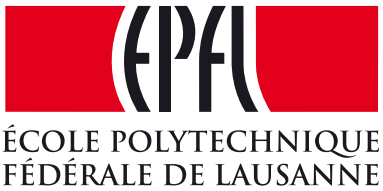


MACHINE LEARNING AND OPTIMIZATION LABORATORY, EPFL

Sparse Linear Algebra in the Deeplearning Framework

Audrey LOEFFEL

EPFL Professor:
Martin JÄGGI

Skymind Supervisor:
François GARILLOT

August 11, 2017

Acknowledgements

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Lausanne, 12 Mars 2011 D. K.

Abstract

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Key words:

Contents

Acknowledgements i

Abstract iii

List of figures vii

List of tables ix

1 Sparse Data and Formats 3
- 1.1 Definition . . . 3
- 1.2 The Advantages of Sparse Data . . . 3
- 1.3 Sparse Data are very common in Machine Learning . . . 4
  - 1.3.1 A Real Case of Sparse Dataset . . . 4
- 1.4 Solution: Encode the data into a Sparse Format . . . 4
- 1.5 Formats . . . 4
  - 1.5.1 Matrices . . . 4
  - 1.5.2 Tensors - Multi-dimensional arrays . . . 6

2 The Deeplearning4j Library 7
- 2.1 Architecture of the library . . . 7
- 2.2 The Importance of Nd4j in the Library . . . 7
- 2.3 Nd4j needs a Sparse Representation . . . 8

3 Structure of an Multi-dimensional Array 9
- 3.1 Storing an Array . . . 9
  - 3.1.1 Data Buffer . . . 10
  - 3.1.2 Parameters of an Array . . . 10
- 3.2 Views . . . 10
- 3.3 Indexes . . . 11
- 3.4 Operations . . . 11

4 Implementation 13
- 4.1 Hierarchy of Arrays . . . 13
- 4.2 Limitations and Constraints . . . 14
  - 4.2.1 storing off-heap . . . 14

4.2.2 Workspaces . . . 15
4.2.3 DataBuffers have a fixed length . . . 15
4.3 CSR Matrices . . . 15
4.3.1 Structure . . . 15
4.3.2 Put a value . . . 15
4.3.3 Get a Sub-array . . . 15
4.3.4 Limits with this format . . . 16
4.4 COO Tensors . . . 16
4.4.1 Naive implementation . . . 16
4.4.2 More parameters are needed to define the tensors . . . 18
4.4.3 Computations of the the Parameters . . . 19
4.4.4 Sparse Indexes Translation . . . 21

5 Operations **23**
5.1 Backends . . . 23
5.2 BLAS . . . 23
5.2.1 Level 1 in CSR Matrix . . . 23
5.2.2 Level 1 in COO Tensor . . . 23
5.3 Libnd4j . . . 23

6 Results **25**
6.1 l . . . 25

7 Conclusion **27**

A An appendix **29**

Bibliography **31**

List of Figures

1.1 A matrix stored in COO format 5
1.2 A matrix stored in CSR format 6
1.3 A tensor stored in COO format 6

2.1 Nd4j architecture 8

3.1 Comparison between C-order and F-order 9
3.2 View shares memory with the original array 11

4.1 Arrays hierarchy in Nd4j 14
4.2 Illustration of the different possible datastructure for storing the indexes [0, 2, 1] of a value v 17
4.3 A 3×3 matrix with a 2×2 view in grey 17
4.4 Result of *point(0)* index in grey on a $2 \times 3 \times 3$ tensor 18

List of Tables

Introduction

1 Sparse Data and Formats

Definition

Data are said sparse when it is contains only a few non-null values. That kind of dataset are really common in Machine Learning application and can be an high influence on the computation.

The sparsity of a dataset is defined by :

$$sparsity = \frac{\text{\# non-null values}}{\text{\# values}} \tag{1.1}$$

Conversely when a dataset has only a few null values, the data are said dense. The density of the dataset is defined by the inverse of the sparsity:

$$density = \frac{1}{\text{sparsity}} \tag{1.2}$$

Using dense methods and data structure with sparse data could have a severe bad impact on the performance

The Advantages of Sparse Data

Sparsity is a very useful property in Machine Learning. Some algorithms can have fast optimization, fast evaluation of the model, statistical robustess or other computational advantages. A lot of machine learning application are using sparse dataset such as recommender system, natural language processing algorithm,

Sparse Data are very common in Machine Learning

In Machine Learning it's very common to deal with sparse dataset. We can encounter them in any kind of applications: Natural Language Processing, Retrieving Systems, Recommender Systems, etc.

