



Intel® Data Analytics Acceleration Library 2017 Update 2

Developer Guide

Intel® DAAL

Legal Information

Contents

Legal Information.....	7
Getting Technical Support.....	9
What's New.....	11
Introducing the Intel® Data Analytics Acceleration Library.....	13
 Chapter 1: Data Management	
Numeric Tables.....	20
Generic Interfaces.....	23
Essential Interfaces for Algorithms.....	27
Tensors.....	28
Generic Interfaces.....	28
Essential Interfaces for Algorithms.....	29
Data Sources.....	29
Data Dictionaries.....	30
Data Serialization and Deserialization.....	31
Data Compression.....	32
Data Model.....	35
 Chapter 2: Algorithms	
Algorithm Input.....	37
Algorithm Output.....	37
Algorithm Parameters.....	38
Computation Modes.....	38
Analysis	47
Moments of Low Order	47
Details.....	47
Batch Processing.....	48
Online Processing.....	50
Distributed Processing.....	53
Performance Considerations.....	56
Quantile.....	56
Details	56
Batch Processing.....	57
Correlation and Variance-Covariance Matrices	58
Details.....	58
Batch Processing.....	58
Online Processing.....	60
Distributed Processing.....	63
Performance Considerations.....	66
Cosine Distance Matrix.....	67
Details.....	67
Batch Processing.....	67
Performance Considerations.....	68
Correlation Distance Matrix.....	68
Details.....	68

Batch Processing.....	69
Performance Considerations.....	69
K-Means Clustering.....	70
Details.....	70
Initialization.....	71
Computation.....	89
Performance Considerations.....	97
Principal Component Analysis.....	97
Details.....	97
Batch Processing.....	97
Online Processing.....	99
Distributed Processing.....	102
Performance Considerations.....	106
Cholesky Decomposition	107
Details.....	107
Batch Processing.....	107
Performance Considerations.....	108
Singular Value Decomposition.....	108
Details.....	109
Batch and Online Processing.....	109
Distributed Processing.....	110
Performance Considerations.....	117
Association Rules	118
Details.....	118
Batch Processing.....	118
Performance Considerations.....	121
QR Decomposition	121
QR Decomposition without Pivoting.....	122
Pivoted QR Decomposition	130
Performance Considerations.....	132
Expectation-Maximization.....	133
Details.....	133
Initialization.....	134
Computation.....	136
Performance Considerations.....	138
Multivariate Outlier Detection.....	139
Details.....	139
Batch Processing.....	139
Performance Considerations.....	142
Univariate Outlier Detection.....	142
Details.....	142
Batch Processing.....	143
Performance Considerations.....	144
Kernel Functions	144
Linear Kernel	144
Radial Basis Function Kernel	146
Quality Metrics.....	148
Quality Metrics for Binary Classification Algorithms.....	148
Quality Metrics for Multi-class Classification Algorithms.....	151
Quality Metrics for Linear Regression.....	153

Working with User-defined Quality Metrics.....	159
Sorting.....	159
Details.....	159
Batch Processing.....	159
Normalization.....	160
Z-score.....	160
Min-max.....	162
Performance Considerations.....	163
Math Functions.....	164
Logistic.....	164
Softmax.....	165
Hyperbolic Tangent.....	166
Absolute Value (abs).....	168
Rectifier Linear Unit (ReLU).....	169
Smooth Rectifier Linear Unit (SmoothReLU).....	170
Performance Considerations.....	171
Optimization Solvers.....	172
Objective Function.....	172
Iterative Solver.....	177
Training and Prediction	190
Regression	190
Usage Model: Training and Prediction	191
Linear Regression	194
Ridge Regression	194
Computation.....	195
Classification	204
Usage Model: Training and Prediction	204
Naïve Bayes Classifier.....	208
Boosting.....	217
Support Vector Machine Classifier.....	226
Multi-class Classifier.....	229
k-Nearest Neighbors (kNN) Classifier.....	231
Recommendation Systems.....	233
Usage Model: Training and Prediction	233
Implicit Alternating Least Squares.....	236
Neural Networks.....	251
Usage Model: Training and Prediction	252
Batch Processing.....	254
Distributed Processing.....	256
Initializers	263
Layers.....	266
Chapter 3: Services	
Extracting Version Information	389
Handling Errors	390
Managing Memory	390
Managing the Computational Environment	390
Bibliography.....	393

Legal Information

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.

This document contains information on products, services and/or processes in development. All information provided here is subject to change without notice. Contact your Intel representative to obtain the latest forecast, schedule, specifications and roadmaps.

The products and services described may contain defects or errors which may cause deviations from published specifications. Current characterized errata are available on request.

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.

Cilk, Intel, the Intel logo, Intel Atom, Intel Core, Intel Inside, Intel NetBurst, Intel SpeedStep, Intel vPro, Intel Xeon Phi, Intel XScale, Itanium, MMX, Pentium, Thunderbolt, Ultrabook, VTune and Xeon are trademarks of Intel Corporation in the U.S. and/or other countries.

*Other names and brands may be claimed as the property of others.

Java is a registered trademark of Oracle and/or its affiliates.

Third Party Content

Intel® Data Analytics Acceleration Library (Intel® DAAL) includes content from several 3rd party sources that is governed by the licenses referenced below:

- bzip2 compression and decompression algorithm.

This program, "bzip2", the associated library "libbzip2", and all documentation, are copyright © 1996-2007 Julian R Seward. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- The origin of this software must not be misrepresented; you must not claim that you wrote the original software. If you use this software in a product, an acknowledgment in the product documentation would be appreciated but is not required.
- Altered source versions must be plainly marked as such, and must not be misrepresented as being the original software.
- The name of the author may not be used to endorse or promote products derived from this software without specific prior written permission.
- Zlib compression and decompression algorithm.

Copyright © 1995-2013 Jean-loup Gailly and Mark Adler.

This software is provided 'as-is', without any express or implied warranty. In no event will the authors be held liable for any damages arising from the use of this software.

Permission is granted to anyone to use this software for any purpose, including commercial applications, and to alter it and redistribute it freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must not claim that you wrote the original software. If you use this software in a product, an acknowledgment in the product documentation would be appreciated but is not required.
 2. Altered source versions must be plainly marked as such, and must not be misrepresented as being the original software.
 3. This notice may not be removed or altered from any source distribution.
- Safe C Library.

MIT License

Copyright (c)

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Read more about this license at <http://www.opensource.org/licenses/mit-license.php>

Copyright© 2017, Intel Corporation. All rights reserved.

Optimization Notice
Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice. Notice revision #20110804

Getting Technical Support

Intel® Data Analytics Acceleration Library (Intel® DAAL) provides a product web site that offers timely and comprehensive product information, including product features, white papers, and technical articles. For the latest information, check: <http://www.intel.com/software/products/support>.

Intel also provides a support web site that contains a rich repository of self help information, including getting started tips, known product issues, product errata, license information, user forums, and more (visit <http://www.intel.com/software/products/>).

Registering your product entitles you to one year of technical support and product updates through Intel® Premier Support. Intel Premier Support is an interactive issue management and communication web site providing these services:

- Submit issues and review their status.
- Download product updates anytime of the day.

To register your product, contact Intel, or seek product support, please visit <http://www.intel.com/software/products/support>.

What's New

This Developer Guide documents Intel® Data Analytics Acceleration Library (Intel® DAAL) 2017 Update 2.

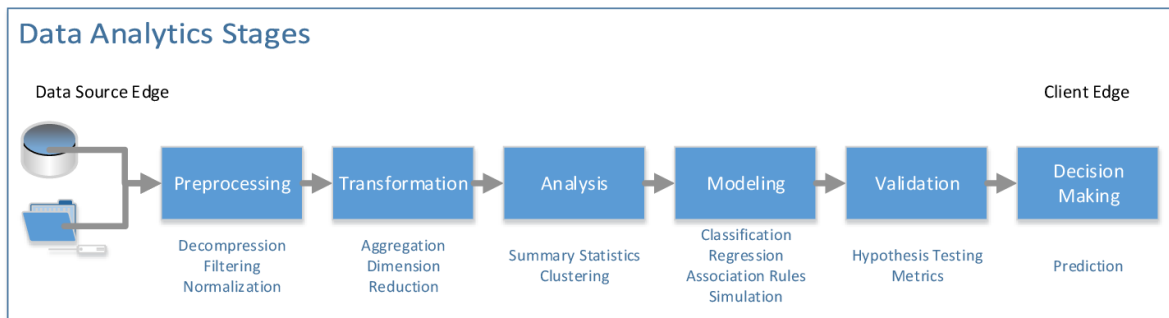
The document has been updated to reflect new functionality and enhancements to the product and to improve usability and clarity:

- Introduction to neural networks has been updated (for more details, see [Neural Networks](#)).
- The description of neural network usage model has been updated (for more details, see [Neural Networks > Usage Model: Training and Prediction](#)).
- Truncated Gaussian initializer of neural network layers has been added (for more details, see [Truncated Gaussian Initializer](#)).
- Parameters of the 2D spatial pyramid pooling backward layer have been updated (for more details, see [2D Spatial Pyramid Pooling Backward Layer > Batch Processing](#)).
- Transposed 2D convolution layer has been added (for more details, see [2D Transposed Convolution Forward Layer](#) and [2D Transposed Convolution Backward Layer](#)).
- Documentation for 2D convolution layers has been updated (for more details, see [2D Convolution Forward Layer](#) and [2D Convolution Backward Layer](#)).
- Parameters of univariate outlier detection have been updated (for more details, see [Univariate Outlier Detection > Batch Processing](#)).
- Parameters of multivariate outlier detection have been updated (for more details, see [Multivariate Outlier Detection > Batch Processing](#)).
- For the stochastic gradient descent (SGD) iterative solver, the momentum computation method has been added (for more details, see [Stochastic Gradient Descent Algorithm](#)).
- The algorithm of the loss softmax cross-entropy layer has been updated (for more details, see [Loss Softmax Cross-entropy Forward Layer](#) and [Loss Softmax Cross-entropy Backward Layer](#)).
- Loss logistic cross-entropy layer has been added (for more details, see [Loss Logistic Cross-entropy Forward Layer](#) and [Loss Logistic Cross-entropy Backward Layer](#)).
- The description of the K-Means clustering algorithm has been updated (for more details, see [K-Means Clustering > Details](#)).
- Documentation for distributed processing of the K-Means clustering algorithm initialization has been updated (for more details, see [K-Means Clustering > Initialization > Distributed Processing](#)).
- Layer input has been updated for the backward split layer (for more details, see [Split Backward Layer > Batch Processing](#)).
- Layer output has been updated for the forward split and backward concat layers (for more details, see [Split Forward Layer > Batch Processing](#) and [Concat Backward Layer > Batch Processing](#)).
- Objective function with precomputed characteristics has been added (for more details, see [Objective Function with Precomputed Characteristics Algorithm](#)).
- Reshape layer has been added (for more details, see [Reshape Forward Layer](#) and [Reshape Backward Layer](#)).
- Common parameters of neural network layers have been updated. (for more details, see [Common Parameters](#)).

