

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Master Thesis

Sparse Linear Algebra in the Deeplearning4j Framework

Author

Audrey LÖFFEL
audrey.loeffel@epfl.ch

August 15, 2017

Supervisors

Prof. Martin JÄGGI
MLO | EPFL
martin.jaggi@epfl.ch

François GARILLOT
Skymind Inc.
francois@skymind.io



Abstract

In this work we present an implementation of sparse multi-dimensional array and a set of operations for the Deeplearning4j ecosystem. We study different sparse storage formats and their possible implementation into the library to be compliant with the existing API. We also present the integration of the accelerated linear algebra operations with the BLAS routines and the native library of DL4J.

... develop !

Contents

Abstract	i
List of figures	vii
1 Introduction	1
Introduction	1
1.1 Sparse Data is very common in Machine Learning	1
1.2 The Advantages of Sparse Data	1
1.2.1 A Real Case of Sparse Dataset	2
1.3 Solution: Sparse Representation and Operations	2
2 Sparsity and Formats	3
2.1 Definition	3
2.2 Formats	3
2.2.1 Matrices	3
2.2.2 Tensors - Multi-dimensional arrays	5
3 The Deeplearning4j Library	7
3.1 Architecture of the library	7
3.2 The Importance of Nd4j in the Library	7
3.3 Nd4j needs a Sparse Representation	8
4 Structure of an Multi-dimensional Array	9
4.1 Storing an Array	9
4.1.1 Data Buffer	10
4.1.2 Parameters of an Array	10
4.2 Views	10
4.3 Indexes	11
4.4 Operations	11
5 Implementation	13
5.1 Hierarchy of Arrays	13
5.2 Limitations and Constraints	14
5.2.1 Storing off-heap	14
	iii

Contents

5.2.2	Workspaces	14
5.2.3	DataBuffers have a fixed length	15
5.3	CSR Matrices Implementation	15
5.3.1	Structure	15
5.3.2	Put a value	15
5.3.3	Get a Sub-array	16
5.3.4	Limits of this format	17
5.4	COO Tensors	17
5.4.1	Naive implementation	17
5.4.2	Reverse coordinates	18
5.4.3	Put a Value	19
5.4.4	More parameters are needed to define the tensors	19
5.4.5	Sparse Indexes Translation	22
5.4.6	Computations of the the Parameters	22
5.4.7	Final Implementation	25
5.4.8	Get a Sub-Array	26
6	Operations	29
6.1	Basic Linear Algebra Subprograms (BLAS)	29
6.2	Backends	30
6.2.1	In-Place Routines	30
6.2.2	Level 1 Routines	31
6.2.3	Level 2 and Level 3 Routines	31
6.3	Libnd4j	32
7	Results	33
7.1	Storing a huge data set	33
7.2	Operations performance	33
7.2.1	level 1	33
7.2.2	level 2	33
8	Future	35
8.1	Operations	35
8.2	Support of the GPU backend	35
8.3	Make the sparse array API compliant	35
8.4	Support More Sparse Formats	35
8.5	Tensor contraction indexing	36
8.6	Optimization	36
9	Conclusion	39
A	Code Snippets	41
A	Sparse Offset - An execution example	43

A Indexes Translation - An execution example	45
Bibliography	47

List of Figures

2.1	A matrix stored in COO format	4
2.2	A matrix stored in CSR format	5
2.3	A $3 \times 3 \times 3$ tensor	5
2.4	A tensor stored in COO format	6
3.1	Nd4j architecture	8
4.1	Comparison between C-order and F-order	9
4.2	View shares memory with the original array	11
5.1	Arrays hierarchy in Nd4j	14
5.2	Illustration of the different possible datastructure for storing the indexes $[0, 2, 1]$ of a value ν	17
5.3	A 3×3 matrix with a 2×2 view in grey	18
5.4	A $2 \times 3 \times 3$ tensor with a slice along the first dimension in grey	20
5.5	Example of a specified index behavior on a matrix	21
5.6	A tensor and a view of it	26
5.7	Parameters defining the view T_{view} in 5.6b	26
6.1	A vector ν and its compressed form representation	31

1 Introduction

Linear algebra forms the backbone of many machine learning algorithms, especially in deep learning algorithms. It provides structures such as vectors and matrices to hold the data and parameters to perform operations between them.

Data is said sparse when it contains only a few non-zero values. That kind of dataset is really common in Machine Learning applications and can have a high influence on the computation.

Sparse Data is very common in Machine Learning

Many of machine learning applications tend to deal with sparse dataset. An example is the natural language processing applications, as stated by the Zips' law [1], the frequency of any word is inversely proportional to its rank in the frequency table. ...

– TODO how is it used ? frequency table as dataset?

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 the DL4J suite of libraries.

Moreover nowadays deep-learning is a highly competitive field. Many libraries exist or are being built in many languages. However Deeplearning4j has a prominent position in this ML-library war since it runs on the Java Virtual Machine (JVM) and is company and production-oriented.

The Advantages of Sparse Data

Linear algebra is a branch of mathematics that is widely used throughout science and engineering. Machine learning and more specifically deep learning algorithms rely on linear

Chapter 1. Introduction

algebra. Sparsity is a very useful property in Machine Learning. Some algorithms can have fast optimization, fast evaluation of the model, statistical robustness or other computational advantages.

A Real Case of Sparse Dataset

In 2008 Netflix launched a contest, the Netflix Grand Prize [2], to improve their recommender system model and to increase the accuracy of predictions. They published a 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 [3]. 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: Sparse Representation and Operations

Even if the compression of sparse data is very efficient when storing as dense data, we could gain in performance during the processing by using a sparse representation with sparse operations.

In sparse operations only the values are accessed and processed while in dense operations, every cell is accessed even if it contains a zero value.

To reduce the size of the memory needed and to gain in performance, 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).