Given the possible optimization that sparse dataset allows and the high number of people that could take advantages of it, it becomes important to add the support of sparse data in Nd4j.

A Real Case of Sparse Dataset

In 2008 Netflix launched a contest, the Netflix Grand Prize [net()], to improve their recommender system model and to increase the accuracy of predictions and published an sample dataset made with the ratings of anonymous Netflix customers. The dataset had more than 100 millions sampled ratings and it contained about $m = 480'186$ users and $m = 17'770$ movies [Koren(2009)]. If stored as a dense matrix, it would need to store $8'532'905'220$ values in memory. That corresponds to a sparsity $\cong \frac{100'000'000}{8'532'905'220} = 0.011719338$.

Storing more than 8 trillions 64-bit floating-point numbers needs more than 64 gigabyte of memory which quickly become unmanageable even for the world's fastest supercomputers.

Solution: Encode the data into a Sparse Format

To avoid the issue due to the high volume of storage needed, we must store the data into a sparse format. There are different kind of formats which each of them is more suitable to different aspects (Storage vs computation).

Formats

There are many methods for storing sparse data, each of them presents different advantages and disadvantages.

Matrices

Coordinates Format

It is the simplest method to encode a sparse array. The coordinates and the value of each non-zero entry are stored in arrays. Typically each element are encoded in a tuple (row, column, value)

Some implementation variations of the COO format exist. The elements can be sorted along a dimension, or it can have duplicate indexes.

$$A_{(M\times N)} = \begin{bmatrix} 0 & 2 & 0 \\ 0 & 0 & 3 \\ 1 & 0 & 4 \\ 0 & 0 & 0 \end{bmatrix} \quad \rightarrow \quad \begin{aligned} Values_{(1\times NNZ)} &= \begin{bmatrix} 2 & 3 & 1 & 4 \end{bmatrix} \\ Rows_{(1\times NNZ)} &= \begin{bmatrix} 0 & 1 & 2 & 2 \end{bmatrix} \\ Columns_{(1\times NNZ)} &= \begin{bmatrix} 1 & 2 & 0 & 2 \end{bmatrix} \end{aligned}$$

Figure 1.1: A matrix stored in COO format

This format provides an easy and fast way to retrieve a value and to insert a new non-zero element. It's also fast and simple to convert into a dense format.

But this format is not the most efficient regarding the memory consumption.

Compressed Row Format

The Compressed Row and the Compressed Column formats are the most general format to store a sparse array. They don't store any unnecessary element conversely to the COO format. But it requires more steps to access a element than the COO format.

Each non-zero element of a row are stored contiguously in the memory. Each row are also contiguously stored.

The format, described by the Intel MKL Sparse Library [mkl()], requires four arrays:

Values All the nonzero values are store contiguously in an array. The array size is NNZ.

Column pointers This array keeps the column position for each values.

Beginning of row pointers Each pointer i points to the first element of the row i in the values array. The array size is the number of rows of the array.

End of row pointers Each pointer i points to the first element in the values array that does not belong to the row i . The array size is the number of rows of the array.

Compressed Column Format

The Compressed Column Format is similar to CSR but it compresses columns instead of rows.

Given a matrix $N \times M$, the pointers arrays will have a size M.

$$A_{(N\times M)} = \begin{bmatrix} 0 & 2 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 4 & 0 \\ 0 & 0 & 2 & 1 \end{bmatrix} \rightarrow \begin{aligned} Values_{(1\times NNZ)} &= \begin{bmatrix} 2 & 3 & 1 & 4 & 2 & 1 \end{bmatrix} \\ Columns_{(1\times NNZ)} &= \begin{bmatrix} 1 & 2 & 0 & 2 & 2 & 3 \end{bmatrix} \\ pointersB_{(1\times N)} &= \begin{bmatrix} 0 & 1 & 2 & 2 & 4 \end{bmatrix} \\ PointersE_{(1\times N)} &= \begin{bmatrix} 1 & 2 & 2 & 4 & 6 \end{bmatrix} \end{aligned}$$

Figure 1.2: A matrix stored in CSR format

Tensors - Multi-dimensional arrays

A tensor is a multi-dimensional array. The order of the tensor is the dimensionality of the array needed to represent it. Matrices and vectors can be represented as tensors where the order is equals to 2 and 1 respectively.

