Memory-Efficient Data Structures for Learning and Prediction

Damian Eads
Wise.io, GE Digital
San Francisco, California
damian.eads@ge.com

Paul Baines
Wise.io, GE Digital
San Francisco, California
paul.baines@ge.com

Joshua S. Bloom
Wise.io, GE Digital
San Francisco, California
josh.bloom@ge.com

ABSTRACT

In many real-world machine learning systems, memory efficiency, data pipeline speed, and prediction latency are paramount. Here, we introduce two data structures—already in production in industrial machine learning settings—that aim to address these practical needs. EdgeFrame is a data frame optimized for memory-efficient machine learning on industrial data. It supports lightweight views that reuse the same components to minimize memory footprints. An EdgeFrame and Columns can be forked (with copy-on-write semantics) so that several components of a data pipeline can share the same data while avoiding side effects caused by local modifications. An EdgeFrame can be stored off-heap so it can be shared among multiple worker processes on the same machine. EdgeFrame can also be stored out-of-core so data sets larger than RAM can be handled. EdgeFrame supports arrays as a first-class data type so that image and sensor data can easily be represented in the same data set. We also introduce and motivate the representation of machine learning models in a parse-free format, allowing intelligent use of I/O bandwidth when loaded from disk. This greatly decreases the time to first prediction when the models are pulled from cold storage.

1 INTRODUCTION

In machine learning, accuracy is often the primary metric to compare techniques [2]. Training run time and prediction time are also extensively considered [9]. Though often not given as much attention, memory usage is critically important in the real world. Predictable memory usage is required to ensure reliable production operation as crashes due to inadequate memory cannot easily be recovered without provisioning new machines with more resources.

Excessive memory usage was a common problem among many participants in the AutoML challenge, and the problem persisted even when the available memory was doubled by the benchmark organizers [7]. Data partitioning during model construction is often a culprit for excessive copying of the same data sets. Moreover, the time it takes to deploy a model on a newly provisioned instance is often ignored entirely. The overheads of data storage, loading, and movement are also, typically, neglected from consideration.

To address the first issue of memory inefficiency, we introduce EdgeFrame, a mutable data frame implementation with several features designed to keep the memory footprint low and improve data movement and ingestion. To address the second issue, we propose flat models, a way of storing model data to minimize parsing, maximize effective use of I/O, and allow off-heap storage.

Data Frames are a pervasive throughout the data science, machine learning, and database communities. Pandas is the most popular and sophisticated data frame, but its memory overhead may limit the size of data sets that can be trained on a single machine. Moreover, it is not optimized for bare metal performance of machine learning [10]. The Dask project offers a distributed data frame within its framework [13]. SFrame's focus is on out-of-core processing [14]. Spark DataFrame packs each row as an element of an RDD. It is geared towards database queries on large clusters [1]. EdgeFrame supplements these projects by providing a high performance, memory-efficient data frame implementation optimized for bare metal performance on single machines where workers and stages can share the same copy of data components as possible.

2 EDGEFRAME

EdgeFrame is a memory-efficient data frame optimized for machine learning on industrial data. It is used to represent ephemeral training and prediction data sets. Our in-production implementation supports:

- **copy-on-write** forking: columns and data sets are copy-on-write so a deep copy is only necessary when a view is modified;
- **off-heap data storage**: flat layouts enable parts of an EdgeFrame to be stored off-heap for seamless data sharing between workers;
- **out-of-core**: blocks can be cached in tertiary storage (e.g., on-blade SSDs) to allow for data sets larger than RAM;
- **lightweight slicing**: both strided and non-contiguous slicing of an EdgeFrame results in a shallow view;
- **tensor data**, which is important for sensor-heavy data sets common in industrial problem domains (e.g., aircraft engines, power turbines, MRI machines); and
- **mixed sparse and dense features**: so that dense meta-data can be combined with sparse features.

Shallow Views and Copy-on-write Data Frames

The same data is often reused in multiple stages of a data pipeline. Changes made in one part of a build pipeline may inadvertently lead to side effects in other stages. Requiring a data frame to be copied between stages avoids this pathology at a high cost.

