-parallel training). Dedicated checkpointing techniques for deep learning are rudimentary: TensorFlow checkpoints model to files in its SavedModel format, [1] or in HDF5 files through Keras.[2] These file-based methods, while simple and adapted for single-node training, are becoming a bottleneck when scaling data-parallel training to a large number of compute nodes. Our own previous work [23, 25] introduced scalable approaches to address these limitations. Although not flexible enough in the general clone and revisit scenarios, they can be used as a building block for DataStates.

In a quest to achieve scalability and flexibility, HPC storage stacks have become increasingly heterogeneous [14]. In addition to parallel file systems, modern supercomputers feature a variety of additional storage subsystems (e.g., burst buffers [7] or key-value stores such as DAOS [16]) and deep memory hierarchies (HBM, DDRAM, NVM). Such storage subsystems focus on raw I/O performance acceleration by implementing low-level read/write or put/get abstractions. They complement well the rigid POSIX model used by parallel file systems (e.g., lack of efficient support for fine-grained I/O operations and concurrency control). However, this is not enough to implement the high-level capabilities necessary for clone and revisit. Furthermore, the large diversity of services leads to added complexity and limited sharing and reuse potential because of the lack of performance portability.

In the big data community, Spark [34, 28] has gained considerable traction as a generic analytics framework. Part of its success lies in the functional data processing model that hides the details of parallelism from the user, enabling ease of use and performance portability through high-level in-memory transformations. Notable in this context is the concept of RDDs [33] (Resilient Distributed Data Sets), which are Spark's abstraction for intermediate data. Despite efforts to leverage heterogeneous storage for RDDs (e.g., Apache Ignite [6]), they are tied to the rigid programming model of Spark, which emphasizes loosely coupled patterns and high-level languages that trade off performance for productivity. Therefore, such abstractions are unsuitable for the HPC ecosystem, which emphasizes high performance and scalability, tightly-coupled patterns and hybrid programming models.

Provenance tracking and reproducibility is another area closely related to DataStates. In the HPC ecosystem, EMPRESS [12] aims to provide an alternative to rudimentary attribute capabilities offered by HDF5 and NetCDF through extensible metadata. This broadens the scope beyond single files or application-specific formats, but does not feature a lineage. In the Spark ecosystem, RDDs feature a computation-centric lineage that records what data transformations were applied. This lineage is hidden from the application and used internally

[1] https://www.tensorflow.org/guide/saved_model
[2] https://www.tensorflow.org/guide/keras/save_and_serialize

to recompute RDDs (e.g., in case of failures or need to reuse). By contrast, DataStates has the opposite goal: a lineage that records actual data snapshots annotated with metadata (thus avoiding expensive recomputation), which is exposed to the application and used as a tool to revisit previous states. In itself, this is already a powerful introspection mechanism that aids provenance tracking and reproducibility. Of course, there is value in combining both approaches to create a complete picture. Unfortunately, capturing the computational context in the HPC ecosystem is nontrivial, as it involves a large number of libraries and runtimes. Containers are one possible solution and are used by approaches such as Data Pallets [15]. DataStates can complement well such efforts.

Versioning and revision control systems (e.g. SVN [10], GIT [8], Mercurial [5]) are widely used to keep track of changes to source code during software development. They feature native support for data-centric provenance: users can keep track of successive changes and revisit, roll-back, branch, merge, etc. They also feature an entire array of space-efficient techniques to store only incremental differences. However, these optimizations are designed for text data (mostly source code) and are not designed for high-performance and scalability (they assume each user has room to maintain a whole local copy of the repository). Systems were proposed before to address this issue: For example, BlobSeer [22] is a distributed data store for binary large objects that manages intermediate snapshots much like revision control systems. However, it was not designed to handle heterogeneous data and metadata (its abstraction is a blob, i.e., a large sequence of bytes): largely, it behaves like a key-value store with versioning support, therefore missing support to search and navigate the history.

Repositories for VM images [27] and containers (e.g. Docker [17]) are an industry standard to facilitate collaboration and sharing between multiple users for computational environments. In a similar spirit, recent efforts such as DLHub [9] aim to build model repositories for deep learning applications: users publish, discover and share full models, including dependencies (e.g., Python environment), into executable servables (that may include Docker images, Amazon S3 buckets, etc.) through REST APIs. DataStates also focuses on enabling search and reuse semantics, but from a different perspective: it introduces a general data model (useful beyond deep learning), lightweight (HPC-oriented) and data life-cycle oriented (mix ephemeral with persistent data, leverage local storage and in-situ capabilities, data-centric lineage). This is more appropriate for the DNN model clone and revisit scenarios we target, where an external repository can become a bottleneck. On the other hand, DataStates is well complemented by approaches like DLHub, as they can handle security and other aspects needed to enable multi-user sharing beyond a single supercomputer.