2 Sparsity and Formats

Definition

The sparsity of a dataset is defined by :

$$sparsity = \frac{\# \text{ non-zero values}}{\# \text{ values}} \quad (2.1)$$

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:

$$density = \frac{1}{sparsity} \quad (2.2)$$

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 is encoded in a tuple (row, column, value)

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.

$$A_{(M \times N)} = \begin{bmatrix} 0 & 2 & 0 \\ 0 & 0 & 3 \\ 1 & 0 & 4 \\ 0 & 0 & 0 \end{bmatrix} \rightarrow \begin{aligned} \text{Values}_{(1 \times NNZ)} &= [2 \quad 3 \quad 1 \quad 4] \\ \text{Rows}_{(1 \times NNZ)} &= [0 \quad 1 \quad 2 \quad 2] \\ \text{Columns}_{(1 \times NNZ)} &= [1 \quad 2 \quad 0 \quad 2] \end{aligned}$$

Figure 2.1: A matrix stored in COO 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.

Similarly to the row-major ordering explained in section 4.1, this format stores the values by row and use pointers to differentiate each row.

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 [4], 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. The values are stored in a column-major ordering, as explained in section 4.1.

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

$$A_{(N \times M)} = \begin{bmatrix} 0 & 2 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 4 & 0 \\ 0 & 0 & 2 & 1 \end{bmatrix} \rightarrow \begin{aligned} \text{Values}_{(1 \times NNZ)} &= [2 \ 3 \ 1 \ 4 \ 2 \ 1] \\ \text{Columns}_{(1 \times NNZ)} &= [1 \ 2 \ 0 \ 2 \ 2 \ 3] \\ \text{pointersB}_{(1 \times N)} &= [0 \ 1 \ 2 \ 2 \ 4] \\ \text{PointersE}_{(1 \times N)} &= [1 \ 2 \ 2 \ 4 \ 6] \end{aligned}$$

Figure 2.2: A matrix stored in CSR format

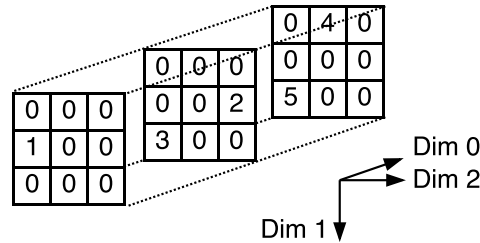
Tensors - Multi-dimensional arrays

A tensor is a multi-dimensional array. The order of the tensor is its number of dimension. It corresponds to the number of indexes needed to index a value. Matrices, vectors and scalars can be represented as tensors where the order is equals to 2, 1 and 0 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.

Figure 2.3: A $3 \times 3 \times 3$ tensor

The tensor, illustrated in figure 2.3, of order $K = 3$ with shape $3 \times 3 \times 3$ 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]

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

$$\begin{aligned} \text{Values}_{(1 \times NNZ)} &= [1, \ 2, \ 3, \ 4, \ 5] \\ \text{Indexes}_{(NNZ \times K)} &= [[0, 1, 0], \ [1, 1, 2], \ [1, 2, 0], \ [2, 0, 1], \ [2, 2, 0]] \end{aligned}$$

Figure 2.4: A tensor stored in COO format

3 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.

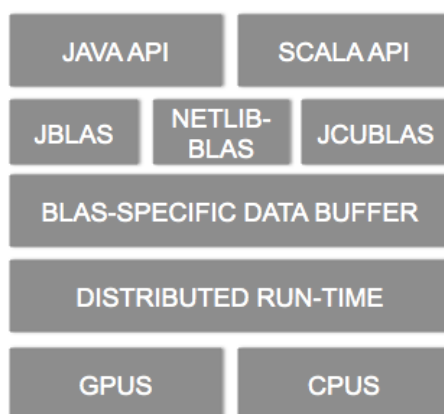


Figure 3.1: Nd4j architecture

Nd4j needs a Sparse Representation

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.

Currently there is no library in the JVM ecosystem that supports the sparse operations. Despite the fact that the Breeze (a numerical processing library for Scala) library [5] support sparse vectors and sparse CSR matrices but no sparse tensor. Moreover it is only support the operations of the first level but not accelerated.

Nd4j will be the first JVM library to support the sparse tensors with their accelerated operations.

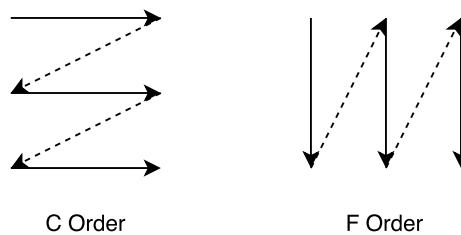
4 Structure of an Multi-dimensional Array

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 (avoid the interruptions of the garbage collector), 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 4.1 shows how a two-dimensional array is stored according to the order.



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

Figure 4.1: Comparison between C-order and F-order

The data is 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

Chapter 4. Structure of an Multi-dimensional Array

case of matrix *A* defined in figure 4.1, 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 is 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, assuming a 64bit-architecture).

// TODO speak about memory copy, host and device memory, etc

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 in memory that belongs to the array or view (explained in section 4.2).
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 array will impact the content of the other arrays. Figure 4.2 illustrates how a view shares its data with its original array.

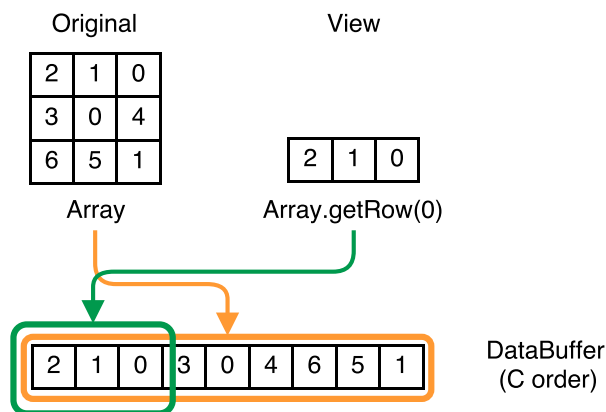


Figure 4.2: View shares memory with the original array

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.