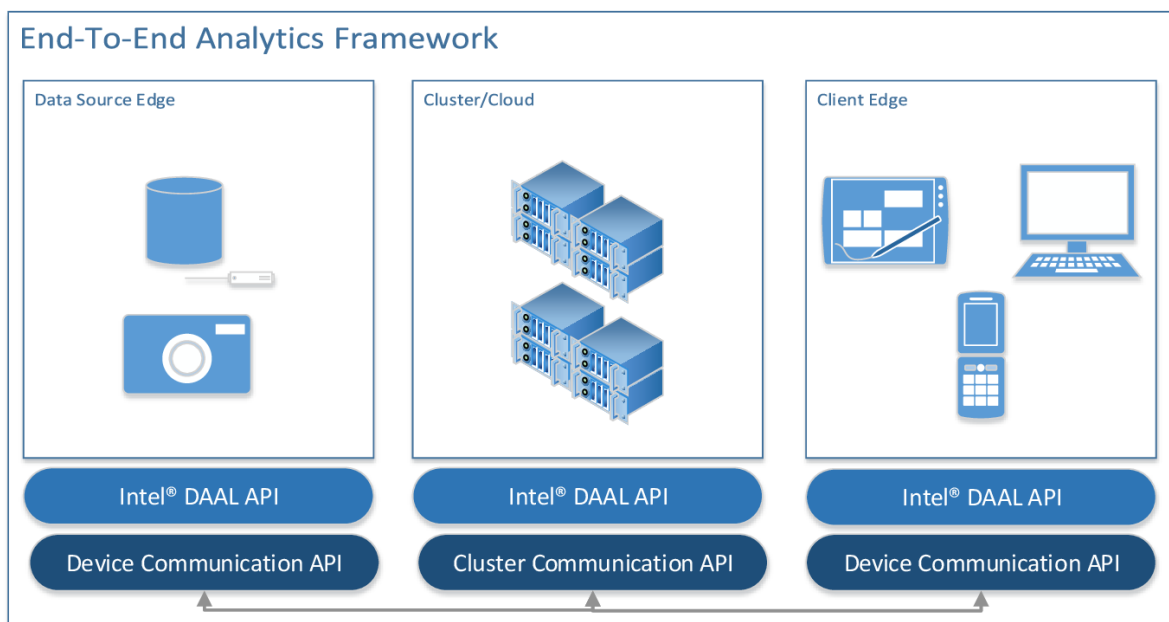
Additionally, updates have been made to correct inaccuracies in the document.

Introducing the Intel® Data Analytics Acceleration Library

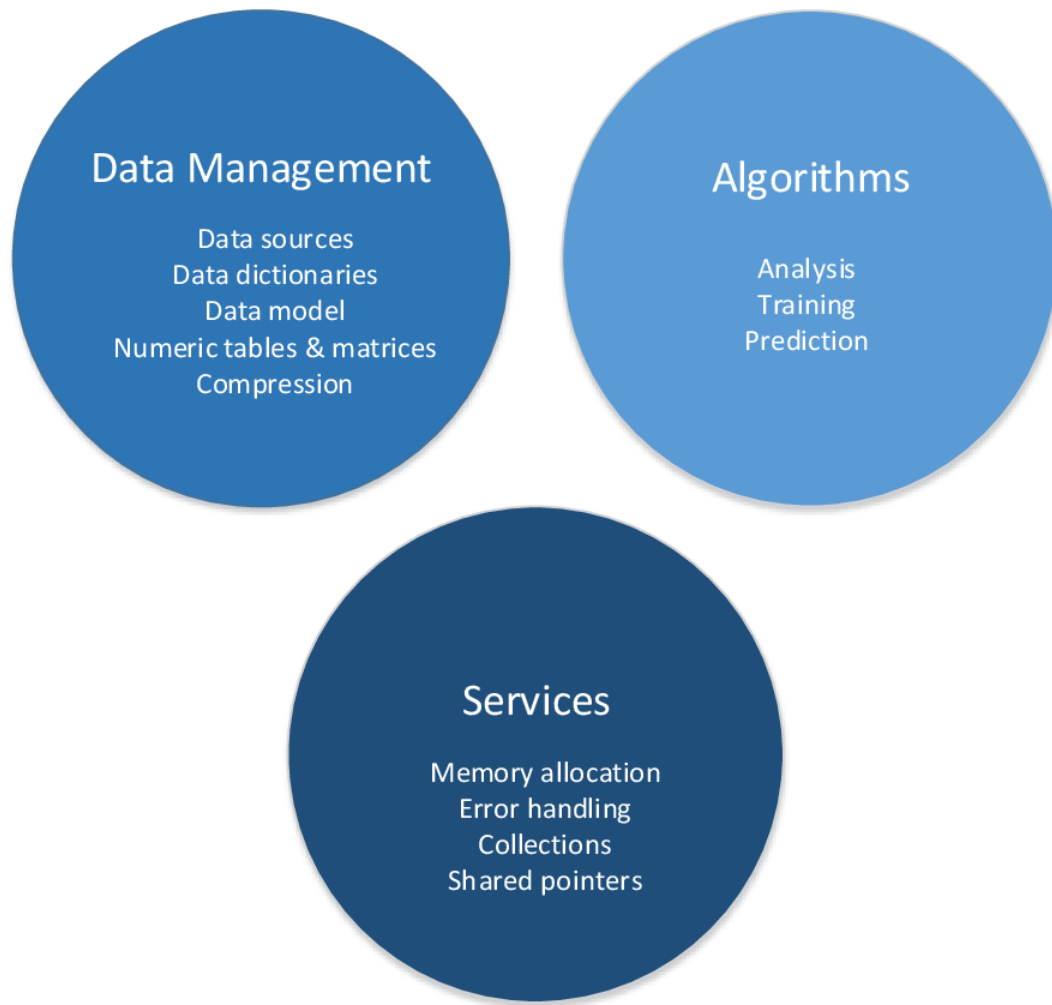
The Intel® Data Analytics Acceleration Library (Intel® DAAL) is the library of Intel® Architecture optimized building blocks covering all data analytics stages: data acquisition from a data source, preprocessing, transformation, data mining, modeling, validation, and decision making. To achieve best performance on a range of Intel® processors, Intel DAAL uses optimized algorithms from the Intel® Math Kernel Library and Intel® Integrated Performance Primitives.



Intel DAAL supports the concept of the end-to-end analytics when some of data analytics stages are performed on the edge devices (close to where the data is generated and where it is finally consumed). Specifically, Intel DAAL Application Programming Interfaces (APIs) are agnostic about a particular cross-device communication technology and therefore can be used within different end-to-end analytics frameworks.



Intel DAAL consists of the following major components: Data Management, Algorithms, and Services:



The Data Management component includes classes and utilities for data acquisition, initial preprocessing and normalization, for data conversion into numeric formats done by one of supported Data Sources, and for model representation. The `NumericTable` class and its derivative classes within the Data Management component are intended for in-memory numeric data manipulation. The `Model` class mimics the actual data and represents it in a compact way so that you can use the library when the actual data is missing, incomplete, noisy, or unavailable. These are the key interfaces for processing a data set with algorithms. The `DataSourceDictionary` and `NumericTableDictionary` classes provide generic methods for dictionary manipulation, such as accessing a particular data feature, setting and retrieving the number of features, and adding a new feature.

The Algorithms component consists of classes that implement algorithms for data analysis (data mining), and data modeling (training and prediction). These algorithms include matrix decompositions, clustering, classification, and regression algorithms, as well as association rules discovery.

Algorithms support the following computation modes:

- Batch processing
- Online processing
- Distributed processing

In the *batch processing* mode, the algorithm works with the entire data set to produce the final result. A more complex scenario occurs when the entire data set is not available at the moment or the data set does not fit into the device memory.

In the *online processing* mode, the algorithm processes a data set in blocks streamed into device memory by doing incrementally updating partial results, which are finalized upon processing of the last data block.

In the *distributed processing* mode, the algorithm operates on a data set distributed across several devices (compute nodes). The algorithm produces partial results on each node, which are finally merged into the final result on the master node.

Distributed algorithms in Intel DAAL are abstracted from underlying cross-device communication technology, which enables use of the library in a variety of multi-device computing and data transfer scenarios. They include but are not limited to MPI* based cluster environments, Hadoop*/Spark* based cluster environments, low-level data exchange protocols, and more.

You can find typical use cases for MPI*, Hadoop*, and Spark* at [https://software.intel.com/en-us/product-code-samples?field_software_product_tid\[\]=79775](https://software.intel.com/en-us/product-code-samples?field_software_product_tid[]=79775).

Depending on the usage, algorithms operate both on actual data (data set) and data models. Analysis algorithms typically operate on data sets. Training algorithms typically operate on a data set to train the appropriate data model. Prediction algorithms typically work with the trained data model and with a working data set.

Models produced by training algorithms are associated with interfaces for generation of metrics that characterize the models. These metrics can be used during creation of a model to understand its quality, or in a production mode to track agreement between actual data and the model that represents it.

The Services component includes classes and utilities used across Data Management and Algorithms components. These classes enable memory allocation, error handling, and implementation of collections and shared pointers. A collection enables storing different type of objects in a unified manner. In Intel DAAL, collections are used for handling the input and output of algorithms and for error handling. Intel DAAL implements shared pointers to enable memory handling needed for memory management operations such as deallocation.

Classes implemented in Data Management, Algorithms, and Services components cover most important usage scenarios and allow seamless implementation of complex data analytics workflows through direct API calls. At the same time the library is an object-oriented framework that enables you to customize the API by redefining particular classes and methods of the library.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Data Management