This generalization allows a more generic implementation of a n-dimensional array in the Nd4j library.

Coordinates Format

The COO format can easily be extended to encode tensors by storing an array of indexes instead the row and column coordinates.

A array of order $K = 3$ with shape $N \times M \times P$ which has the following non-zero values :

value	indexes
1	0 1 0
2	1 1 2
3	1 2 0
4	2 0 1
5	2 2 0

can be encoded with one values array and one indexes array :

$$\begin{aligned} Values_{(1\times NNZ)} &= \begin{bmatrix} 1, & 2, & 3, & 4, & 5 \end{bmatrix} \\ Indexes_{(NNZ\times K)} &= \begin{bmatrix} [0,1,0], & [1,1,2], & [1,2,0], & [2,0,1], & [2,2,0] \end{bmatrix} \end{aligned}$$

Figure 1.3: A tensor stored in COO format

2 The Deeplearning4j Library

Deeplearning4j is a open-source Deep Learning library for the JVM. It runs on distributed CPU's and GPU's.

–> TODO

Architecture of the library

The library is composed by several sub-libraries:

Deeplearning4j provides the tools to implement neural networks and build computation graphs

Nd4j is the mathematical back-end of Deeplearning4j. It provides the data structures for the n-dimensional arrays and allow Java to access the native libraries via JavaCPP and the Java Native Interface.

Libnd4j is the computing library that provides native operations on CPU and GPU. It's written in C++ and Cuda.

Datavec provides the operations for the data processing such that data ingestion, normalization and transformation into feature vectors.

The Importance of Nd4j in the Library

Nd4j is at the base of the Deeplearning4j library, it provides data storage, manipulations, and operations. It gives the atomic pieces needed to build more complex deep learning systems. Nd4j stands for N-Dimensional Arrays for Java and is basically a scientific computing library for the JVM. It features n-dimensional array object and the support of CPU and GPU via Cuda.

The APIs provided by the library are essentially wrappers for the different version of BLAS

(Basic Linear Algebra Subprogram).

BLAS is a specification that defines the low-level routines for linear algebra operations (for vectors and matrices). There exist several libraries implementing those subroutines in C or Fortran for dense or sparse formats. In Nd4j the BLAS subroutines can directly be called from Java thanks to JavaCPP, that internally uses the Java Native Interface (JNI) to call native routines from the JVM environment. This architecture allows the library to benefit from the advantages of the native side.

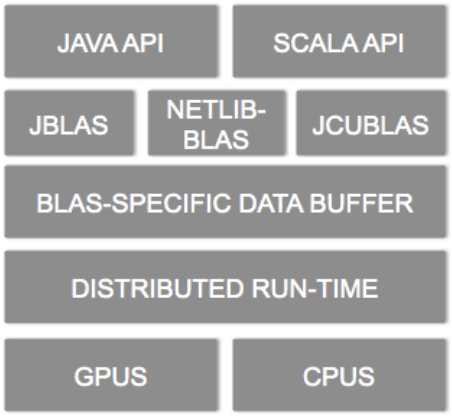


Figure 2.1: Nd4j architecture

Nd4j needs a Sparse Representation

Currently in Deeplearning4j, Sparse Data are treated as dense and use the dense operations of BLAS and Libnd4j to perform computations. With a new sparse representation we can gain in storage space and computation speed.

TODO -> develop

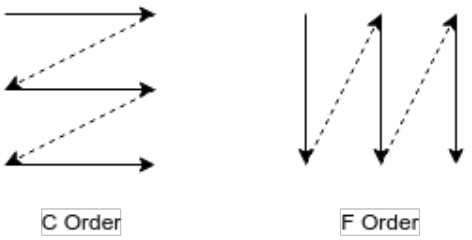
3 Structure of an Multi-dimensional Array

The new sparse array have to be compliant with the API and inter-operable with the current dense array implementation.