Most EdgeFrame operations generate a shallow copy – slicing `x` with a stride `a:b:s` or an index array `ind` generates a lightweight view:

```
y = x[a:b:s]
y = x[ind]
```

An EdgeFrame can be built from more than one EdgeFrame without a copy. For example,

```
z = hstack(x[a]["age"],
           x[b]["ZC"].rename("zip"), y[a])
```

generates a lightweight EdgeFrame `z` from `x` and `y` without a copy (`a` and `b` are the same length). The `rename` operation results in a new EdgeFrame Column view without a copy. We can generate three data sets from an input data set

```
train, validate, test = ds.stratified([8,1,1], "Y")
```

using stratified sampling (80% for training, 10% for validation, and 10% for testing) without a copy.

Until now, we have focused on immutable operations. An EdgeFrame or a Column can be 'forked' to create a new light-weight copy as in `y = x.fork()`.

This is akin to forking a source code repository in git[15] or a BSD process `vfork` [8]. An EdgeFrame or a Column fork allows side effect-free changes while reducing memory overhead caused by excessive copying. To demonstrate, we cloned 10^7 copies of a column (10^8 rows of type `uint8`, 100 MB) into a new EdgeFrame. Without EdgeFrame's shallow copy scheme, this data frame would be over a petabyte in dense form, but used approximately 2.2 GB, mostly to store pointers. The data set is forked and 4.9 GB is used. We then modified a column, but only a single new dense column was allocated.

Data Ingestion and Data Movement Production-grade data ingestion demands performance and non-ambiguity. An explicit schema reduces the possibility of type errors in a data pipeline due to incorrect type inference from the input data. Text formats such as CSV and JSON are notoriously CPU-bound. The effects of serialization are stark as we found experimentally with our parallel-text parsing project, Paratext [3]. We showed multi-threaded can lead to more effective use of overall I/O bandwidth [3]. Though binary formats are preferable, these benchmarks prompted us to profile our entire stack for bottlenecks.

The movement of a data frame or column from one machine to another often requires serialization. Serialization is a common source of overhead in data ingestion and communication between workers. A flat layout where data are not packed, the schema is strongly typed, and the offset of every field is known a priori can greatly reduce the cost of serialization[4, 5, 17]. Apache Arrow, a common data payload for interchanging data between tools throughout the big data ecosystem was founded to address this problem [11].

Off-heap Data If a worker requires data as input that is output from a previous stage of computation, it may become idle while it waits for its producer to serialize its result, and if so, the consumer will waste CPU to deserialize its input payload. For this reason, we have adopted flat layouts for data payloads within an EdgeFrame to maximize use of the I/O bandwidth. These payloads are stored off-heap in shared memory objects so multiple workers can share the same copy of a data set. Moreover, data lifetimes can be independent of process lifetimes.

EdgeFrame stores a column's scalar data in an Arrow array. Arrow is a multi-language array abstraction designed to share data payloads between different tools in the big data ecosystem. Non-scalar dtypes such as tensors can be stored individually in an arrow-backed array. This avoids serialization for a Column's contents. Data can be copied without parsing in a manner that makes effective use of the I/O bandwidth. To demonstrate interprocess sharing, we loaded a 14 GB dataset off-heap. Two processes were able to share the same data set with a negligible impact on RAM usage.

Out-of-core Storage and Caching Since an EdgeFrame is stored off-heap via a memory map, the page cache can be used to support data sets larger than RAM. Ephemeral SSDs serve as an intermediate caching layer. We have found in practice that AWS/EBS is a poor backing choice because it relies on network transport, which is higher latency and lower throughput relative to ephemeral SSDs.

Mixed Sparse & Dense EdgeFrame supports combining sparse and dense columns in the same frame. This is particularly useful for data sets with dense meta-data and sparse sensor readings.

	Msgpack	**Pickle**	**Parse-Free**
Cold write (s)	12.5	87	9.2
Latency to 1st prediction (s)	11.5	86	0.7

Figure 1: Cold I/O performance of a 1000 tree decision forest ensemble trained on MNIST (no depth limit). Setup: ext4 mounted on RAID-0 with 2 on-blade SSDs.