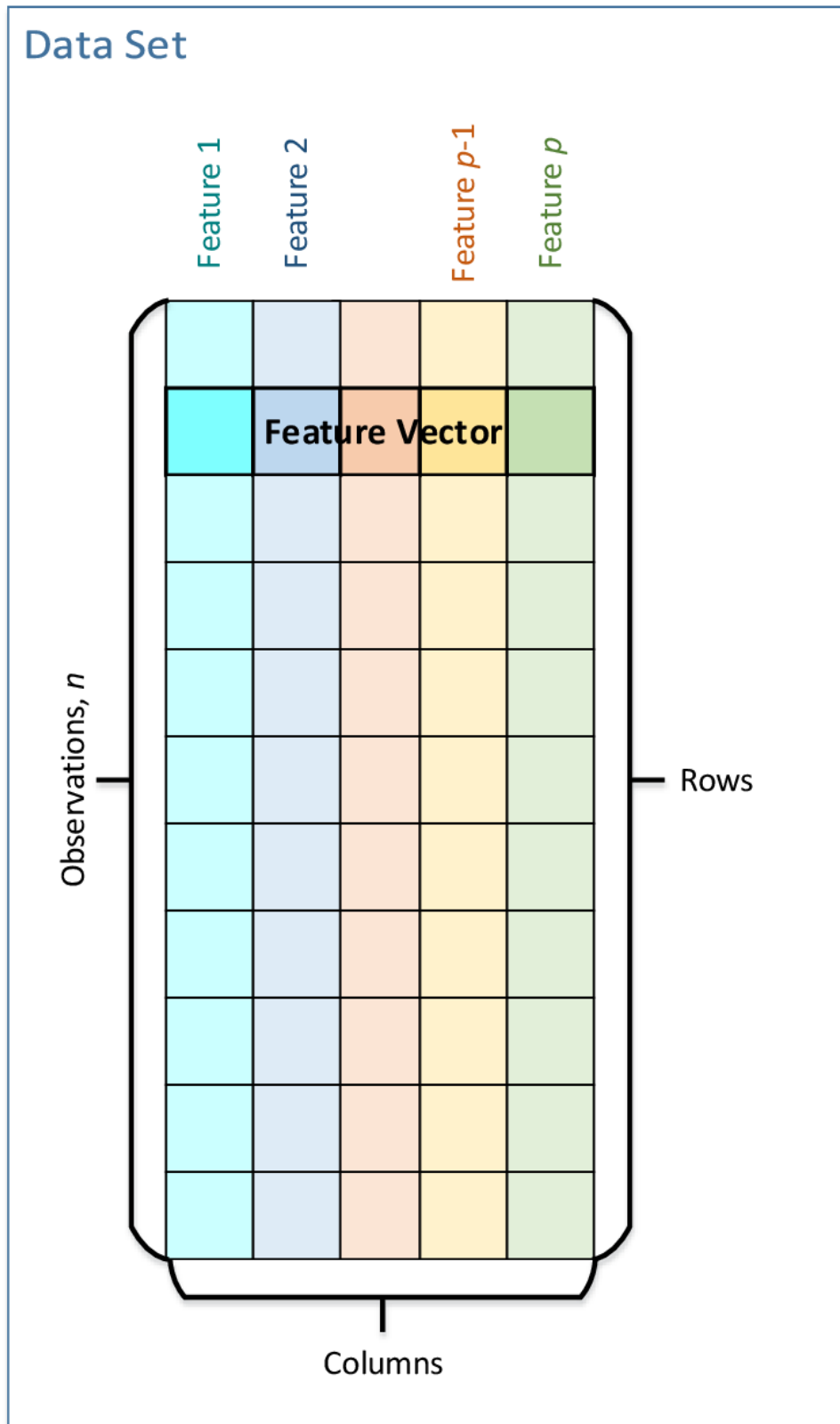
Effective data management is among key constituents of the performance of a data analytics application. For Intel® Data Analytics Acceleration Library (Intel® DAAL), effective data management requires effectively performing the following operations:

1. Raw data acquisition, filtering, and normalization with data source interfaces.
2. Conversion of the data to a numeric representation for numeric tables.
3. Data streaming from a numeric table to an algorithm.

Depending on the usage model, you may also want to apply compression and decompression to the data you operate on. You can either use compression and decompression embedded into data source interfaces or apply data serialization and deserialization interfaces.

Intel DAAL provides a set of customizable interfaces to operate on your out-of-memory and in-memory data in different usage scenarios, which include batch processing, online processing, and distributed processing, as well as more complex scenarios, such as a combination of online and distributed processing.

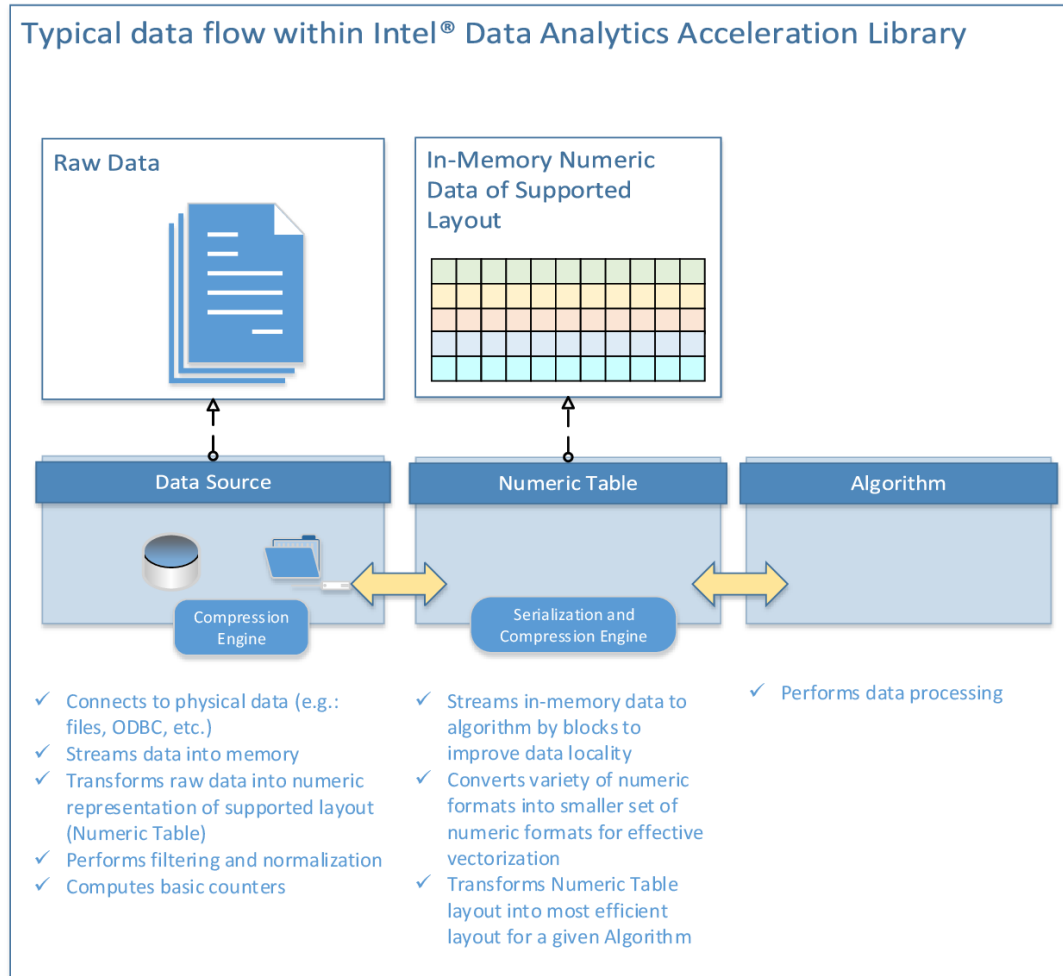
One of key concepts of Data Management in Intel DAAL is a data set. A *data set* is a collection of data of a defined structure that characterizes an object being analyzed and modeled. Specifically, the object is characterized by a set of attributes (Features), which form a Feature Vector of dimension p . Multiple feature vectors form a set of Observations of size n . Intel DAAL defines a tabular view of a data set where table rows represent observations and columns represent features.



An observation corresponds to a particular measurement of an observed object, and therefore when measurements are done, at distinct moments in time, the set of observations characterizes how the object evolves in time.

It is not a rare situation when only a subset of features can be measured at a given moment. In this case, the non-measured features in the feature vector become blank, or missing. Special statistical techniques enable recovery (emulation) of missing values.

It is not a rare situation either when a given measurement introduces an outlier or when an observed object can produce an outlier by nature. An *outlier* is the value that has an abnormal deviation from typical values. Special statistical techniques enable detection of outliers and recovery of the abnormal data.



You normally start working with Intel DAAL by selecting an appropriate data source, which provides an interface for your raw data set. Intel DAAL data sources support categorical, ordinal, and continuous features. It means that data sources can automatically transform non-numeric categorical and ordinary data into a numeric representation. When the structure of your raw data is more complex or when the default transformation mechanism does not fit your needs, you may customize the data source by implementing a custom derivative class.

Because a data source is typically associated with out-of-memory data, such as files, databases, and so on, streaming out-of-memory data into memory and back is among major functions of a data source. However you can also use a data source to implement an in-memory non-numeric data transformation into a numeric form.

A numeric table is a key interface to operate on numeric in-memory data. Intel DAAL supports several important cases of a numeric data layout: homogeneous tables, arrays of structures, and structures of arrays, as well as Compressed Sparse Row (CSR) encoding for sparse data.

Intel DAAL algorithms operate with in-memory numeric data accessed through Numeric table interfaces.

Numeric Tables

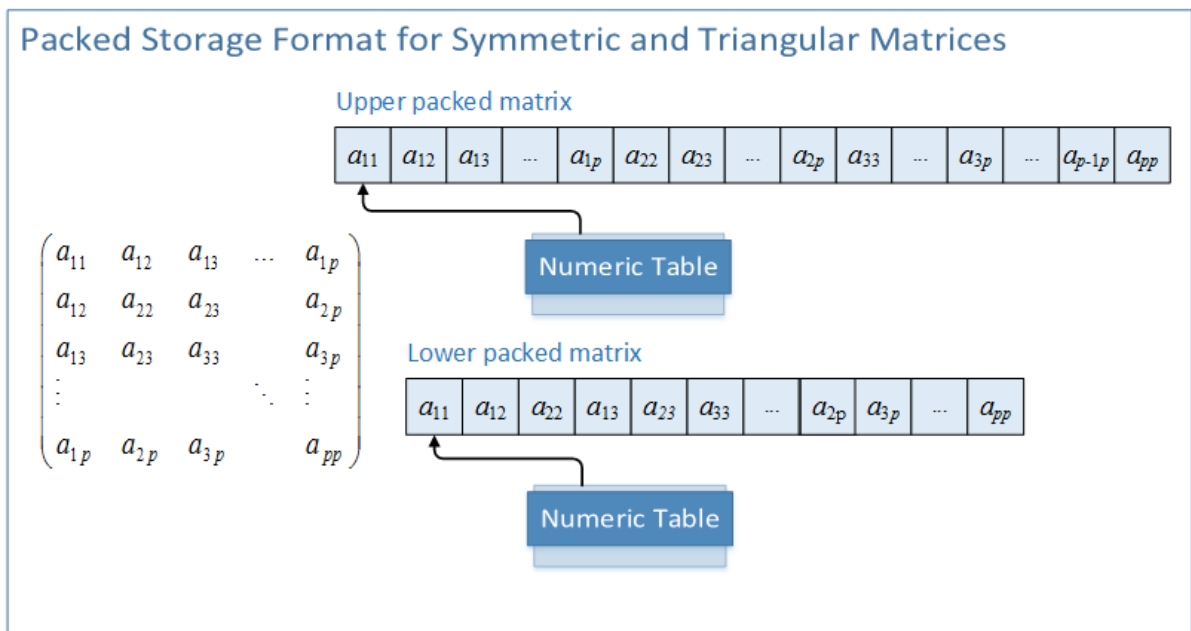
Numeric tables are a fundamental component of in-memory numeric data processing. Intel DAAL supports heterogeneous and homogeneous numeric tables for dense and sparse data, as follows:

- Heterogeneous, Array Of Structures (AOS)
- Heterogeneous, Structure Of Arrays (SOA)
- Homogeneous, dense
- Homogeneous matrix, dense
- Homogeneous symmetric matrix, packed
- Homogeneous triangular matrix, packed
- Homogeneous, sparse CSR

Use homogeneous numeric tables, that is, objects of the `HomogenNumericTable` class, and matrices, that is, objects of the `Matrix`, `PackedTriangularMatrix`, and `PackedSymmetricMatrix` classes, when all the features are of the same basic data type. Values of the features are laid out in memory as one contiguous block in the row-major order, that is, Observation 1, Observation 2, and so on. In Intel DAAL, `Matrix` is a homogeneous numeric table most suitable for matrix algebra operations.