Nd4j emulates the syntax of Numpy[6] and use a similar mechanism to indexes the arrays. Among the different indexes we can note the intervals, the points,...

The indexes provide an efficient mechanism to access Tensors along dimension. A tensor along dimension (TAD) is a view of an actual tensor with a lower rank. The TAD are a very powerful concept to perform fast and optimized operations on tensors.

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.

5 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.

However some methods should be reusable such as traversal operations which operate in an element-wise manner.

But everything has to be defined in the *INDArray* interface. To avoid code duplication, everything that can be shared should be implemented in the higher level of the hierarchy.

Chapter 5. Implementation

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 5.1: 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

The data, encapsulated into DataBuffer object, is stored off-heap. Several reasons drive this design decision.

1. The size of the memory that can be allocated within the JVM is limited to 2 billion. We can not fit matrices into the JVM.
2. We want to use Cuda to accelerate the computation on the GPU. Cuda is not compatible with the JVM environment, the data has to be stored on the native side.
3. Java execution is always sequential, with C++ we can have cheap parallel executions.
4. At C++ level we can have SIMD (Single instruction, multiple data) instructions that allow to execute an instruction in parallel on different blocks of the same data.

Workspaces

Workspaces provides some contiguous memory buffer. No garbage collection mechanism occurs within them, we have to take care of closing the workspace at to avoid memory leaks.

– todo

DataBuffers have a fixed length

Once allocated a *DataBuffer* can not be reallocated. It is not a issue when manipulating dense arrays, we never need to add a new value in a dense array, we only update the existing values. However we need to be able to add new values in the sparse buffers. Each time a zero value become a non-zero value we have to put the value and its indexes in memory.

To reallocate a buffer we need to:

1. Create a new *Pointer* to a bigger memory space and replace the current pointer of the *DataBuffer* .
2. Create a new *Indexer* with the new *Pointer* and replace the current indexer of the *DataBuffer*.
3. Copy the content from the old *Pointer* to the new space.
4. The buffer has two length attributes: *length* which count the number of values in the buffer and *underlyingLength* which reflects the actual allocated size in memory. After the reallocation the *underlyingLength* is increased to the new size, but the *length* only increases at the insertion of a new value.

We over-allocate the buffer in order to limit the amount of reallocation operations.

Another possibility would have been to create a new *DataBuffer* with the content of the old buffer plus the new value. But changing the *DataBuffer* object in a array would have broken the view mechanism since the views would still point to the old *DataBuffer* reference. Moreover this solution doe not allow the over-allocation.

CSR Matrices Implementation

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

Chapter 5. Implementation

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? - coming soon!

Get a Sub-array

1. First, we need to compute the parameters such as the shape, the offsets, ... of the sub-array. We use the *ShapeOffsetResolution* which resolves a combination of indexes and returns the parameters. Despite this class has been made for resolving the indexes of a dense representation, we can reuse it in case of sparse array. The parameters we need are computed independently of the storage format and the internal structure of the array.

For example an interval $[1, 3[$ on a dimension i would result to $offsets[i] = 1$ and $shape[i] = 2$, which is the length of the interval. It does not make any assumption about the underlying storage format.

Here are the parameters we need to compute:

- the shape: an array with two elements containing the new shape of the sub-array.
 - the offsets: an array 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 equal to zeros while having a non-null offset. This situation occurs for example when using only a combination of *all* and *point* indexes In this case we need to compute manually the offsets and override them.

Algorithm 1 Calculate the offsets

procedure OFFSETS(int *offset*)

offsets \leftarrow new int[2]

offsets[0] $\leftarrow \lfloor offset / shape[1] \rfloor$

offsets[1] $\leftarrow offset \% shape[1]$

end procedure

3. With the offsets and the shape we can define the bounds of each dimensions such has $dim_i \in [offsets[i], offsets[i] + shape[i][$

4. Finally we need to loop over every element of the original array. We have to reconstruct the pointers buffers step-by-step by checking if the coordinates of the current element are included in the bounds and update the pointers buffers if needed.

The entire method is shown in appendix A.1.

Limits of this format

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

However we could use this format to represent Tensors Along Dimension (TADs) as explained in section 8.4

COO Tensors

Naive implementation

Based on the description in 2.2.2 the COO encoding needs one data buffer to store all the non-null values and one for the indexes of each value.

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.

The figure 5.2 illustrates the different manner to store the coordinates $[0, 2, 1]$ of a value v_i in a 3-order tensor with four non-zero values.

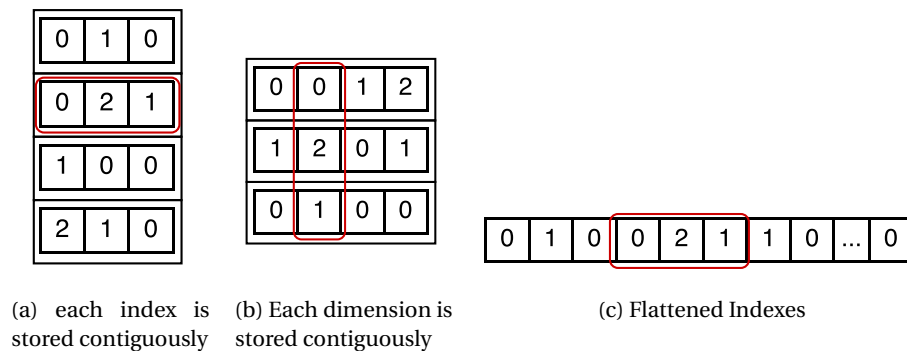


Figure 5.2: Illustration of the different possible datastructure for storing the indexes $[0, 2, 1]$ of a value v

But this implementation makes difficult to be compatible 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 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 not?