Storing an Array

A dense array is stored as a single contiguous block of memory, flatten in a one-dimensional array. Arrays are stored off-heap (outside the JVM environment). The reasons behind this design decision are numerous: better performance, better interoperability with BLAS libraries, and to avoid the disadvantages of the JVM such as the limited size of arrays due to the integer indexing (limited to $2^{31} - 1 \cong 2.14$ billion elements)

There are two methods to store a multi-dimensional array into a linear memory space: row-major order (C) or column-major order (Fortran). Figure 3.1 shows how a two-dimensional array is stored according to the order.



$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{bmatrix} \Rightarrow \begin{array}{l} C-Order = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{21} & a_{22} & a_{23} \end{bmatrix} \\ F-Order = \begin{bmatrix} a_{11} & a_{21} & a_{12} & a_{22} & a_{31} & a_{13} \end{bmatrix} \end{array}$$

Figure 3.1: Comparison between C-order and F-order

The data are accessed via strides which define how to index over contiguous block of data. For each dimension it defines by how many values two consecutive elements are separated. In the case of matrix A defined in figure 3.1, the strides would be $(3,1)$ in case of C-order and $(1,3)$ in

case of F-order. Strides $(3,1)$ means that each row is separated by 3 values and each column is separated by 1 value.

Data Buffer

The DataBuffer is a storage abstraction which provides optimal storage and retrieval depending on the backend. The data are stored off-heap through JavaCPP. It is basically a wrapper around a pointer and an indexer with utility methods to access and modify the data. The pointer points to the allocated memory space and the indexes provides an easy-to-use and efficient way to access a multi-dimensional memory space.

The implementation of the databuffer depends on the data type, because of the length needed to store a single value (int -> 32 bits, long -> 64bits, float -> 32bits, double -> 64bits)

Parameters of an Array

The information about the shape of the array are grouped in a DataBuffer object called ShapeInformation. It groups these following information:

Rank	The number of dimension of the array.
Shape	The shape of the array.
Strides	Provides information about the logical layout of the array for each dimension.
Offset	Provides the position of the first value of the data array that belongs to the array or view.
ElementWiseStride	Indicates how two contiguous elements are physically separated in memory.
Order	C or F order

Views

The data in memory can be shared by multiple NDArrays. An NDArray can refer to a subset of another NDArray. We say that such an array is a view of the original array. This is a powerful concept that avoid the unnecessary copy of the data which is a very expensive operation.

Since the memory space is shared, changes to one will impact the other ones. Figure **??** illustrates how a view shares its data with its original array.

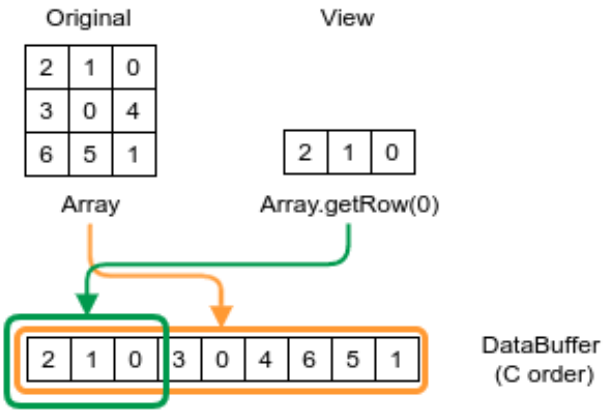


Figure 3.2: View shares memory with the original array

Indexes

NDArrays can be accessed through a combination of indexes. There are many way to index an array. For each dimension of the original array, we can specify if we want everything, an interval or a specific value of it. It gives a lot of flexibility to access sub-part of a array.

The indexes provide an efficient mechanism to access Tensors along dimension. A tensor along a dimension (TAD) is a view of an actual tensor with a lower rank.

Operations

Nd4j defines five types of operations:

1. Scalar: element-wise operations such as addition or multiplication where a scalar is applying on an ndarray.

2. Transform: in-place operations such as logarithm or cosine. There are usually executed in an element-wise manner but it's not always the case; some can be applied along a dimension.