For triangular and symmetric matrices with reduced memory footprint, special classes are available: `PackedTriangularMatrix` and `PackedSymmetricMatrix`. Use the `DataLayout` enumeration to choose between representations of triangular and symmetric matrices:

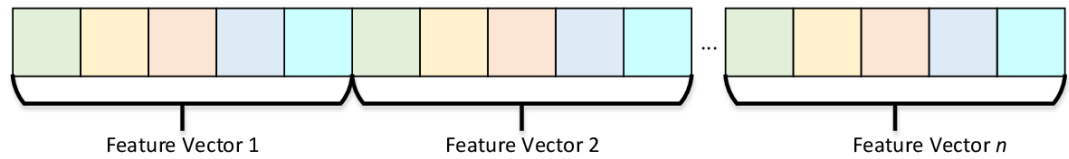
- Lower packed: `lowerPackedSymetricMatrix` or `lowerPackedTriangularMatrix`
- Upper packed: `upperPackedTriangularMatrix` or `upperPackedSymetricMatrix`



Heterogeneous numeric tables enable you to deal with data structures that are of different data types by nature. Intel DAAL provides two ways to represent non-homogeneous numeric tables: AOS and SOA.

AOS Numeric Table provides access to observations (feature vectors) that are laid out in a contiguous memory block:

Memory Layout: Array-Of-Structures (AOS)



Examples

Refer to the following examples in your Intel DAAL directory:

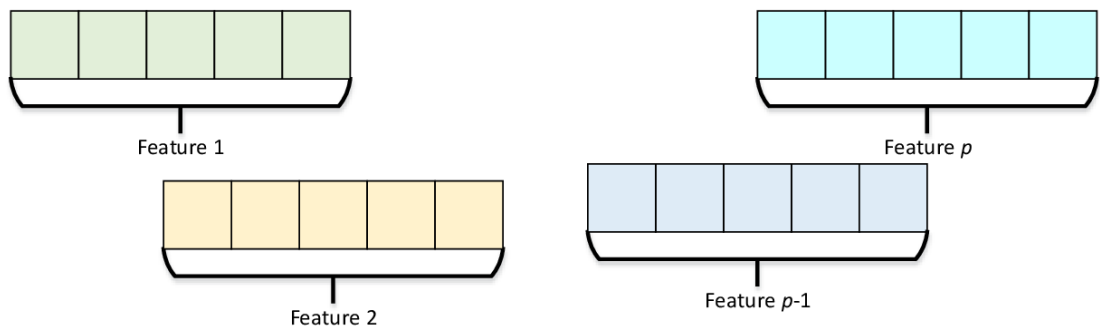
C++: `./examples/cpp/source/datasource/datastructures_aos.cpp`

Java*: `./examples/java/source/com/intel/daal/examples/datasource/DataStructuresAOS.java`

Python*: `./examples/python/source/datasource/datastructures_aos.py`

SOA Numeric Table provides access to data sets where observations for each feature are laid out contiguously in memory:

Memory Layout: Structure-Of-Arrays (SOA)



Examples

Refer to the following examples in your Intel DAAL directory:

C++: `./examples/cpp/source/datasource/datastructures_soa.cpp`

Java*: `./examples/java/source/com/intel/daal/examples/datasource/DataStructuresSOA.java`

Python*: `./examples/python/source/datasource/datastructures_soa.py`

The optimal data layout for homogeneous and heterogeneous numeric tables highly depends on a particular algorithm. You can find algorithm-specific guidance in the Performance Considerations section for the appropriate algorithm.

Intel DAAL offers the `CSRNumericTable` class for a special version of a homogeneous numeric table that encodes sparse data, that is, the data with a significant number of zero elements. The library uses the Condensed Sparse Row (CSR) format for encoding:

Condensed Sparse Row (CSR) 0-Based Encoding

		<i>j</i>								
		0	1	2	3	4	5	6		
<i>M</i> = 2	0	2.0		6.4			1.7		<i>i</i>	<i>k</i>
	1				3.1					
	2									
	3		2.2			2.1				
	4						3.8	5.5		

		0	1	2	3	4	5	6	7
values		2.0	6.4	1.7	3.1	2.2	2.1	3.8	5.5
columns		0	2	5	3	1	4	5	6
rowIndex		0	3	4	4	6	8		

Condensed Sparse Row (CSR) 1-Based Encoding

		<i>j</i>								
		1	2	3	4	5	6	7		
<i>M</i> = 3	1	2.0		6.4			1.7		<i>i</i>	<i>k</i>
	2				3.1					
	3									
	4		2.2			2.1				
	5						3.8	5.5		

		1	2	3	4	5	6	7	8
values		2.0	6.4	1.7	3.1	2.2	2.1	3.8	5.5
columns		1	3	6	4	2	5	6	7
rowIndex		1	4	5	5	7	9		

Three arrays describe the sparse matrix *M* as follows:

- The array *values* contains non-zero elements of the matrix row-by-row.
- The *j*-th element of the array *columns* encodes the column index in the matrix *M* for the *j*-th element of the array *values*.
- The *i*-th element of the array *rowIndex* encodes the index in the array *values* corresponding to the first non-zero element in rows indexed *i* or greater. The last element in the array *rowIndex* encodes the number of non-zero elements in the matrix *M*.

You can specify 1-based (*oneBased*) CSR encoding through the *indexing* parameter of type *CSRIndexing* in the constructor of *CSRNumericTable*.

Examples

Refer to the following examples in your Intel DAAL directory:

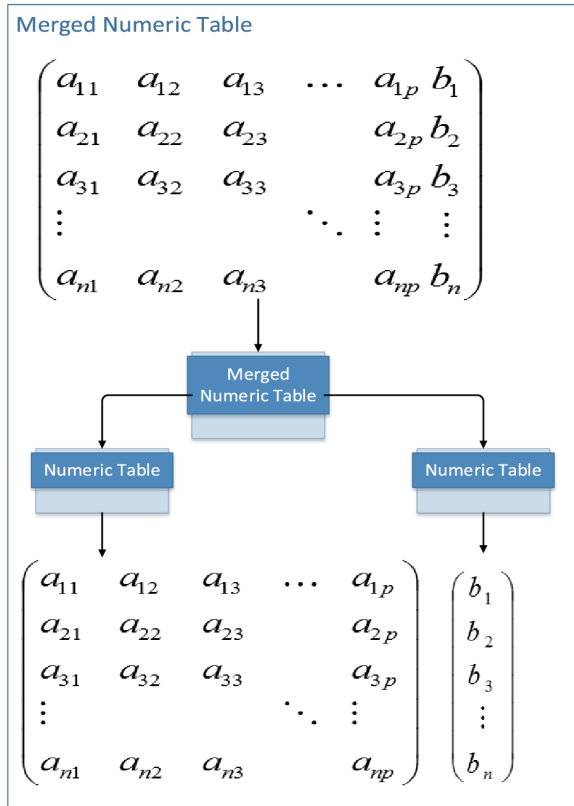
C++: `./examples/cpp/source/datasource/datastructures_csr.cpp`

Java*: `./examples/java/source/com/intel/daal/examples/datasource/DataStructuresCSR.java`

Python*: `./examples/python/source/datasource/datastructures_csr.py`

Intel DAAL offers the *MergedNumericTable* class for tables that provide access to data sets comprising several logical components, such as a set of feature vectors and corresponding labels. This type of tables enables you to read those data components from one data source. This special type of numeric tables can hold several numeric tables of any type but *CSRNumericTable*. In a merged numeric table, arrays are joined

by columns and therefore can have different numbers of columns. In the case of different numbers of rows in input matrices, the number of rows in a merged table equals $\min(r_1, r_2, \dots, r_m)$, where r_i is the number of rows in the i -th matrix, $i = 1, 2, 3, \dots, m$.



Examples

Refer to the following examples in your Intel DAAL directory:

C++: `./examples/cpp/source/datasource/datastructures_merged.cpp`

Java*: `./examples/java/source/com/intel/daal/examples/datasource/DataStructuresMerged.java`

Python*: `./examples/python/source/datasource/datastructures_merged.py`

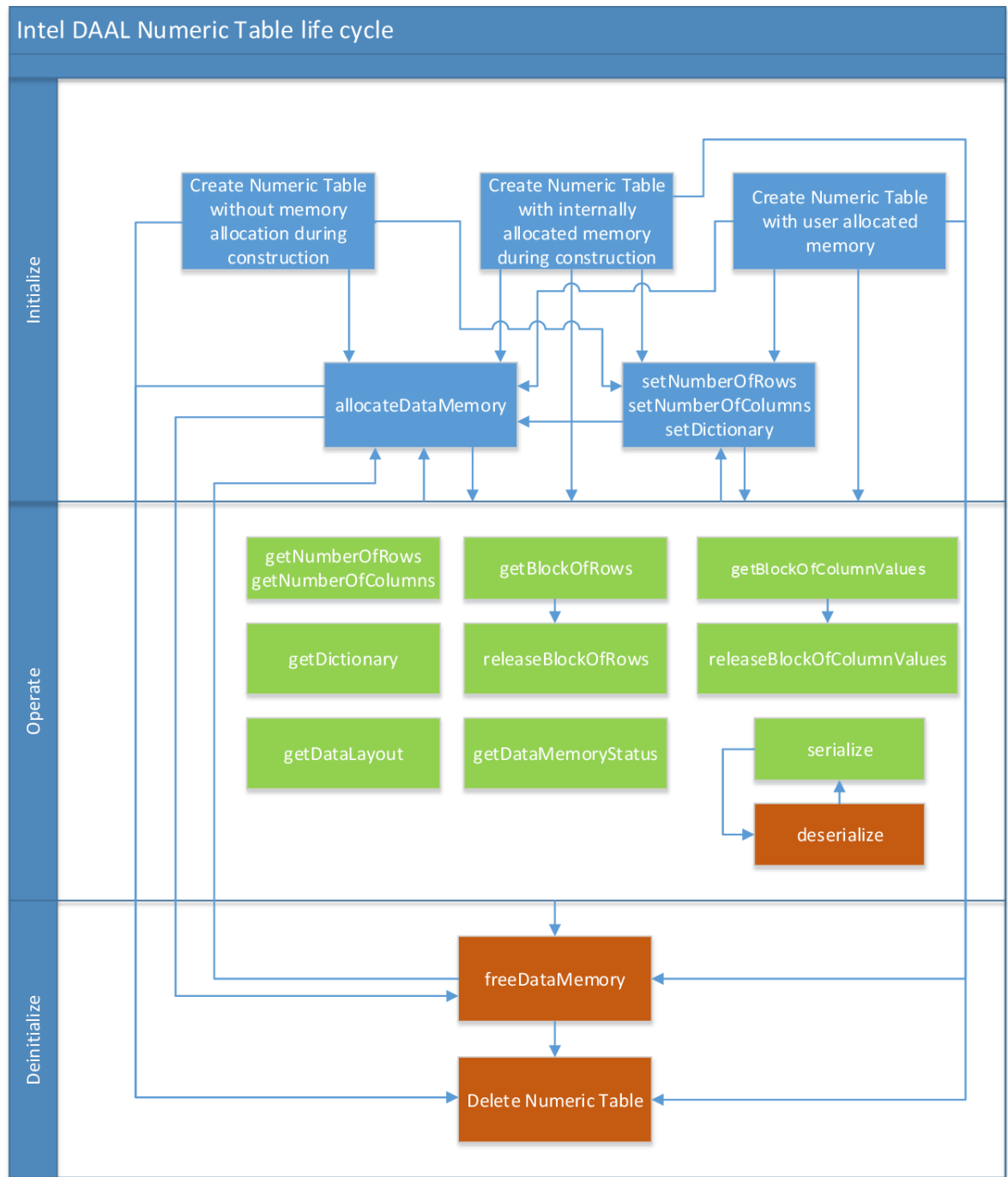
Generic Interfaces

Numeric tables provide interfaces for data management, such as memory allocation and deallocation, and respective memory access methods, dictionary management, and table size management.