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

They can be offset if dimension is partially included in the view. Figure 5.3 shows a matrix and 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 can the indexes be translated between views?

1	2	3
4	5	6
7	8	9

Figure 5.3: 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 if they do not have the same dimensionality?

Reverse coordinates

An important and frequently used method of the API is to get the value of a coordinates. Since the indexes are flatten into one buffer, we need to find it into buffer and get the underlying index.

Since the indexes buffer is lexicographically sorted, a binary search can be applied. But instead of comparing a value, we need to compare a sub-array of the buffer with the target indexes. An additional step is necessary before the comparison: translate the position of the value in the values buffer to get the positions of the indexes in the buffer.

We iterate over the values (NNZ elements) and for each value v_i , we extract the sub-array that contains the coordinates of v_i

The subarray of a value v_i is contained between the following indexes:

```
idx_from = underlyingRank * i;  
idx_end = idx_from + underlyingRank;
```

The `underlyingRank` is the number of coordinates used to index a single value in the original array buffer.

Put a Value

Several steps are needed to put a new value in a COO array:

1. The same value in a view or in the original array can have different coordinates. A translation step is needed between the view indexes and the original array coordinates (the actual coordinates stored in the shared buffer).
2. Verify if the value of this indexes is already non-zero. In this case we can either remove the entry if the new value is zero, or replace the value in the buffer. Nothing else need to be done.
3. If the new value is equal to 0: the indexes and the value are not added in the buffers.
4. If it is a new non-zero value: we need to insert the value and the indexes in the buffers. But first we need to ensure that the buffers have enough spaces for the new entries. If not, we need to reallocate them. The new entries are added at the end of the buffers; the sort are not maintained at insertion to avoid sorting the array too often when it is not needed.

More parameters are needed to define the tensors

Most of the issues cited in section 5.4.1 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 5.3 is the result of the operation :

Chapter 5. Implementation

```
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 equal to 4, which mean that the first row and the first column are not included in the view and the first element in memory that belongs to the view is at position 4.

To be able to identify what values belong to the view and what values do not, we need to define the bounds of each dimension. For this example, we have

$(idx_{row}, idx_{column}) \in view$ if $idx_{row} \in [1, 3[$ and $idx_{column} \in [1, 3[$

We only need to store the lower bound i_{lower} of the interval $idx_{dim} \in [i_{lower}, i_{upper}[$, we can easily compute $i_{upper} = i_{lower} + shape[dim]$. The lower bounds are stored in the *sparseOffset* array.

Point Index

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

Assuming we have a tensor with a shape equals to $(2 \times 3 \times 3)$ as shown in figure 5.4. We want to take a slice along the first dimension, assuming that the first dimension being the pages, the second the rows and the third the columns. We use the following operation:

```
myTensor.get(NDArrayIndex.point(0),  
            NDArrayIndex.all(),  
            NDArrayIndex.all());
```

The result is a view with a shape equals to (3×3) , as shown in grey in figure 5.4

0	2	3
4	0	5
2	8	0

0	3	1
0	0	6
0	1	4

Figure 5.4: A $2 \times 3 \times 3$ tensor with a slice along the first dimension in grey

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 these coordinates and those who actually are in the indexes buffer.

Each value of the tensor has its coordinates in the form (idx_0, idx_1, idx_2) . But in the resulting view the values are defined with only two coordinates: (idx_1, idx_2) , the first dimension is fixed with a value equal to 0.

To translate the view coordinates to the original coordinates which are actually stored into the shared buffer, we need to keep track of the unused dimension and its value. The solution is to

add an additional parameter array that keeps track of the status of each dimension. The array is called *flags* and it can contains either 0, which means *active*, or 1 which means *fixed*.

In this example the flags array would be equal to [1, 0, 0] because the first dimension is fixed at position 0. The value 0 would be keep into the *sparseOffset* array. And the indexes translation would be : $T(idx_1, idx_2) = (sparseOffset[0], idx_1, idx_2)$

Specified Index

Specified index will take specific values of a dimension, depending on the position in the indexes array when calling *myArray.get(indexes...)*. This index takes an array in parameter with the values we want to select.

Assuming we have an 3×5 matrix, as shown in figure 5.5a, on which we call the following operation to select the first, the third and the fourth columns :

```
myMatrix.get(NDArrayIndex.all(),
             new SpecifiedIndex(0, 2, 3));
```

Instead of returning a view of *myMatrix*, it returns a new array containing the data from the original matrix. The data is copied, not shared. The data of the view that would result from this operation is not easily indexable in memory with stride and offsets and it would be difficult to represent it. Therefor the figure 5.5b is an independent 3×3 matrix resulting from the above operation.

0	2	3	1	2
4	0	5	0	0
2	8	0	1	0

(a) A 3×5 matrix with selected columns in grey

0	3	1
4	5	0
2	0	1

(b) A 3×3 matrix created from the column of the matrix in figure 5.5a

Figure 5.5: Example of a specified index behavior on a matrix

New Axis Index

New axis indexes are used to add a new dimension to the array. The new dimension always has a length equal to 1. It can be prepended or inserted in the middle of the dimensions. Since the rank is higher, more coordinates are needed to access a value. However the shared indexes buffers is limited to the original rank. Similarly 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.

Chapter 5. Implementation

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

Sparse Indexes Translation

To translate a view coordinates to the underlying coordinates, we have to take into accounts: the sparse offsets, the hidden dimensions and the flags. With those three parameters we have everything needed for translating.

While iterating over the view dimensions, three situations can happen :

1. The current dimension is hidden: We do nothing
2. The current dimension is fixed: We put the corresponding offset into the result array. Since several fixed dimensions can occur in a row, we need to repeat until we reach an active dimension.
3. The current dimension is active: We sum the view coordinate with the sparse offset.

This method is described in detail in algorithm 2

Example of indexes translation

TODO

Computations of the the Parameters

Computation of the Sparse Offsets

The sparse offsets are computed from the offset. The offset gives 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. For each dimension except the innermost one, we divide the offset by the shape of that dimension. Rounded to the lower integer, this quotient gives us the sparse offset for the dimension. Then we need to remove the number of elements that are in the same dimension but at a lowest value (if the element is in the 4th row, we want to remove all the elements of the previous rows).
2. 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. There is only one operation per dimension, then for a n-order tensor, the complexity of this step is $O(n)$.

Algorithm 2 Translate the indexes from view to underlying context