3. Accumulation: Also called Reduction. Operations applied over the whole ndarray or along a dimension such as the sum of all values.

4. Index Accumulation: Similar to accumulation operations but return an index instead of a value. A classic operation is getting the maximum or minimum value along a dimension.

5. Broadcast: Some of the more useful operations are vector operations, such as *addRowVector* which add a row vector to every row of a matrix.

4 Implementation

Hierarchy of Arrays

The API of an array is defined by an interface called *INDArray* which has a dense implementation for each backend: *NDArray* class for the CPU and *JCublasNDArray* class for the GPU. But since most of the operations and methods are shared between the two backends, they are implemented in an abstract class called *BaseNDArray*.

Adding sparse representations asked two questions:

1. What can be shared with the dense arrays ?
2. What can be between the different sparse arrays and what are format-specific ?

To answer those questions, we need to go a little bit deeper in the implementation.

The dense implementation includes methods that are inherently related to the way dense array is internally made, and other methods are related to the generic parameters such as the shape or are utility method.

The first type of methods is not useful in case of sparse. Dense and sparse arrays are not built in the same manner. Methods such as *getStrides* are not relevant in the sparse context. Reciprocally the sparse array will need methods which will be irrelevant in the dense context.

We encounter the same situation between the different sparse formats. Some will need utility methods that the other ones won't need.

But everything has to be defined in the *INDArray* interface. To avoid code duplication, everything than can be shared should be implemented in the higher level of the hierarchy. The methods that are not compatible with a type of array will simply throw an unsupported operation exception.

The drawback brought with this solution is that we always need to verify the type of the array

before doing any operations.

// TODO update schema

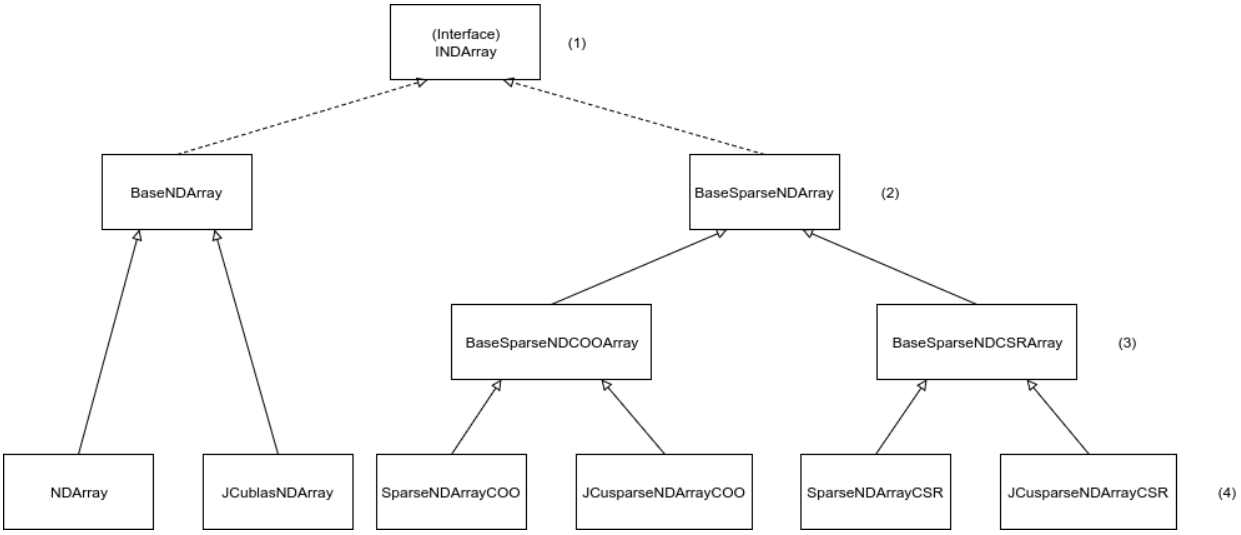


Figure 4.1: Arrays hierarchy in Nd4j

Limitations and Constraints

Nd4j has been made in the perspective of dense arrays. The design has been thought and optimized regarding the dense implementation which brings some constraints to implement the sparse representation

storing off-heap

// TO ASK TO RAVER: What are the advantages for storing data off-heap ? Why Garbadge collector is a bad thing ?