The life cycle of a numeric table consists of the following major steps:

1. Initialize
2. Operate
3. Deinitialize

The following diagram shows possible flows and transitions between the states of the numeric table for each step. The description of these steps applies to different types of numeric tables supported in the library, such as CSR, with appropriate changes in the method names and respective arguments.



Initialize

Intel DAAL provides several constructors for numeric tables to cover a variety of table initialization scenarios. The constructors require the numbers of rows and columns for the table or a dictionary. If you do not have the dictionary or sizes of the numeric table at the time of construction, you can use the constructor with default values and sizes. The following scenarios are available for use of constructors:

- If the table size is unknown at the time of object construction, you can construct an empty table and change the size and allocate the memory later. You can also use the constructor to specify the sizes, but provide a pointer to the memory later:

```
HomogenNumericTable<double> table(nColumns, nRows,
                                   NumericTable::notAllocate);
double data[nColumns * nRows];
table.setArray(data);
```

- If the table size is known but the data is not yet in memory, Intel DAAL can allocate the memory automatically at the time of object construction and even initialize the memory, that is, allocate the matrix with zero elements:

```
HomogenNumericTable<double> table(nColumns, nRows,
                                   NumericTable::doAllocate, 0.0);
```

- If the data is already available in memory by the time of object construction, you can provide a pointer to this data through the appropriate constructor:

```
double data[nColumns * nRows];
HomogenNumericTable<double> table(data, nColumns, nRows);
```

To allocate or reallocate the memory after construction of the numeric table, use service methods:

- `setNumberOfRows()`, `setNumberOfColumns()`, and `setDictionary()`.

These methods modify the table sizes and metadata. Use them when the table size or metadata changes dynamically. In this scenario you are responsible for the correspondence between the table size and the data associated with this table. See [Data Dictionaries](#) for additional details on metadata.

- `allocateDataMemory()`.

This method allocates memory needed to store the data associated with a given numeric table. Note that this method first calls `freeDataMemory()`.

- `setArray()`.

This method registers a pointer to the data represented as a contiguous memory block, such as homogeneous and AOS data. For the numeric tables such as SOA, which require the data to be laid out as a set of contiguous memory blocks by features, the method operates with a pointer to the memory block for a given feature. You are responsible for the correspondence between the table size and the data associated with this table.

Operate

After initialization or re-initialization of a numeric table, you can use the following methods for the numeric table to access the data:

- `getBlockOfRows()` and `releaseBlockOfRows()`.

The `getBlockOfRows()` method provides access to a data block stored in the numeric table. The `rwflag` argument specifies read or write access. Provide the object of the `BlockDescriptor` type to the method to interface the requested block of rows. This object, the block descriptor, represents the data in the contiguous raw-major layout with the number of rows specified in the method and number of columns specified in the numeric table.

In Intel DAAL you can represent the data in the block descriptor with the data type different from the data type of the numeric table. For example: you can represent a homogeneous data with the `float` data type, while the block descriptor represents the requested data in `double`. You can specify the required data type during the construction of the block descriptor object. Make sure to call the `releaseBlockOfRows()` method after a call to `getBlockOfRows()`. The data types of the numeric table and block descriptor, as well as the `rwflag` argument of the `getBlockOfRows()` method, define the behavior of `releaseBlockOfRows()`:

- If `rwflag` is set to `writeOnly` or `readWrite`, `releaseBlockOfRows()` writes the data from the block descriptor back to the numeric table.

- If the numeric table and block descriptor use different data types or memory layouts, `releaseBlockOfRows()` deallocates the allocated buffers regardless of the value of `rwflag`.

```
HomogenNumericTable<double> table(data, nColumns, nRows);
BlockDescriptor<float> block;
table.getBlockOfRows(firstReadRow, nReadRows, readOnly, block);
float *array = block.getBlockPtr();
for (size_t row = 0; row < nReadRows; row++)
{
    for (size_t col = 0; col < nColumns; col++)
    {
        std::cout << array[row * nColumns + col] << "    ";
    }
    std::cout << std::endl;
}
table.releaseBlockOfRows(block);
```

- `getBlockOfColumnValues()` and `releaseBlockOfColumnValues()`.

These methods provide access to values in the specific column of a numeric table, similarly to `getBlockOfRows()` and `releaseBlockOfRows()`.

- `getNumberOfRows()` and `getNumberOfColumns()`.

Call these methods to determine the number of rows and columns, respectively, associated with a given numeric table.

- `getDictionary()` and `resetDictionary()`, as well as `getFeatureType()` and `getNumberOfCategories()`.

These methods provide access to the data dictionary associated with a given numeric table. See [Data Dictionaries](#) for more details.

- `getDataMemoryStatus()`.

Call this method to determine whether the memory is allocated by the `allocateDataMemory()` method, a user provided a pointer to the allocated data, or no data is currently associated with the numeric table. Additionally, the `getArray()` method is complimentary to `setArray()` and provides access to the data associated with a given table of a given layout.

- `serialize` and `deserialize()`.

The `serialize()` method enables you to serialize the numeric table. Call the deserialization method `deserialize()` after each call to `serialize()`, but before a call to other data access methods.

Deinitialize

After you complete your work with a data resource, the appropriate memory is deallocated implicitly in the destructor of the numeric table. For more control over allocation/deallocation of the memory, you can deallocate the memory explicitly by the `freeDataMemory()` method. This method deallocates the memory internally allocated by the library or sets the pointer to the user allocated memory to zero if the numeric table stores this pointer. The destructor of the numeric table calls the `freeDataMemory()` method.

NOTE

Python*: When creating a numpy array from a numeric table, make sure that a reference to the numeric table exists as long as a reference to the derived numpy array is being used.

Examples

Refer to the following examples in your Intel DAAL directory:

C++:

- `./examples/cpp/source/datasource/datastructures_merged.cpp`
- `./examples/cpp/source/datasource/datastructures_homogen.cpp`

Java*:

- `./examples/python/source/datasource/DataStructuresMerged.java`
- `./examples/python/source/datasource/DataStructuresHomogen.java`

Python*:

- `./examples/python/source/datasource/datastructures_merged.py`
- `./examples/python/source/datasource/datastructures_homogen.py`

Essential Interfaces for Algorithms

In addition to [Generic Interfaces](#), more methods enable interfacing numeric tables with algorithms.

The `getDataLayout` method provides information about the data layout:

Data Layout	Description
<code>soa</code>	Structure-Of-Arrays (SOA). Values of individual data features are stored in contiguous memory blocks.
<code>aos</code>	Array-Of-Structures (AOS). Feature vectors are stored in contiguous memory block.
<code>csr_Array</code>	Condensed-Sparse-Row (CSR).
<code>lowerPackedSymetricMatrix</code>	Lower packed symmetric matrix
<code>lowerPackedTriangularMatrix</code>	Lower packed triangular matrix
<code>upperPackedSymetricMatrix</code>	Upper packed symmetric matrix
<code>upperPackedTriangularMatrix</code>	Upper packed triangular matrix
<code>unknown</code>	No information about data layout or unsupported layout.

Rather than access the entire in-memory data set, it is often more efficient to process it by blocks. The key methods that Intel DAAL algorithms use for per-block data access are `getBlockOfRows()` and `getBlockOfColumnValues()`. The `getBlockOfRows()` method accesses a block of feature vectors, while the `getBlockOfColumnValues()` method accesses a block of values for a given feature. A particular algorithm uses `getBlockOfRows()`, `getBlockOfColumnValues()`, or both methods to access the data. The efficiency of data access highly depends on the data layout and on whether the data type of the feature is natively supported by the algorithm without type conversions. Refer to the Performance Considerations section in the description of a particular algorithm for a discussion of the optimal data layout and natively supported data types.

When the data layout fits the per-block data access pattern and the algorithm requests the data type that corresponds to the actual data type, the `getBlockOfRows()` and `getBlockOfColumnValues()` methods avoid data copying and type conversion. However, when the layout does not fit the data access pattern or when type conversion is required, both methods automatically re-pack and convert data as required.

When dealing with custom or unsupported data layouts, you must implement `NumericTableIface`, `DenseNumericTableIface` interfaces, and optionally `CSRNumericTableIface` or `PackedNumericTableIface` interfaces.

Some algorithms, such as [Moments of Low Order](#), compute basic statistics (minimums, maximums, and so on). The other algorithms, such as [Correlation and Variance-Covariance Matrices](#) or [Principal Component Analysis](#), require some basic statistics on input. To avoid duplicated computation of basic statistics, Intel DAAL provides methods to store and retrieve basic statistics associated with a given numeric table: `basicStatistics.set()` and `basicStatistics.get()`. The following basic statistics are computed for each numeric table:

- `minimum` - minimum

- `maximum` - `maximum`
- `sum` - `sum`
- `sumSquares` - `sum of squares`

Special Interfaces for the `HomogenNumericTable` and `Matrix` Classes

Use the `assign` method to initialize elements of a dense homogeneous numeric table with a certain value, that is, to set all elements of the matrix to zero.

Use the `operator []` method to access rows of a homogeneous dense numeric table.

Special Interfaces for the `PackedTriangularMatrix` and `PackedSymmetricMatrix` Classes

While you can use generic `getArray()` and `setArray()` methods to access the data in a packed format, in algorithms that have specific implementations for a packed data layout, you can use more specific `getPackedValues()` and `releasePackedValues()` methods.

Special Interfaces for the `CSRNumericTable` Class

To access three CSR arrays (*values*, *columns*, and *rowIndex*), use `getArrays()` and `setArrays()` methods instead of generic `getArray()` and `setArray()` methods. For details of the arrays, see [CSR data layout](#). Similarly, in algorithms that have specific implementations for the CSR data layout, you can use more specific `getBlockOfCSRValues()` and `releaseBlockOfCSRValues()` methods.

Special Interfaces for the `MergedNumericTable` Class

To add a new array to the object of the `MergedNumericTable` class, use the `addNumericTable()` method.