3 PARSE-FREE MODELS

Machine learning models are often serialized with PMML, Pickle, PFA or other file formats [6, 12, 16]. While convenient, deserialization can greatly increase the latency to first prediction when pulling a model from cold storage or scaling prediction worker nodes. We introduce parse-free models to address this problem. A parse-free model is much like a struct where fields of a struct may contain pointers to data or other structs. However, offsets to each field are stored instead of pointers because they are invariant to the starting address of the struct to which they belong. A heap-like sequence of pages (or **off-heap store**) is maintained as a memory-mapped file for each model. An off-heap store contains all the objects needed to represent a single model. It is a self-contained block that can be copied to disk, archived to the cloud, or sent over the network to other workers. When it is retrieved, all that is needed to deploy the model is a simple memory map operation. Figure 1 shows that parse-free models improve the latency to the first prediction by at least an order of magnitude over serialization-based approaches such Msgpack and Pickle. Note that the file system caches are flushed beforehand.

4 CONCLUSION

We introduce EdgeFrame: a memory-efficient data frame for machine learning on array-heavy, industrial data sets. EdgeFrame supports off-heap storage so that data can be exchanged between worker nodes with minimal serialization overhead and multiple worker processes can share the same copy of a data set. Multiple stages of a data pipeline can share the same underlying data and make changes with no side effects because of EdgeFrame's shallow views and copy-on-write semantics. Copies are delayed until a shared resource is modified. Tensors are a first-class data type in EdgeFrame so that sensor data can be handled seamlessly. Sparse and dense columns can be stored in the same EdgeFrame, a requirement for many industrial datasets. Lastly, we also extended the off-heap concept to our representation of machine learning models reducing the latency to first prediction significantly.

REFERENCES

[1] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. 2015. Spark SQL: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 1383–1394.

[2] Rich Caruana and Alexandru Niculescu-Mizil. 2006. An Empirical Comparison of Supervised Learning Algorithms. In *Proceedings of the 23rd International Conference on Machine Learning (ICML '06)*. ACM, New York, NY, USA, 161–168. https://doi.org/10.1145/1143844.1143865

[3] Damian Eads. 2016. ParaText. (2016). https://deads.gitbooks.io/paratext-bench/content/

[4] Ion Gaztanaga. 2012. Boost Interprocess. (2012).

[5] Google. 2015. FlatBuffers. (2015). http://google.github.io/flatbuffers/

[6] Alex Guazzelli, Michael Zeller, Wen-Ching Lin, Graham Williams, et al. 2009. PMML: An open standard for sharing models. *The R Journal* 1, 1 (2009), 60–65.

[7] Isabelle Guyon, Imad Chaabane, Hugo Jair Escalante, Sergio Escalera, Damir Jajetic, James Robert Lloyd, Núria Macià, Bisakha Ray, Lukasz Romaszko, Michèle Sebag, Alexander Statnikov, Sébastien Treguer, and Evelyne Viegas. 2016. A brief Review of the ChaLearn AutoML Challenge: Any-time Any-dataset Learning without Human Intervention. In *Proceedings of the Workshop on Automatic Machine Learning (Proceedings of Machine Learning Research)*, Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren (Eds.), Vol. 64. PMLR, New York, New York, USA, 21–30.

[8] William N Joy. 1983. *4.2 BSD system manual*.

[9] Yann Lecun, L.D. Jackel, Leon Bottou, Corinna Cortes, J. S. Denker, Harris Drucker, I. Guyon, U.A. Muller, Eduard Sackinger, Patrice Simard, and V. Vapnik. 1995. *Learning algorithms for classification: A comparison on handwritten digit recognition*. World Scientific, 261–276.

[10] W McKinney. 2014. Pandas, Python Data Analysis Library. 2015. *Reference Source* (2014).

[11] Jacques Nadeau, Todd Lipcon, and Ted Dunning. 2017. Apache Arrow. (2017). http://arrow.apache.org

[12] Jim Pivarski, Collin Bennett, and Robert L Grossman. 2016. Deploying analytics with the portable format for analytics (PFA). In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 579–588.

[13] Matthew Rocklin. 2015. Dask: parallel computation with blocked algorithms and task scheduling. In *Proceedings of the 14th Python in Science Conference*. 130–136.

[14] The Turi Team. 2016. SFrame. (2016). https://pypi.python.org/pypi/SFrame

[15] Linus Torvalds and Junio Hamano. 2010. Git: Fast version control system. (2010). http://git-scm.com

[16] Guido Van Rossum et al. 2007. Python Programming Language.. In *USENIX Annual Technical Conference*, Vol. 41. 36.

[17] Kenton Varda. 2015. Cap'n Proto. (2015). https://capnproto.org/