What the idea behind the workspaces and how does it work ?

Before the workspaces, data were already stored off-heap. What do workspaces bring new ? How could

Why it wasn't possible to have arrays of databuffers native side ? what we had to flatten everything?

Workspaces

DataBuffers have a fixed length

explain what is in the ToC doc

——— -> we can reallocate memory from java side but it's impossible from native side. so we need to overallocated to the max result size before any operations !

1 - estimate the size needed (only the size of the view -> which avoid to allocated at the max size of the array - it wouldn't fit in memory) 2 - reallocate 3 - perform the op

-> add in op interface

CSR Matrices

Structure

The representation uses 4 data buffers to encore a CSR matrix. One for the non-zero values, one for the columns indexes and two for the row pointers (to the beginning and to the end of each row).

Put a value

To insert a new value or to update an existing non-null value, we need to identify where the values of row we want ot insert to are located in the values buffer and in the columns buffer. The beginning and end of rows pointers give us the range of indexes.

While iterating over the range of values, if we find a value with the same column index than the one we want to insert to, we can update the values and nothing needs to be changed in the three other buffer.

However if there is no value with that column, we need to insert a new one at the correct position. Then we need to update the end of row pointer for this row. Finally each row pointers that come after need to be increment by one.

– ADD pseudo code for putscalar of csr

Get a Sub-array

1. First, we need to resolve the index to get the new shape of the array, the new rank, etc. In the case we use the resolution of the dense array but we are only interested in

    - the shape: an array with two elements containing the new shape of the sub-array.

- the offsets: an arays with two elements that indicate the first row and the first column that belongs to the sub-array.
- the offset: indicates the position in the data buffer of the first element that belongs to the sub-array.

2. Sometimes the offsets are equals to zeros while having a non-null offset. In this case we need to override the offsets.

```
1       offsets[0] = (int) resolution.getOffset() / shape()[1];
2       offsets[1] = (int) resolution.getOffset() % shape()[1];
```

3. With the offsets we can now compute the first and the last position of each dimension.

```
1      long firstRow = offsets[0];
2      long lastRow = firstRow + shape[0];
3      long firstColumn = offsets[1];
4      long lastColumn = firstElement + shape[1];
```

4. Now we have the bounds for each dimension, we can reconstruct the new beginning and end of row pointers et the columns indexes.

Limits with this format

This formats only works with two dimensions and cannot be extended to tensors. Therefor it makes it difficult to be compatible with the API. Moreover the operations to get or put values aren't straightforward. Several step are necessary before accessing the value.

COO Tensors

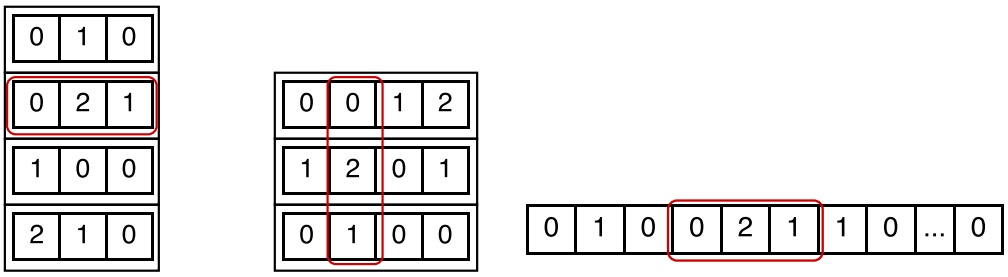
Naive implementation

Based on the description in 1.5.2 the COO encoding needs one data buffer to store all the non-null values and one for the indexes of each values.

An easy solution would have been to store the indexes into a multi-dimension array of *DataBuffers*: One buffer for each value, or one buffer for each dimension. Due to the native constraints that makes hardly manageable to have such arrays (Difficulty to pass the array to the native side and Cuda side), we choose to flatten the indexes into one buffer.