```

procedure TRANSLATE(int[] viewIndexes)
     $\triangleright$  underlyingRank, hiddenDimensions and sparseOffsets are class fields

    result  $\leftarrow$  new int[underlyingRank]
    idxPhy  $\leftarrow$  0, hidden  $\leftarrow$  0

    for idxVir = 0 to viewIndexes.length do
         $\triangleright$  The current dimension is hidden, it does not appear in the result
        if hidden < hiddenDimension.length & hiddenDimension[hidden] ==
idxVir then
            hidden  $\leftarrow$  hidden + 1
        else
            while idxPhy < underlyingRank & isFixed(idxPhy) do
                result[idxPhy]  $\leftarrow$  sparseOffsets[idxPhy]  $\triangleright$  If the dimension is fixed, the
coordinate takes the value of the offset
                idxPhy  $\leftarrow$  idxPhy + 1
            end while
             $\triangleright$  If the dimension is not fixed, we add the offset to the coordinate
            if idxPhy < underlyingRank & !isFixed(idxPhy) then
                result[idxPhy]  $\leftarrow$  sparseOffsets[idxPhy] + viewIndexes[idxVir]
                idxPhy  $\leftarrow$  idxPhy + 1
            end if
        end if
    end for
end procedure

```

3. We have the sparse offsets given a set of indexes. But we might be computing the sparse offsets of a view which is already offset from an original array. We need to merge the new sparse offsets with those of the current array. The final result is the sum of the existing offset and the freshly computed offset.

During the merge we should particularly be careful with fixed dimensions of the current array because they are absent from the sparse offset resolution explained above. The result does not contain any information about them, we need to add these fixed dimensions with their current offset to final offsets array.

Algorithm 3 Calculate the sparse offsets

```
procedure CREATESPAREOFFSETS(int offset)
  ▷ rank, underlyingRank (the rank of the original array and the number of dimension of
  the indexes buffer), shape and sparseOffsets are class fields of the array instance

  newOffsets ← new int[rank]                                ▷ Compute the new offsets
  for i = 0 to (rank − 2) do
    nbElements ←  $\prod_{j=i+1}^{rank} shape[j]$ 
    newOffsets[i] ←  $\lfloor offset \div nbElements \rfloor$ 
    offset ← offset − newOffsets[i] × nbElements
  end for
  newOffsets[rank − 1] ← offset mod shape[rank − 1]

  finalOffsets ← new int[underlyingRank] ▷ Merge with sparseOffsets of this array
  active ← 0
  for i = 0 to underlyingRank do
    if flags[i] == 0 then      ▷ If the dimension is already inactive in the current array
      finalOffsets[i] ← sparseOffsets[i]
    else
      finalOffsets[i] ← newOffsets[active] + sparseOffsets[i]
      i ← i + 1
    end if
  end for
  return finalOffsets
end procedure
```

Computation of the Flags

The *Flags* array determines which dimension is active and which one is hidden from the point of view of the array. The dimension can only be reduced using a *point* index, which means the flags can be computed during the offset resolution. An interval with a length equals 1 does not reduced the dimensionality of the array. 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 4 Calculate the hidden dimensions

```

procedure CREATEHIDDENDIMENSIONS(int[] newAxis)
    ▷ hiddenDimensions is a class field of the array

    if newAxis is empty or null then return hiddenDimensions
    end if

    if hiddenDimensions is empty then return newAxis
    end if

    size ← newAxis.length + hiddenDimensions.length           ▷ Merge both arrays
    newArray ← new int[size], newArrayIdx ← 0, newDim ← 0

    for (oldDim = 0 to hiddenDimensions.length) do
        while ((oldDim ≥ hiddenDimensions.length || newAxis[newDim] ≤
hiddenDimensions[i]) && newDim < newAxis.length) do
            newArray[i] ← newAxis[newDim] + oldDim
            newArrayIdx ← newArrayIdx + 1
            newDim ← newDim + 1
        end while
        newArray[i] ← hiddenDimensions[oldDim] + newDim
        newArrayIdx ← newArrayIdx + 1
    end for return newArray
end procedure

```

Final Implementation

The final representation is encoded with 5 *DataBuffers*:

1. Values: Stored the value linearly.
2. Indexes: Stored the flattened indexes of each values.
3. Flags: Define which dimensions are active and which are fixed.
4. Sparse Offsets: Define the bound of each dimension.
5. Hidden Dimensions: Keep track of the position in the shape array of the hidden dimension.

The indexes and values are sorted in the lexicographic order in order to make the search by indexes easily via binary search. The sort is ensured before performing any operations.

The flags, the offsets and the hidden dimensions arrays are grouped in one *DataBuffer*, similarly to the *ShapeInformation* buffer.

Example

Figure 5.6a shows a tensor T with a shape $(2 \times 3 \times 3)$ and Figure 5.6b shows the view resulting from the operation below:

```
1 T_view = T.get(NDArrayIndex.newAxis(),
2               NDArrayIndex.point(0),
3               NDArrayIndex.interval(1, 3),
4               NDArrayIndex.interval(1, 3));
```

The first index prepends a new dimension and increases the rank, the second takes the first page of the tensor and reduces the rank. Then the next ones select the sub-array in the first page. The parameters of T_{view} are listed in figure 5.7

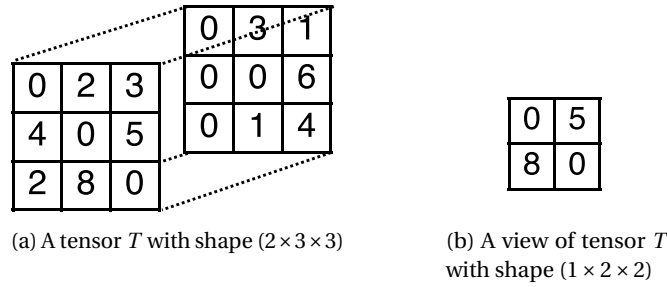


Figure 5.6: A tensor and a view of it

$Shape = [1 \ 2 \ 2] \quad Values = [5 \ 8]$
 $Indexes = [0 \ 1 \ 2 \ 0 \ 2 \ 1] \quad Flags = [1 \ 0 \ 0]$
 $SparseOffsets = [0 \ 1 \ 1] \quad HiddenDimension = [0]$

Figure 5.7: Parameters defining the view T_{view} in 5.6b

Get a Sub-Array

The new parameters make the task easier. Once the indexes resolved and the parameters computed, a new array is created with the current values and indexes buffers and the new parameters.

However if one of the indexes is a *SpecifiedIndex* we need to create a new array and copy the data. In this case the indexes resolution is translate any type of indexes to specified index:

Assuming we have a $3 \times 3 \times 3 \times 3$ 4-order tensor. The indexes array of the following operation:

```
1 myTensor.get(NDArrayIndex.all(),
2             NDArrayIndex.point(1),
3             NDArrayIndex.interval(0, 2),
4             new SpecifiedIndex(0,2));
```

would become : [SpecifiedIndex(0,1,2), SpecifiedIndex(1), SpecifiedIndex(0,1), SpecifiedIndex(0,2)].

Then we iterate over all the included combination of coordinates and add the elements into a new empty array.

6 Operations

Basic Linear Algebra Subprograms (BLAS)

As introduced in section 3.2, BLAS defines the low-level routines for linear algebra operations (for vectors and matrices). Nd4j supports OpenBlas and Intel MKL as the CPU's libraries and Cuda (Cublas) as the GPU's library.

Those libraries are using a dense format only. In case of sparse we need to support two additional libraries: Intel MKL Sparse Blas and CuSparse. To be able to call the MKL sparse methods in the native side, we need to add its presets. The presets are generated by JavaCPP and define the signatures of each routine for which we need to link it into a wrapper method in the java side.

BLAS is organized into three sets of routines called *levels*:

Level 1 performs scalar, vector and vector-vector operations (dot product, norm,...).

Level 2 performs matrix-vector operations (matrix-vector multiplication (gemv)).

Level 3 performs matrix-matrix operations (general matrix multiplication (gemm)).

All the operations are accessible through a BLAS wrapper of *Nd4j*. They can be directly called via static methods or by calling them through an higher abstraction in *INDArray* objects.

This wrapper only wraps the methods of the dense BLAS library, this is why we need a specific wrapper for sparse BLAS library. However we want the different implementations to be transparent for the users, thus we use the dense wrapper as entry point and then redirect the call to the sparse wrapper internally. Thereby the user does not need to know the type of array is manipulating.

A wrapper contains the implementations of each level and Lapack, which is a linear equation system resolver library. The *Levels* interface define the set of operations for each level and then

the implementations gather the the parameter and link to the JavaCPP presets and the native side.

Similarly the sparse architecture has to have the same architecture: One sparse wrapper which contains the sparse implementations for each level.

// TODO explain/show the architecture in nd4j?

Backends

Nd4j can run on the CPU or the GPU.

It supports interchangeable backends. The API is the same, the differences between the backends are transparent for the user. Each backend is parameter-able

– TODO : find doc

In-Place Routines

Some of the operations are performed in-place, that means the input matrix is modified during the operations.

We might have an issue when the number of non-null values increases. We could run out of space in the different buffers because the memory can not be reallocated from the native side and because we can not know how many new values will be added. To avoid this problem we have to ensure having enough space before performing the operations.

The solution is to allocate the maximum size that the routine could use to the *DataBuffers*. At first it may look like inefficient and counterproductive according to the benefits we want to gain with the sparse implementation. However such operations are never executed on the whole array but only one a small sub-part of it.

Example

To illustrate this issue let's take an array *myArray* with a size equals to $(100'000 \times 100'000 \times 100)$. The following operation take a slice of the array *myArray* (which corresponds in a row in this example) and assign 1.0 to each value.