Tensors

Tensors represent in-memory numeric multidimensional data. Intel DAAL supports homogeneous tensors for dense data.

Use the `HomogenTensor` class when all values are of the same basic data type and are laid out in memory as one contiguous block with dimensions in the row-major order. The data in objects of the `HomogenTensor` class is described with the memory block, number of dimensions, and size of each dimension.

Examples

Refer to the following examples in your Intel DAAL directory:

C++: `./examples/cpp/source/datasource/datastructures_homogentensor.cpp`

Java*: `./examples/java/source/com/intel/daal/examples/datasource/DataStructuresHomogenTensor.java`

Python*: `./examples/python/source/datasource/datastructures_homogentensor.py`

Generic Interfaces

Tensors provide interfaces for data management, such as memory allocation and deallocation, and respective methods for memory access and dimension management.

Use the `getNumberOfDimensions()` and `getDimensions()` methods to determine the number of dimensions and size of all dimensions, respectively. Use the `setDimensions()` method to set or change the number of dimensions and their sizes. Use these methods when a tensor size changes dynamically. In this scenario you are responsible for the correspondence between the tensor size and the data associated with this tensor.

Use the `shuffleDimensions()` method to change logical indexing of tensor dimensions without changing the data layout. This method changes the dimensions order, visible to algorithms.

Use the `allocateDataMemory()` and `freeDataMemory()` methods to allocate and deallocate memory needed to store the data associated with a given tensor. The dimensions and data associated with them determine the amount of memory required to store the data. The `getDataMemoryStatus()` method enables you to determine whether the memory is allocated by the `allocateDataMemory()` method, a user provided a pointer to the allocated data, or no data is currently associated with the tensor.

Use the `assign` method to initialize elements of a tensor with a certain value.

Essential Interfaces for Algorithms

In addition to [Generic Interfaces](#), more methods enable interfacing tensors with algorithms.

The key methods that Intel DAAL algorithms use for per-block data access are `getSubtensor()`/`releaseSubtensor()`. These methods access a part of the tensor assuming the row-major order. The efficiency of data access highly depends on the data layout and on whether the data type is natively supported by the algorithm without type conversions. Refer to the Performance Considerations section in the description of a particular algorithm for a discussion of the data layout and natively supported data types.

Use the `getSize()` method to get the number of elements stored in a `Tensor` object.

Special Interfaces for the `HomogenTensor` Class

Use the `getArray()` method to get direct access to the memory block associated with the `HomogenTensor` object.

Data Sources

Data sources define interfaces for access and management of data in raw format and out-of-memory data. A data source is closely coupled with the data dictionary that describes the structure of the data associated with the data source. To create the associated data dictionary, you can do one of the following:

- While constructing a data source object, specify whether it should automatically create and initialize the associated data dictionary.
- Call the `createDictionaryFromContext()` method.

The `getDictionary()` method returns the dictionary associated with the data source.

Data sources stream and transform raw out-of-memory data into numeric in-memory data accessible through numeric table interfaces. A data source is associated with the corresponding numeric table. To allocate the associated numeric table, you can do one of the following:

- While constructing a data source object, specify whether it should automatically allocate the numeric table.
- Call the `allocateNumericTable()` method.

The `getNumericTable()` method returns the numeric table associated with the data source.

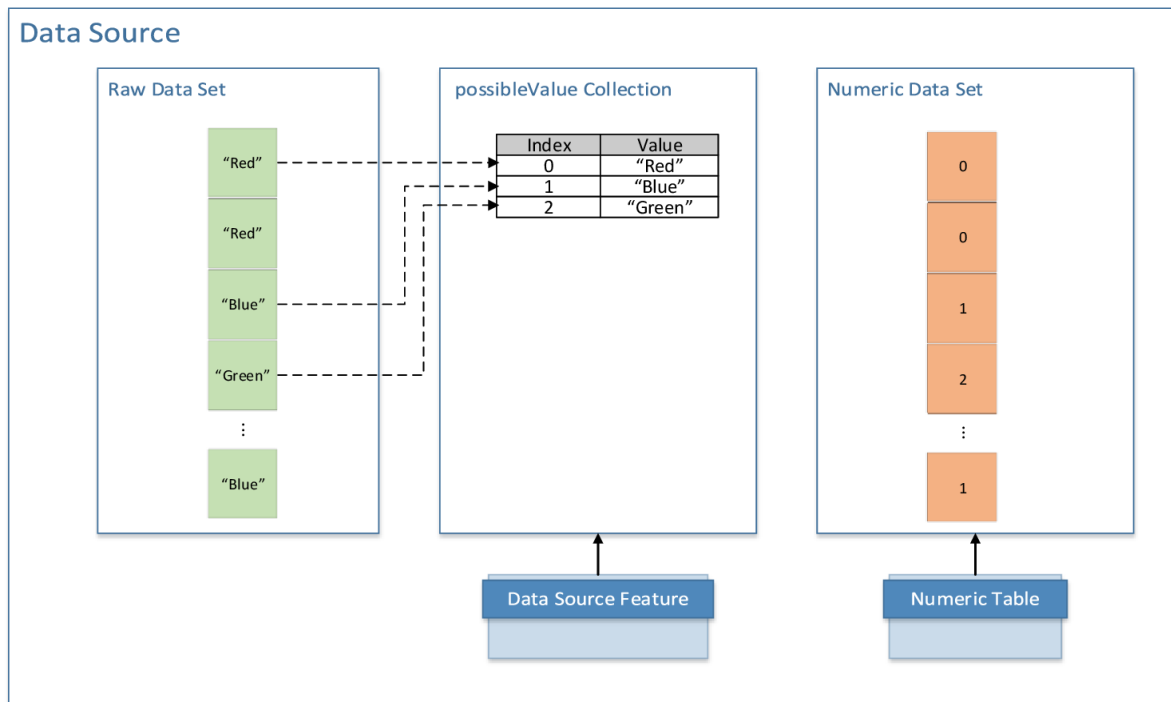
To retrieve the number of columns (features) in a raw data set, use the `getNumberOfColumns()` method. To retrieve the number of rows (observations) available in a raw data set, use the `getNumberOfAvailableRows()` method. The `getStatus()` method returns the current status of the data source:

- `readyForLoad` - the data is available for the load operation.
- `waitingForData` - the data source is waiting for new data to arrive later; designated for data sources that deal with asynchronous data streaming, that is, the data arriving in blocks at different points in time.
- `endOfData` - all the data is already loaded.

Because the entire out-of-memory data set may fail to fit into memory, as well as for performance reasons, Intel DAAL implements data loading in blocks. Use the `loadDataBlock()` method to load the next block of data into the numeric table. This method enables you to load a data block into an internally allocated numeric table or into the provided numeric table. In both cases, you can specify the number of rows or not. The method also recalculates basic statistics associated with this numeric table.

Intel DAAL maintains the list of possible values associated with categorical features to convert them into a numeric form. In this list, a new index is assigned to each new value found in the raw data set. You can get the list of possible values from the `possibleValues` collection associated with the corresponding feature in the data source. In the case you have several data sets with same data structure and you want to use continuous indexing, do the following:

1. Retrieve the data dictionary from the last data source using the `getDictionary()` method.
2. Assign this dictionary to the next data source using the `setDictionary()` method.
3. Repeat these steps for each next data source.



Intel DAAL implements classes for some popular types of data sources. Each of these classes takes a feature manager class as the class template parameter. The feature manager parses, filters, and normalizes the data and converts it into a numeric format. The following are the data sources and the corresponding feature manager classes:

- Text file (`FileDataSource` class), to be used with the `CSVFeatureManager` class
- ODBC (`ODBCDataSource` class), to be used with the `MySQLFeatureManager` class
- In-memory text (`StringDataSource` class), to be used with the `CSVFeatureManager` class

Data Dictionaries

A data dictionary is the metadata that describes features of a data set. The `NumericTableFeature` and `DataSourceFeature` structures describe a particular feature within a dictionary of the associated numeric table and data source respectively. These structures specify:

- Whether the feature is continuous, categorical, or ordinal

- Underlying data types (double, integer, and so on) used to represent feature values

The `DataSourceFeature` structure also specifies:

- Possible values for a categorical feature
- The feature name

The `DataSourceDictionary` class is a data dictionary that describes raw data associated with the corresponding data source. The `NumericTableDictionary` class is a data dictionary that describes in-memory numeric data associated with the corresponding numeric table. Both classes provide generic methods for dictionary manipulation, such as accessing a particular data feature, setting and retrieving the number of features, and adding a new feature. Respective `DataSource` and `NumericTable` classes have generic dictionary manipulation methods, such as `getDictionary()` and `setDictionary()`.

To create a dictionary from the data source context, you can do one of the following:

- Set the `doDictionaryFromContext` flag in the `DataSource` constructor.
- Call to the `createDictionaryFromContext()` method.

Examples

Refer to the following examples in your Intel DAAL directory:

C++:

- `./examples/cpp/source/datasource/datastructures_aos.cpp`
- `./examples/cpp/source/datasource/datastructures_soa.cpp`
- `./examples/cpp/source/datasource/datastructures_homogen.cpp`

Java*:

- `./examples/java/source/com/intel/daal/examples/datasource/DataStructuresAOS.java`
- `./examples/java/source/com/intel/daal/examples/datasource/DataStructuresSOA.java`
- `./examples/java/source/com/intel/daal/examples/datasource/DataStructuresHomogen.java`

Python*:

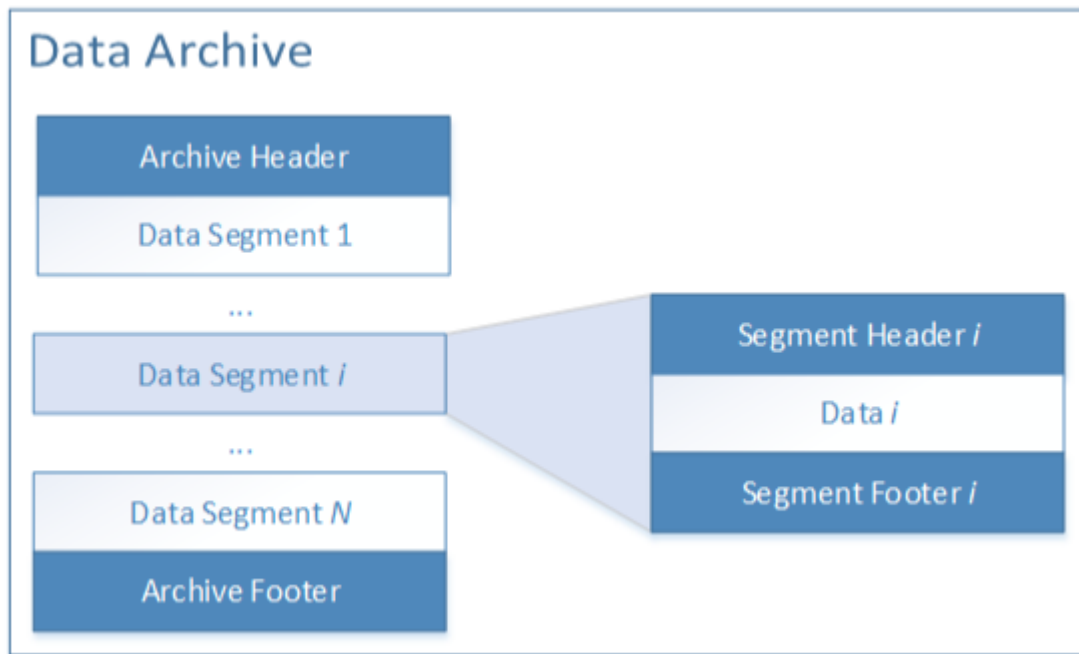
- `./examples/python/source/datasource/datastructures_aos.py`
- `./examples/python/source/datasource/datastructures_soa.py`
- `./examples/python/source/datasource/datastructures_homogen.py`

Data Serialization and Deserialization

Intel DAAL provides interfaces for serialization and deserialization of data objects, which are an essential technique for data exchange between devices and for implementing data recovery mechanisms on a device failure.