5 Conclusions

In this position paper we have introduced *DataStates*, a new data model that exposes high-level primitives to capture, fork, search and reuse of scientific datasets. Such high-level primitives are especially important for an efficient implementa-

tion of many deep learning scenarios that involve the need to capture intermediate DNN models and explore a large number of alternative training and/or inference paths.

Despite increasing complexity due to distributed DNN model state, as well as a mix of distributed training approaches (data, model, pipeline, layer-wise parallel), DataStates is well positioned to leverage the opportunity such circumstances present, especially with respect to overlapping the back-propagation phase of training with asynchronous fine-grain operation in the background in order to progress on data management aspects with minimal overhead on an ongoing training. Additionally, DataStates has three other advantages: it brings FAIR (findable, accessible, inter-operable, reusable) semantics to deep learning frameworks, it enables performance portability by separating data management intents (defined by the user) from actions necessary to satisfy them, it enables high performance and scalability by introducing lightweight, in-situ data manipulation semantics that are close to the data life-cycle of DNN models.

Encouraged by promising initial results, especially for the related problem of scalable checkpointing and cloning of DNN models for data-parallel training approaches, we plan to illustrate in future work the benefits of DataStates in the context of deep learning.

Acknowledgments

This material is based upon work supported by the U.S. Department of Energy (DOE), Office of Science, Office of Advanced Scientific Computing Research, under Contract DE-AC02-06CH11357.

References

1. Abadi, M., Agarwal, A., Barham, P., et al.: TensorFlow: Large-scale machine learning on heterogeneous systems (2015), http://tensorflow.org/, software available from tensorflow.org
2. et al, M.D.W.: The fair guiding principles for scientific data management and stewardship. Scientific Data 3(160018) (2016)
3. Balaprakash, P., Egele, R., Salim, M., Wild, S.M., Vishwanath, V., Xia, F., Brettin, T., Stevens, R.: Scalable reinforcement-learning-based neural architecture search for cancer deep learning research. In: SC'19: The 2019 International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 37:1–37:33 (2019)
4. Bautista-Gomez, L., Tsuboi, S., Komatitsch, D., Cappello, F., Maruyama, N., Matsuoka, S.: FTI: High performance fault tolerance interface for hybrid systems. In: SC '11: The 2011 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 32:1–32:32. Seattle, USA (2011)
5. Bernard, J.: Mercurial–revision control approximated. Linux J. 2011(212) (2011)
6. Bhuiyan, S., Zheludkov, M., Isachenko, T.: High Performance In-memory Computing with Apache Ignite. Lulu.com (2017)

7. Cao, L., Settlemyer, B.W., Bent, J.: To share or not to share: Comparing burst buffer architectures. In: HPC '17: The 25th High Performance Computing Symposium. pp. 4:1–4:10. Virginia Beach, Virginia (2017)
8. Chacon, S., Straub, B.: Pro Git. Apress, Berkely, CA, USA, 2nd edn. (2014)
9. Chard, R., Ward, L., Li, Z., Babuji, Y., Woodard, A., Tuecke, S., Chard, K., Blaiszik, B., Foster, I.: Publishing and serving machine learning models with dlhub. In: PEARC '19: Practice and Experience in Advanced Research Computing on Rise of the Machines (Learning). Chicago, USA (2019)
10. Collins-Sussman, B.: The subversion project: Buiding a better cvs. Linux J. 2002(94) (2002)
11. Dean, J., Corrado, G.S., Monga, R., Chen, K., Devin, M., Le, Q.V., Mao, M.Z., Ranzato, M., Senior, A., Tucker, P., Yang, K., Ng, A.Y.: Large scale distributed deep networks. In: NIPS'12: The 25th International Conference on Neural Information Processing Systems. p. 1223–1231. Lake Tahoe, USA (2012)
12. Lawson, M., Ulmer, C., Mukherjee, S., Templet, G., Lofstead, J.F., Levy, S., Widener, P.M., Kordenbrock, T.: Empress: extensible metadata provider for extreme-scale scientific simulations. In: PDSW-DISCS@SC'17: The 2nd Joint International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems. pp. 19–24 (2017)
13. Li, J., Nicolae, B., Wozniak, J., Bosilca, G.: Understanding scalability and fine-grain parallelism of synchronous data parallel training. In: MLHPC'19: 5th Workshop on Machine Learning in HPC Environments (in conjunction with SC19). pp. 1–8. Denver, USA (2019)
14. Lockwood, G., Hazen, D., Koziol, Q., Canon, R., Antypas, K., Balewski, J., e.a.: Storage 2020: A vision for the future of hpc storage. Tech. rep., Lawrence Berkeley National Laboratory (2017)
15. Lofstead, J., Baker, J., Younge, A.: Data pallets: Containerizing storage for reproducibility and traceability. In: ISC'19: 2019 International Conference on High Performance Computing. pp. 36–45 (2019)
16. Lofstead, J., Jimenez, I., Maltzahn, C., Koziol, Q., Bent, J., Barton, E.: Daos and friends: A proposal for an exascale storage system. In: SC '16: The 2016 International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 50:1–50:12. Salt Lake City, Utah (2016)
17. Merkel, D.: Docker: Lightweight linux containers for consistent development and deployment. Linux J. 2014(239) (2014)
18. Moody, A., Bronevetsky, G., Mohror, K., Supinski, B.R.d.: Design, modeling, and evaluation of a scalable multi-level checkpointing system. In: SC '10: The 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 1:1–1:11. New Orleans, USA (2010)
19. Narayanan, D., Harlap, A., Phanishayee, A., Seshadri, V., Devanur, N.R., Ganger, G.R., Gibbons, P.B., Zaharia, M.: Pipedream: Generalized pipeline parallelism for dnn training. In: SOSP '19: The 27th ACM Symposium on Operating Systems Principles. p. 1–15. Huntsville, Canada (2019)
20. Nicolae, B.: Towards Scalable Checkpoint Restart: A Collective Inline Memory Contents Deduplication Proposal. In: IPDPS '13: The 27th IEEE International Parallel and Distributed Processing Symposium. Boston, USA (2013), http://hal.inria.fr/hal-00781532/en
21. Nicolae, B.: Leveraging naturally distributed data redundancy to reduce collective I/O replication overhead. In: IPDPS '15: 29th IEEE International Parallel and Distributed Processing Symposium. pp. 1023–1032 (2015)