But this implementation makes difficult to be compliant with the API. It brings several issues:

- The key of views is the sharing of their data. In case of COO format, views have to share the data buffer and the indexes buffer. Without the indexes it is not possible to add a value in the original array by adding it in a view. If we only put the new value in the shared value buffer without updating the indexes, the original array would have a value



(a) each index is stored contiguously

(b) Each dimension is stored contiguously

(c) Indexes are flatten

Figure 4.2: Illustration of the different possible datastructure for storing the indexes [0, 2, 1] of a value v

buffer bigger than its indexes buffer and there would be an offset between the values and the indexes.

Even when sharing both buffers, how would we know which value is included in the view and which is ot?

- The coordinates of a value in a view are not necessary the sames of the same value in the original array.

  They can be offset if dimension is partially included in the view. Figure 4.3 shows a matrix an a view (in red). The value $v_i = 5$ would have the coordinates [1, 1] in the original array while it has the coordinates [0, 0] in the view. How the indexes can be translated between views?

1	2	3
4	5	6
7	8	9

Figure 4.3: A 3×3 matrix with a 2×2 view in grey

- A view can have a lower or higher rank than its original array. How can the view indexes be stored ?

More parameters are needed to define the tensors

Most of the issues cited in section 4.4.1 are due to the support of the different types of indexes. Each index implements the interface *INDArrayIndex* and extends from *NDArrayIndex*. They provide a very efficient and powerful mechanism to access part of a array but they introduce some constraints when implementing views for COO.

All Indexes

All indexes are the most straightforward of the library. They are used to collect all the elements of a dimension.

Interval Indexes

Interval indexes takes an subpart of a dimension containing in an interval. They don't modify the rank

The grey sub-array in figure 4.3 is the result of the operation :

```
1     myArray.get(NDArrayIndex.interval(1, 3), NDArrayIndex.interval(1, 3));
```

After the resolution of the indexes, we obtain offsets equal to $[1,1]$ with an offset equals to 4. In the COO perspective that means a value having any of its dimensions equal to 0 does not belong to the view. We need to define bounds for each dimension in order to be able to filter out the values.

Point Index

Point indexes take one unique element of a dimension. They reduces the rank of the array.

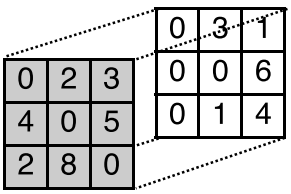


Figure 4.4: Result of *point(0)* index in grey on a $2 \times 3 \times 3$ tensor

Figure 4.4 shows a $2 \times 3 \times 3$ tensors with a 2×2 matrix view which is the result of the operations:

```
1     myArray.get(NDArrayIndex.point(0));
```

The coordinates of the resulting matrix have only two dimensions instead of three. We have an issue when trying to access a value given a pair of coordinates in the view context. There is no direct matching between those coordinates and those actually in the indexes buffer.

The solution is to add an additional parameter array that keep track of the status of each dimension. The array is called flags, it can contains either 0, which means *active*, or 1 which means *fixed*. The flags array for the view shown in figure 4.4 would be equal to [1, 0, 0] because the first dimension is fixed at position 0.

Specified Index

When using at least one specified index in the set of indexes used to get a sub-array, it always returns a copy of the original data. It can not be a view because the specified indexes are not deterministic and can not be translated into logical strides, shape or offsets.

In this case we need to iterate over each dimension to access every element of the array and test if the current value belongs to the view. If it is the case, we add it in a new array. We are still using the indexes resolution to get the new shapes, the offsets, etc.

New Axis Index

New axis indexes are used to add a new dimension to the array. The new dimension always has a length equals to 1. It can be perpended or inserted in the middle of the dimensions. Since the rank is higher, more cooridinates are needed to access a value. However the shared indexes buffers is limited to the original rank. Similary to the point index, we need a new parameter array that keep tracks of the position of the new dimensions to be able to translate the coordinates from view to original context.

Assuming we have a 3×3 matrix: calling *myArray.get(NDArrayIndex.newAxis())* will prepend a new dimension. The view is a $1 \times 3 \times 3$ tensor with a hidden dimension parameter array equal to [0].

Computations of the the Parameters

Computation of the Sparse Offsets

The sparse offsets are computed from the offset. The offset give us the position of the first element in the array. We want to reverse it the coordinates of this values which will be the sparse offset.