```
1      myArray.get(NDIndexArray.point(0),
2                NDIndexArray.point(0),
3                NDIndexArray.all())
4                .assign(1.0),
```

Despite the huge size of the array, we only add at most 100 new values. If the buffers have not the capacity to get this amount of new values, we need to reallocate them prior the operation.

Level 1 Routines

Sparse BLAS Level 1 routines in Intel MKL library [7] is a set of functions that perform vector operations on sparse vectors stored in a compressed format.

The vector is represented in the compressed form by two arrays, *values* and *indexes*. Figure 6.1 shows how a vector is represented in that format.

$$V_{(N \times 1)} = \begin{bmatrix} a_{k_0} \\ 0 \\ a_{k_1} \\ \dots \\ a_{k_{NNZ}} \end{bmatrix} \rightarrow \begin{array}{l} Value = [a_{k_0} \quad a_{k_1} \quad \dots \quad a_{k_{NNZ}}] \\ Indexes = [k_0 \quad k_1 \quad \dots \quad k_{NNZ}] \end{array}$$

Figure 6.1: A vector v and its compressed form representation

The *values* array can be easily obtained from the CSR and COO format since it corresponds to the *values* buffer in both representations.

In case of CSR vector the *indexes* array corresponds to the column indexes buffer and can also be easily obtained.

However a value in a COO vector is always defined by two coordinates. It is due to a Nd4j singularity where the lowest possible rank for an array is 2, even in case of vector. The *indexes* array has to be extract from the *indexes* buffer depending whether it is a row vector or a column vector.

Once we get the two arrays, the procedure to call the BLAS routines is identical regardless of the underlying sparse format of the array. That mean we only need one implementation. The logic behind the arrays getters are implemented in the format-specific classes.

Level 2 and Level 3 Routines

The level 2 routines perform operations between a sparse matrix and dense vectors whereas level 3 perform between a sparse matrix and dense matrices.

Intel MKL library supports several sparse matrix storage formats such as COO and CSR and each type has its own implementation of the routines.

For now only the general matrix-vector product is implemented (Gemv) for the COO format. All the input parameters of the COO routines can be extracted by a *SparseCOOGemvParameters* object which makes the future implementation of the level 2 routines easier.

Gemv Routine

This routine computes the matrix-vector product of a COO matrix and a dense vector. It can either compute $y = A * x$ or $y = A^T * x$ according to the parameter received in argument.

All the parameters needed to call the BLAS routine are extracted and computed according to the type of sparse format. In case of COO, we need an array of values, an array with the row indexes and an array with the column indexes. Since the indexes are flattened into one buffer, we split have to it into the two arrays.

Libnd4j

Most of the operations are implemented into libNd4j. Basically we used gemv and gemm from blas .. // TODO find doc or ask raver

7 Results

Storing a huge data set

Storing a huge sparse data set (example of gloVe algo dataset?) is now possible with COO while it wasn't with dense. -> compare the theoretical size since dense is not possible to instance

Operations performance

level 1

compare 2-3 operations, which ones?

level 2

Gemv

8 Future

Operations

All the level 1 operations are available but the level 2 and 3 still have to be implemented. The backbone of the implementation is done, we only need to extract the arguments from the array and call the sparse routines.

Regarding the native operations, everything still needs to be done. – TODO how? no idea..

The operations on the tensors work using contractions. a Contraction is a block or a slice along a dimension of the tensor. If the contraction has a rank equal to 2, the matrix operations can be used. The operations on tensors are performed on sub-parts.

Support of the GPU backend

Currently only the backbone of the GPU implementation has been added in Nd4j. We need to add the JavaCPP presets for CuSparse, then similarly to the dense methods, we need to link the API methods to the cuda methods.

– TODO

Make the sparse array API compliant

Most of the methods of the INDArray interface are still to be implemented. We want the sparse representation be compatible and interchangeable with the dense array. It's important that both representations have the same behavior and same features.

Support More Sparse Formats

An additional step would be to support different type of sparse format.

Due to the necessity of having a sparse representation that is not limited to the matrices and vectors, the CSR implementation has been interrupted in favor of the COO format. The CSR representation needs to be completed: Not all the indexes are supported, it misses the BLAS levels 2 and 3, native operations, GPU backend support.

Once the CSR format implemented, it will be easily to extend it to CSC since only the order is different between the two formats.

Then we will need to make the different formats compatible between them, with conversion methods from a format to another. This will allow to get two-dimensional slices of tensors in a different format to benefit of the advantages of each format.

Tensor contraction indexing

Currently the mechanism to reverse a index and get the value (as explained in section 5.4.2) used a binary search. Sadly it does not provide an equal complexity in case of tensor contraction. A tensor contraction is a lower rank view of a tensor. Since the indexes buffer is sorted along the first dimension in a lexicographical order, it makes the task to access values of a contraction which does not contain the first dimension harder since the indexes are not sorted.

We could think to several possibility:

1. Extend the sorting mechanism to sort the indexes buffer according to a ordering of dimension. We could imagine a sort function to which we pass an array with the ordering of the dimensions such as:

```
sortAlongDimension(indexesBuffer, new int[]{2, 0, 1});
```

In this example the coordinates within the buffer would be inverted:

$(i_0, i_1, i_2) \rightarrow (i_2, i_0, i_1)$ and then would be sorted along the new dimension ordering.

2. Use a more complex data-structure to store the indexes such as a skip list or a tree.

The goal is to make the complexity constant regardless of the contraction dimensions.

Optimization

Nd4j is built with the idea to avoid the JVM environnement for storing the data. It is based on the postulate that data is usually huge and does not fit into the memory. However with the new sparse implementation, we can store huge datasets into a reasonable size of memory (as long as it has a high sparsity).

We should study the possibility of storing sparse arrays on-heap with data structure in managed memory such as ArrayList. A further idea would be to decompose the arrays and storing

by block. The blocks with a high sparsity could be stored as on-heap in a sparse format, while the dense blocks would be stored off-heap as dense.

But the question about how on-heap data would interact with the Cuda remains open.

9 Conclusion

?

A Code Snippets

Listing A.1: Extract a sub-array of a CSR matrix