The `InputDataArchive` class provides interfaces for creation of a serialized object archive. The `OutputDataArchive` class provides interfaces for deserialization of an object from the archive. To reduce network traffic, memory, or persistent storage footprint, you can compress data objects during serialization and decompress them back during deserialization. To this end, provide `Compressor` and `Decompressor` objects as arguments for `InputDataArchive` and `OutputDataArchive` constructors respectively. For details of compression and decompression, see [Data Compression](#).

A general structure of an archive is as follows:



Headers and footers contain information required to reconstruct the archived object.

All serializable objects, such as numeric tables, a data dictionary, and models, have serialization and deserialization methods. These methods take input archive and output archive, respectively, as method parameters.

Examples

Refer to the following examples in your Intel DAAL directory:

C++: `./examples/cpp/source/serialization/serialization.cpp`

Java: `./examples/java/source/com/intel/daal/examples/serialization/SerializationExample.java`

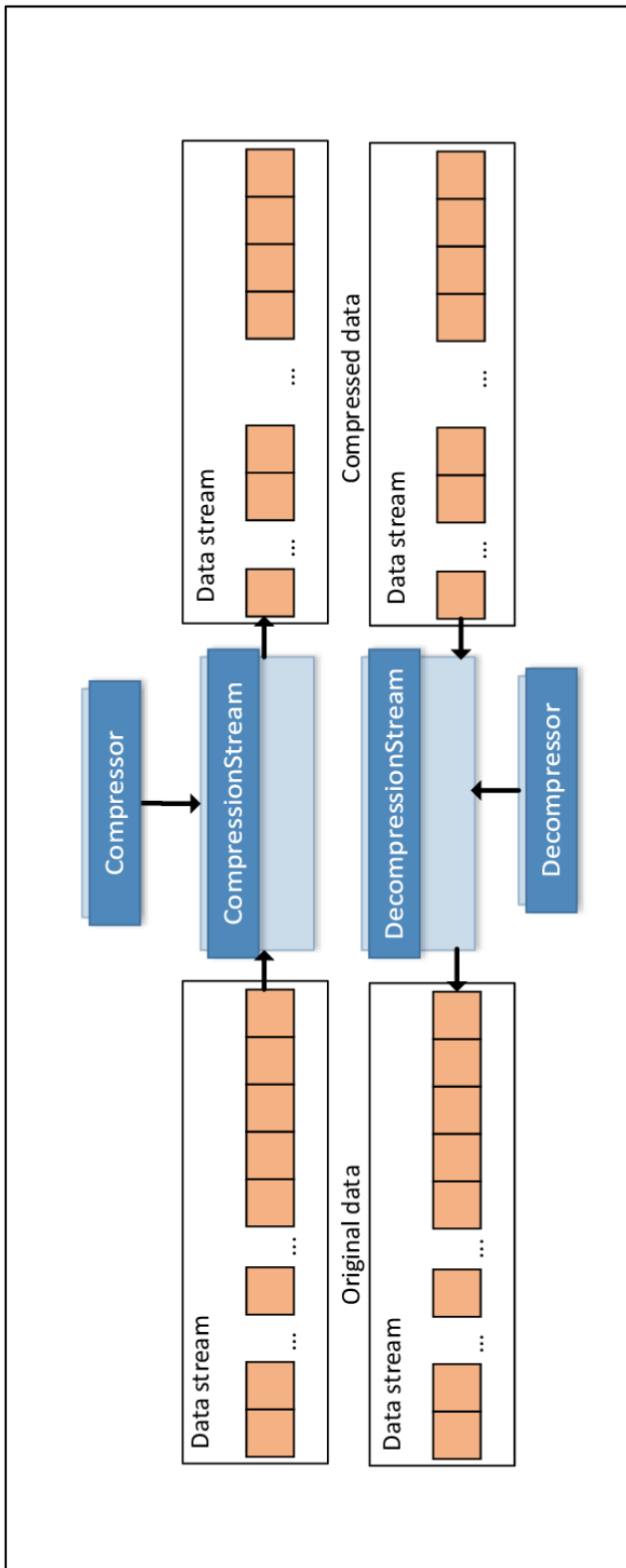
Python*: `./examples/python/source/serialization/serialization.py`

Data Compression

When large amounts of data are sent across devices or need to be stored in memory or in a persistent storage, data compression enables you to reduce network traffic, memory, and persistent storage footprint. Intel DAAL implements several most popular generic compression and decompression methods, which include ZLIB, LZO, RLE, and BZIP2.

General API for Data Compression and Decompression

The `CompressionStream` and `DecompressionStream` classes provide general methods for data compression and decompression. The following diagram illustrates the compression and decompression flow at a high level:



To define compression or decompression methods and related parameters, provide `Compressor` or `Decompressor` objects as arguments to `CompressionStream` or `DecompressionStream` constructors respectively. For more details on `Compressor` and `Decompressor`, refer to [Compression and Decompression Interfaces](#).

Use operator `<<` of `CompressionStream` or `DecompressionStream` to provide input data for compression or decompression stream. By default, all compression and decompression stream methods allocate the memory required to store results of compression and decompression. For details of controlling memory allocation, refer to [Compression and Decompression Interfaces](#).

The following methods are available to retrieve compressed data stored in `CompressionStream`:

- Copy compressed data blocks into a contiguous array using the `copyCompressedArray()` method.
You can define the data blocks to copy by specifying the number of bytes to copy. The method copies the data from the beginning of the stream and removes the copied data from `CompressionStream`, so next time you call the `copyCompressedArray()` method, it copies the next block of data. To copy all the data, before a call to `copyCompressedArray()`, call the `getCompressedBlocksSize()` method to get the total size of compressed data in the stream.
- Call the `getCompressedBlocksCollection()` method.
Unlike the `copyCompressedArray()` method, `getCompressedBlocksCollection()` does not copy compressed blocks but provides a reference to the collection of compressed data blocks. The collection is available until you call the `getCompressedBlocksCollection()` method next time.

The following methods are available to retrieve decompressed data stored in `DecompressionStream`:

- Copy decompressed data blocks into a contiguous array using the `copyDecompressedArray()` method.
You can define the data blocks to copy by specifying the number of bytes to copy. The method copies the data from the beginning of the stream and removes the copied data from `DecompressionStream`, so next time you call the `copyDecompressedArray()` method, it copies the next block of data. To copy all the data, before a call to `copyDecompressedArray()`, call the `getDecompressedBlocksSize()` method to get the total size of decompressed data in the stream.
- Call the `getDecompressedBlocksCollection()` method.
Unlike the `copyDecompressedArray()` method, `getDecompressedBlocksCollection()` does not copy decompressed blocks but provides a reference to the collection of decompressed data blocks. The collection is available until you call the `getDecompressedBlocksCollection()` method next time.

Compression and Decompression Interfaces

`CompressionStream` and `DecompressionStream` classes cover most typical usage scenarios. Therefore, you need to work directly with `Compressor` and `Decompressor` objects only in the cases as follows:

- `CompressionStream` and `DecompressionStream` classes do not cover your specific usage model.
- You want to control memory allocation and deallocation for results of compression and decompression.
- You need to modify compression and decompression default parameters.

The `Compressor` and `Decompressor` classes provide interfaces to supported compression and decompression methods (ZLIB, LZO, RLE, and BZIP2).

Compression and decompression objects are initialized with a set of default parameters. You can modify parameters of a specific compression method by accessing the `parameter` field of the `Compressor` or `Decompressor` object.

To perform compression or decompression using the `Compressor` or `Decompressor` classes, respectively, provide input data using the `setInputDataBlock()` method and call the `run()` method. This approach requires that you allocate and control the memory to store the results of compression or decompression. In general, it is impossible to accurately estimate the required size of the output data block, and the memory you provide may be insufficient to store results of compression or decompression. However, you can check

whether you need to allocate additional memory to continue the `run()` operation. To do this, use the `isOutputDataBlockFull()` method. You can also use the `getUsedOutputDataBlockSize()` method to obtain the size of compressed or decompressed data actually written to the output data block.

You can use your own compression and decompression methods in `CompressionStream` and `DecompressionStream`. In this case, you need to override `Compressor` and `Decompressor` objects.

Examples

Refer to the following examples in your Intel DAAL directory:

C++:

- `./examples/cpp/source/compression/compressor.cpp`
- `./examples/cpp/source/datasource/compression_batch.cpp`
- `./examples/cpp/source/datasource/compression_online.cpp`

Java*:

- `./examples/java/source/com/intel/daal/examples/compression/CompressorExample.java`
- `./examples/java/source/com/intel/daal/examples/datasource/CompressionBatch.java`
- `./examples/java/source/com/intel/daal/examples/datasource/CompressionOnline.java`

Python*:

- `./examples/python/source/compression/compressor.py`
- `./examples/python/source/datasource/compression_batch.py`
- `./examples/python/source/datasource/compression_online.py`

Data Model

The Data Model component of the Intel® Data Analytics Acceleration Library (Intel® DAAL) provides classes for model representation. The model mimics the actual data and represents it in a compact way so that you can use the library when the actual data is missing, incomplete, noisy or unavailable.

There are two categories of models in the library: Regression models and Classification models. Regression models are used to predict the values of dependent variables (responses) by observing independent variables. Classification models are used to predict to which sub-population (class) a given observation belongs.

A set of parameters characterizes each model. Intel DAAL model classes provide interfaces to access these parameters. It also provides the corresponding classes to train models, that is, to estimate model parameters using training data sets. As soon as a model is trained, it can be used for prediction and cross-validation. For this purpose, the library provides the corresponding prediction classes.

See Also

[Training and Prediction](#)

Algorithms

The Algorithms component of the Intel® Data Analytics Acceleration Library (Intel® DAAL) consists of classes that implement algorithms for data analysis (data mining), and data modeling (training and prediction). These algorithms include matrix decompositions, clustering algorithms, classification and regression algorithms, as well as association rules.