22. Nicolae, B., Antoniu, G., Bougé, L., Moise, D., Carpen-Amarie, A.: BlobSeer: Next-generation data management for large scale infrastructures. J. Parallel Distrib. Comput. 71, 169–184 (2011)
23. Nicolae, B., Li, J., Wozniak, J., Bosilca, G., Dorier, M., Cappello, F.: Deepfreeze: Towards scalable asynchronous checkpointing of deep learning models. In: CGrid'20: 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing. pp. 172–181. Melbourne, Australia (2020)
24. Nicolae, B., Moody, A., Gonsiorowski, E., Mohror, K., Cappello, F.: VeloC: Towards high performance adaptive asynchronous checkpointing at large scale. In: IPDPS'19: The 2019 IEEE International Parallel and Distributed Processing Symposium. pp. 911–920. Rio de Janeiro, Brazil (2019)
25. Nicolae, B., Wozniak, J.M., Dorier, M., Cappello, F.: DeepClone: Lightweight State Replication of Deep Learning Models for Data Parallel Training. In: CLUSTER'20: The 2020 IEEE International Conference on Cluster Computing. Kobe, Japan (2020)
26. Real, E., Moore, S., Selle, A., Saxena, S., Suematsu, Y.L., Tan, J., Le, Q.V., Kurakin, A.: Large-scale evolution of image classifiers. In: ICML'17: The 34th International Conference on Machine Learning. p. 2902–2911. Sydney, Australia (2017)
27. Saurabh, N., Kimovski, D., Ostermann, S., Prodan, R.: Vm image repository and distribution models for federated clouds: State of the art, possible directions and open issues. In: Euro-Par 2016: Parallel Processing Workshops. pp. 260–271. Grenoble, France (2016)
28. Shanahan, J.G., Dai, L.: Large scale distributed data science using apache spark. In: KDD '15: The 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. pp. 2323–2324. Sydney, Australia (2015)
29. Shu, H., Zhu, H.: Sensitivity analysis of deep neural networks. In: AAAI'19: The 33rd AAAI Conference of Artificial Intelligence. pp. 4943–4950 (2019)
30. Teerapittayanon, S., McDanel, B., Kung, H.T.: Branchynet: Fast inference via early exiting from deep neural networks. In: ICPR'16: The 23rd International Conference on Pattern Recognition. pp. 2464–2469. Cancun, Mexico (2016)
31. Tseng, S.M., Nicolae, B., Bosilca, G., Jeannot, E., Cappello, F.: Towards portable online prediction of network utilization using MPI-level monitoring. In: EuroPar'19 : 25th International European Conference on Parallel and Distributed Systems. pp. 1–14. Goettingen, Germany (2019)
32. Wozniak, J., Jain, R., Balaprakash, P., et al.: Candle/supervisor: A workflow framework for machine learning applied to cancer research. BMC Bioinformatics 19(491) (2018)
33. Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M.J., Shenker, S., Stoica, I.: Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In: NSDI'12: Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation. pp. 2–2. San Jose, USA (2012)
34. Zaharia, M., Chowdhury, M., Franklin, M.J., Shenker, S., Stoica, I.: Spark: Cluster computing with working sets. In: HotCloud'10: The 2Nd USENIX Conference on Hot Topics in Cloud Computing. pp. 10–10. Boston, MA (2010)
35. Zhang, S., Boehmer, W., Whiteson, S.: Deep residual reinforcement learning. In: AAMAS '20: The 19th International Conference on Autonomous Agents and MultiAgent Systems. p. 1611–1619. Auckland, New Zealand (2020)