```
1 public INDArray subArray(ShapeOffsetResolution resolution) {
2
3     long[] offsets = resolution.getOffsets();
4     int[] shape = LongUtils.toInts(resolution.getShapes());
5
6
7     List<Integer> accuColumns = new ArrayList<>();
8     List<Integer> accuPointerB = new ArrayList<>();
9     List<Integer> accuPointerE = new ArrayList<>();
10
11     if (shape.length == 2) {
12
13         if (resolution.getOffset() != 0) {
14             offsets[0] = (int) resolution.getOffset() / shape()[1];
15             offsets[1] = (int) resolution.getOffset() % shape()[1];
16         }
17         long firstRow = offsets[0];
18         long lastRow = firstRow + shape[0];
19         long firstElement = offsets[1];
20         long lastElement = firstElement + shape[1];
21
22         int count = 0;
23         int i = 0;
24         for (int rowIdx = 0; rowIdx < lastRow; rowIdx++) {
25
26             boolean isFirstInRow = true;
27             for (int idx = pointerB.getInt(rowIdx); idx < pointerE.getInt(rowIdx)
28                 ; idx++) {
29
30                 int colIdx = columnsPointers.getInt(count);
31
32                 // add the element in the subarray if it belongs to the view
33                 if (colIdx >= firstElement && colIdx < lastElement && rowIdx >=
34                     firstRow && rowIdx < lastRow) {
35
36                     // add the new column pointer for this element
37                     accuColumns.add((int) (colIdx - firstElement));
38
39                     if (isFirstInRow) {
```

Appendix A. Code Snippets

```
38         // Add the index of the first element of the row in the
39         // pointer array
40         accuPointerB.add(idx);
41         accuPointerE.add(idx + 1);
42         isFirstInRow = false;
43     } else {
44         // update the last element pointer array
45         accuPointerE.set((int) (rowIdx - firstRow), idx + 1);
46     }
47     count++;
48 }
49
50 // If the row doesn't contain any element and is included in the
51 // selected rows
52 if (isFirstInRow && rowIdx >= firstRow && rowIdx < lastRow) {
53     int lastIdx = i == 0 ? 0 : accuPointerE.get(i - 1);
54     accuPointerB.add(lastIdx);
55     accuPointerE.add(lastIdx);
56 }
57 if (rowIdx >= firstRow && rowIdx <= lastRow) {
58     i++;
59 }
60
61 int[] newColumns = Ints.toArray(accuColumns);
62 int[] newPointerB = Ints.toArray(accuPointerB);
63 int[] newPointerE = Ints.toArray(accuPointerE);
64
65 INDArray subarray = Nd4j.createSparseCSR(values, newColumns,
66     newPointerB, newPointerE, shape);
67
68 return subarray;
69
70 } else {
71     throw new UnsupportedOperationException();
72 }
```

A Sparse Offset - An execution example

This appendix presents an execution of the algorithm 3 presented in section 5.4.6.

Let's start with an tensor *myTensor* with a shape equal to [2,4,4,5] on which we are calling the following operation:

```
myTensor.get(NDArrayIndex.all(),
NDArrayIndex.point(0),
NDArrayIndex.point(3),
NDArrayIndex.all());
```

Assuming that *myTensor* is a view of a 5-order tensor and has a the following parameters:

```
sparseOffsets = [1, 1, 0, 0, 0]
flags = [1, 0, 0, 0, 0]
```

Assuming we name the dimension as [book, page, row, column], we are taking all the columns of the last row of the first pages of each book.

The indexes resolution returns the following parameters:

```
offsets = [0, 0]
shape = [2, 5]
offset = 15
```

First step is to iterate over the dimension:

Iteration: i=0

Number of element in one book: $numElement = 4 \times 4 \times 5 = 60$

Book offset = $\lfloor offset / numElement \rfloor = \lfloor 15 / 60 \rfloor = 0$

Then we update the offset: $offset = offset - 0 * 60$

Iteration: i=1

Number of element in one page = $numElement = 4 \times 5 = 20$

Page offset = $\lfloor offset / numElement \rfloor = \lfloor 15 / 20 \rfloor = 0$

Then we update the offset: $offset = 15 - 0 * 20$

Iteration: i=2

Number of element in one row = $numElement = 5$

Appendix A. Sparse Offset - An execution example

Page offset = $\lfloor offset / numElement \rfloor = \lfloor 15 / 5 \rfloor = 3$

Then we update the offset: $offset = 15 - 3 * 5 = 0$

Finally we reach the last dimension:

Column offset = $offset \bmod numElement = 0 \bmod 5 = 0$

We get an temporary offsets array equal to $newOffsets = [0, 0, 3, 0]$. Now we need to merge with the existing offsets of *myTensor*.

Its first dimension is fixed, so we copy its sparse offset :

$finalOffset[0] = myTensor.sparseOffset[0] = 1$

Its second dimension is active and there is already an non-zero offset. The offset is equal to

$finalOffset[1] = newOffset[0] + myTensor.sparseOffset[1] = 0 + 1 = 1$

Its second dimension is active. The offset is equal to

$finalOffset[2] = newOffset[1] + myTensor.sparseOffset[2] = 0 + 0 = 0$

Its third dimension is active. The offset is equal to

$finalOffset[3] = newOffset[2] + myTensor.sparseOffset[3] = 3 + 0 = 3$

Its third dimension is active. The offset is equal to

$finalOffset[4] = newOffset[3] + myTensor.sparseOffset[4] = 0 + 0 = 0$

We finally get the final sparseOffset : $[1, 1, 0, 3, 0]$

A Indexes Translation - An execution example

This appendix presents an execution of the algorithm 2 presented in section ??

Bibliography

- [1] “Zipf’s law.” <https://en.wikipedia.org/wiki/Zipf>
- [2] “Netflix grand prize.” <http://www.netflixprize.com>. [Online; accessed 8-August-2017].
- [3] Y. Koren, “1 the bellkor solution to the netflix grand prize,” 2009.
- [4] “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].
- [5] “Breeze - a numerical processing library for scala.” <https://github.com/scalanlp/breeze>. [Online; accessed 14-August-2017].
- [6] “Numpy.” <http://www.numpy.org/>. [Online; accessed 15-August-2017].
- [7] “Intel MKL sparse level 1 routines.” <https://software.intel.com/en-us/mkl-developer-reference-c-sparse-blas-level-1-routines>. [Online; accessed 13-August-2017].