1. We need an array with a length equals to the view rank.
2. For each dimension except the innermost one, we divide the offset by the number of elements in one dimension's element (i.e divide the offset by the number of value in a row). Rounded to the lower integer, this quotient give us the sparse offset for thie dimension. Then we need to remove the number of elements that are in the same dimension but with a lowest value. We want to isolate the dimension's value (to get only

a row).

3. We iterate until the last element. In this case we have the offsets set up up to the row dimension. To find which column the value is in, we need to take the modulo of the remaining offset by the number of columns.

4. We have the sparse offsets given a set of indexes. But it does not take into account the possibly already non-null sparse offsets. This is the case when we take a view from a view. In this case we need to merge the two sparse offsets arrays.

5. We should particularly be careful with hidden dimensions because they are absent from the sparse offset resolution explained above. The result does not contain any information about the hidden dimension, we need to add them in a new array which has a length equals to the underlying rank. The sparse offsets of the active dimensions are simply added.

Algorithm 1 Calculate the sparse offsets

function CREATESPARSEOFFSETS(int $offset$)

▷ Compute the new offsets

$newOffsets \leftarrow new\ int[rank]$
for $i = 0$ to $(rank-2)$ **do**
  $nbElements \leftarrow \prod_{j=i+1}^{rank} shape[j]$
  $newOffsets[i] \leftarrow \lfloor offset \div nbElements \rfloor$
  $offset \leftarrow offset - newOffsets[i] \times nbElements$
end for
$newOffsets[rank-1] \leftarrow offset \mod shape[rank-1]$

▷ Merge with sparseOffsets of this array

$finalOffsets \leftarrow new\ int[underlyingRank]$
$active \leftarrow 0$
for $i = 0$ to $underlyingRanke$ **do**
  if $flags[i] == 0$ **then**
    $finalOffsets[i] \leftarrow sparseOffsets[i]$
  else
    $finalOffsets[i] \leftarrow newOffsets[active] + sparseOffsets[i]$
    $i \leftarrow i+1$
  end if
end for
return $finalOffsets$

Computation of the Flags

Flags determine which dimension is active and which one is hidden from the point of view of the array. The dimension can only be reduced with point index, which means the flags can be

computed during the offset resolution. We fill the flags array while iterating over the indexes array.

Computation of the Hidden Dimensions

The resolution returns an array containing the position of the newAxis indexes in the indexes array. The array may already have some hidden dimensions. In this case we need to adapt the result with the current hidden dimension.

Algorithm 2 Calculate the hidden dimensions

function CREATEHIDDENDIMENSIONS(int[] $newAxis$)

if $newAxis$ is empty or null **then return** $hiddenDimensions$
end if

if $hiddenDimensions$ is empty **then return** $newAxis$
end if

$size \leftarrow newAxis.length + hiddenDimensions.length$ ▷ Merge both arrays
$newArray \leftarrow$ new $int[size], i \leftarrow 0, newDim \leftarrow 0$
for $oldDim = 0$ to $hiddenDimensions.length$ **do**
  $j \leftarrow 0$
  while $newAxis[newDim] \leq hiddenDimensions[i]$ **do**
    $newArray[i] \leftarrow newAxis[newDim]$
    $i \leftarrow i+1, j \leftarrow j+1$
  end while
  $newArray[i] \leftarrow hiddenDimensions[oldDim] + j$ ▷ add an offset corresponding to the amount of axis added before this dimension
  $i \leftarrow i+1$
end forreturn $newArray$

Sparse Indexes Translation

..

5 Operations

Backends

BLAS

Level 1 in CSR Matrix

Level 1 in COO Tensor

Libnd4j

..

6 Results

..

l

..

7 Conclusion

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

A An appendix

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Bibliography

[mkl()] Intel MKL sparse format. https://software.intel.com/en-us/mkl-developer-reference-c-sparse-blas-csr-matrix-storage-format. [Online; accessed 8-August-2017].

[net()] Netflix grand prize. http://www.netflixprize.com. [Online; accessed 8-August-2017].

[Koren(2009)] Yehuda Koren. 1 the bellkor solution to the netflix grand prize, 2009.