All Algorithms classes are derived from the base class `AlgorithmIface`. It provides interfaces for computations covering a variety of usage scenarios. Basic methods that you typically call are `compute()` and `finalizeCompute()`. In a very generic form algorithms accept one or several numeric tables or models as an input and return one or several numeric tables and models as an output. Algorithms may also require algorithm-specific parameters that you can modify by accessing the `parameter` field of the algorithm. Because most of algorithm parameters are preset with default values, you can often omit initialization of the parameter.

Algorithm Input

An algorithm can accept one or several numeric tables or models as an input. In computation modes that permit multiple calls to the `compute()` method, ensure that the structure of the input data, that is, the number of features, their order, and type, is the same for all the calls. The following methods are available to provide input to an algorithm:

<code>input.set(Input ID, InputData)</code>	Use to sets a pointer to the input argument with the <code>Input ID</code> identifier. This method overwrites previous input pointer stored in the algorithm.
<code>input.add(Input ID, InputData)</code>	Use in the distributed computation mode to add the pointers with the <code>Input ID</code> identifier. Unlike the <code>input.set()</code> method, <code>input.add()</code> does not overwrite the previously set input pointers, but stores all the input pointers until a call to the <code>compute()</code> method.
<code>input.get(Input ID)</code>	Use to get a reference to the pointer to the input data with the <code>Input ID</code> identifier.

For the input that each specific algorithm accepts, refer to the description of this algorithm.

Algorithm Output

Output of an algorithm can be one or several models or numeric tables. To retrieve the results of the algorithm computation, call the `getResult()` method. To access specific results, use the `get(Result ID)` method with the appropriate `Result ID` identifier. In the distributed processing mode, to get access to partial results of the algorithm computation, call the `getPartialResult()` method on each computation node. For a full list of algorithm computation results available, refer to the description of an appropriate algorithm.

By default, all algorithms allocate required memory to store partial and final results. Follow these steps to provide user allocated memory for partial or final results to the algorithm:

1. Create an object of an appropriate class for the results. For the classes supported, refer to the description of a specific algorithm.
2. Provide a pointer to that object to the algorithm by calling the `setPartialResult()` or `setResult()` method as appropriate.

3. Call the `compute()` method. After the call, the object created contains partial or final results.

Algorithm Parameters

Most of algorithms in Intel DAAL have a set of algorithm-specific parameters. Because most of the parameters are optional and preset with default values, you can often omit parameter modification. Provide required parameters to the algorithm using the constructor during algorithm initialization. If you need to change the parameters, you can do it by accessing the public field `parameter` of the algorithm. Some algorithms have an initialization procedure that sets or precomputes specific parameters needed to compute the algorithm. You can use the `InitializationProcedureIface` interface class to implement your own initialization procedure when the default implementation does not meet your specific needs.

Each algorithm also has generic parameters, such as the floating-point type, computation method, and computation step for the distributed processing mode. In C++, these parameters are defined as template parameters, and in most cases they are preset with default values. You can change the template parameters while declaring the algorithm. In Java, the generic parameters have no default values, and you need to define them in the constructor during algorithm initialization.

For a list of algorithm parameters, refer to the description of an appropriate algorithm.

Computation Modes

The library algorithms support the following computation modes:

- Batch processing
- Online processing
- Distributed processing

You can select the computation mode during initialization of the `Algorithm`.

For a list of computation parameters of a specific algorithm in each computation mode, possible input types, and output results, refer to the description of an appropriate algorithm.

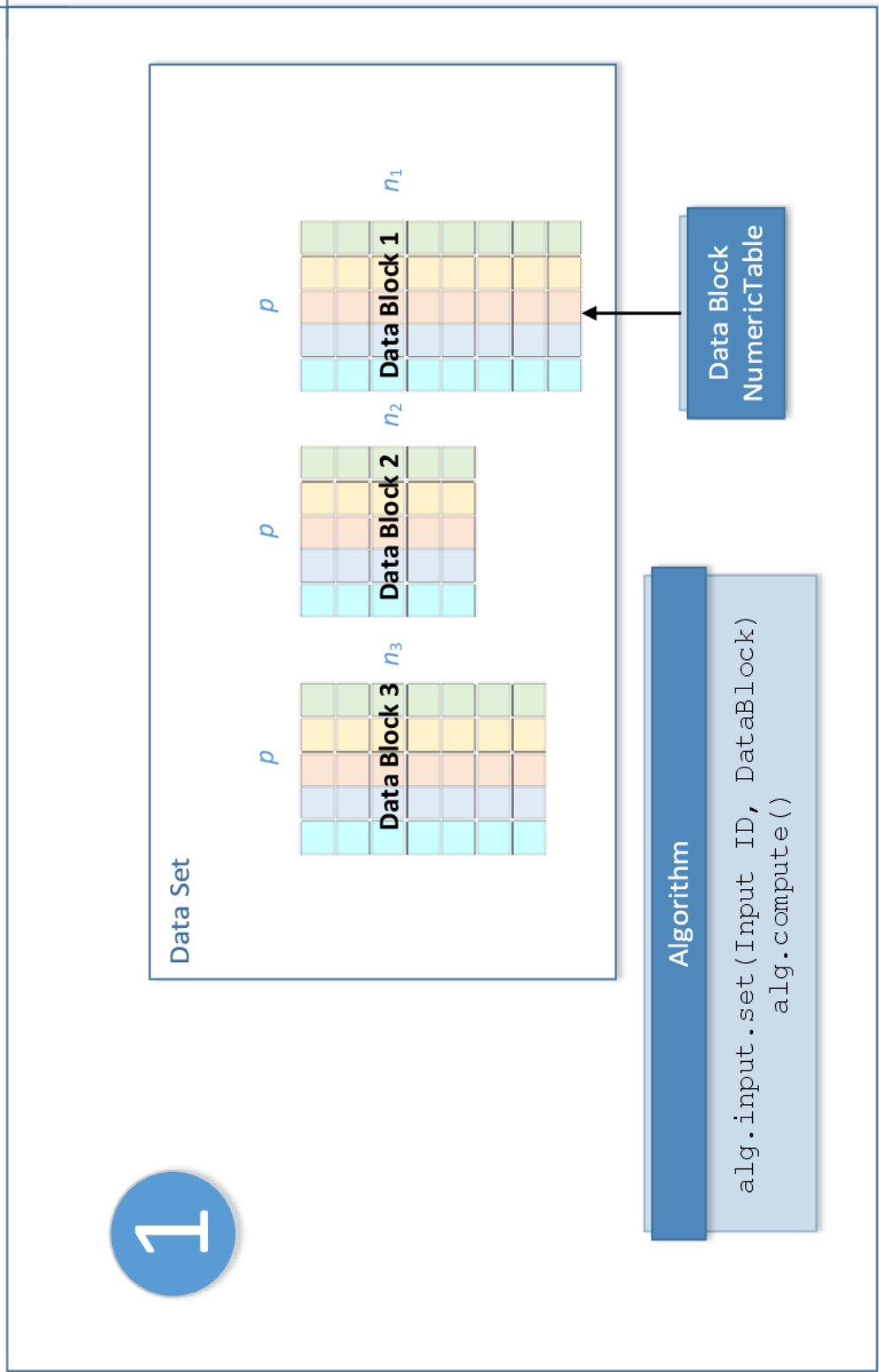
Batch Processing

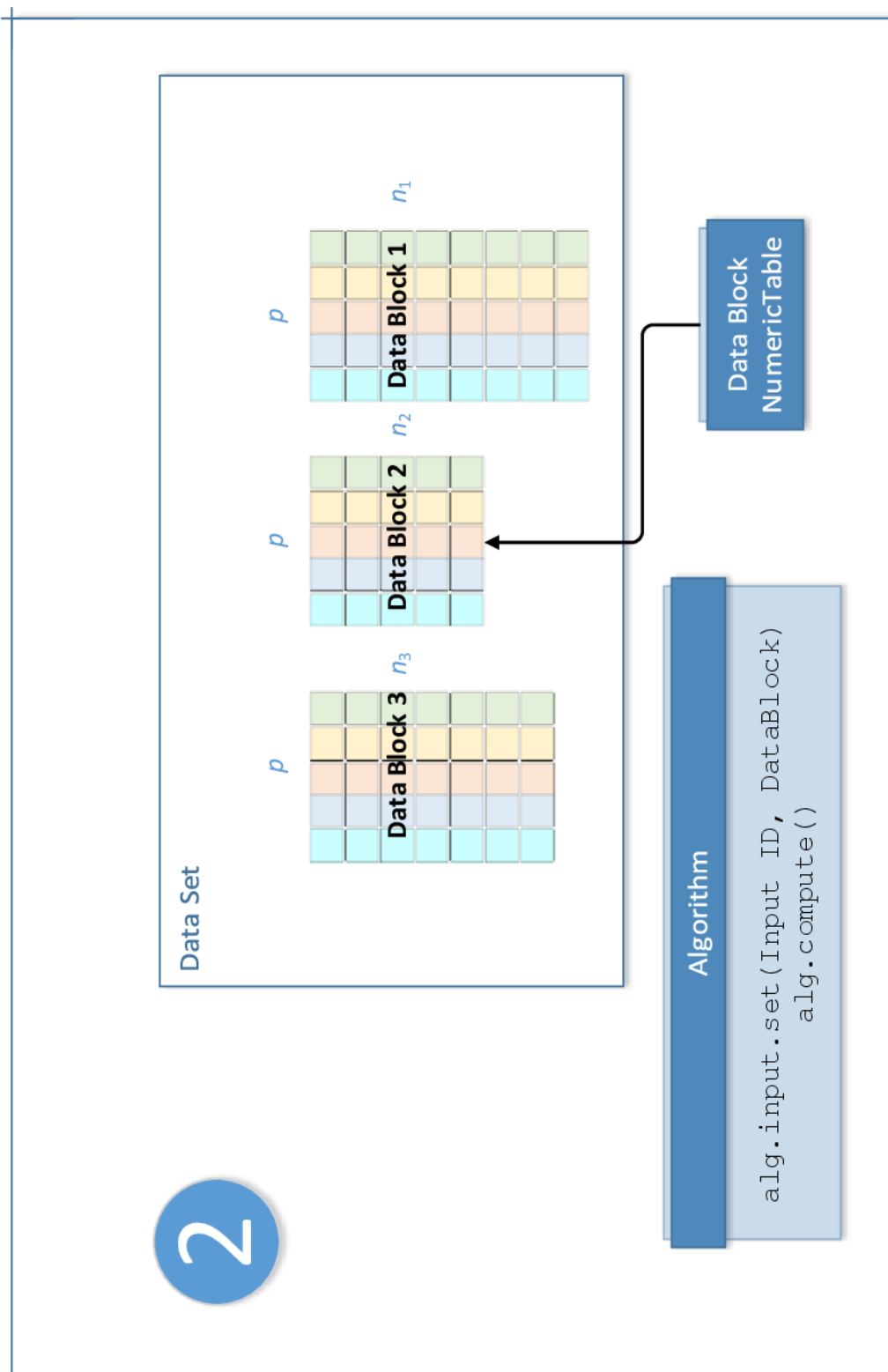
All Intel DAAL algorithms support at least the batch processing computation mode. In the batch processing mode, the only `compute` method of a particular algorithm class is used.

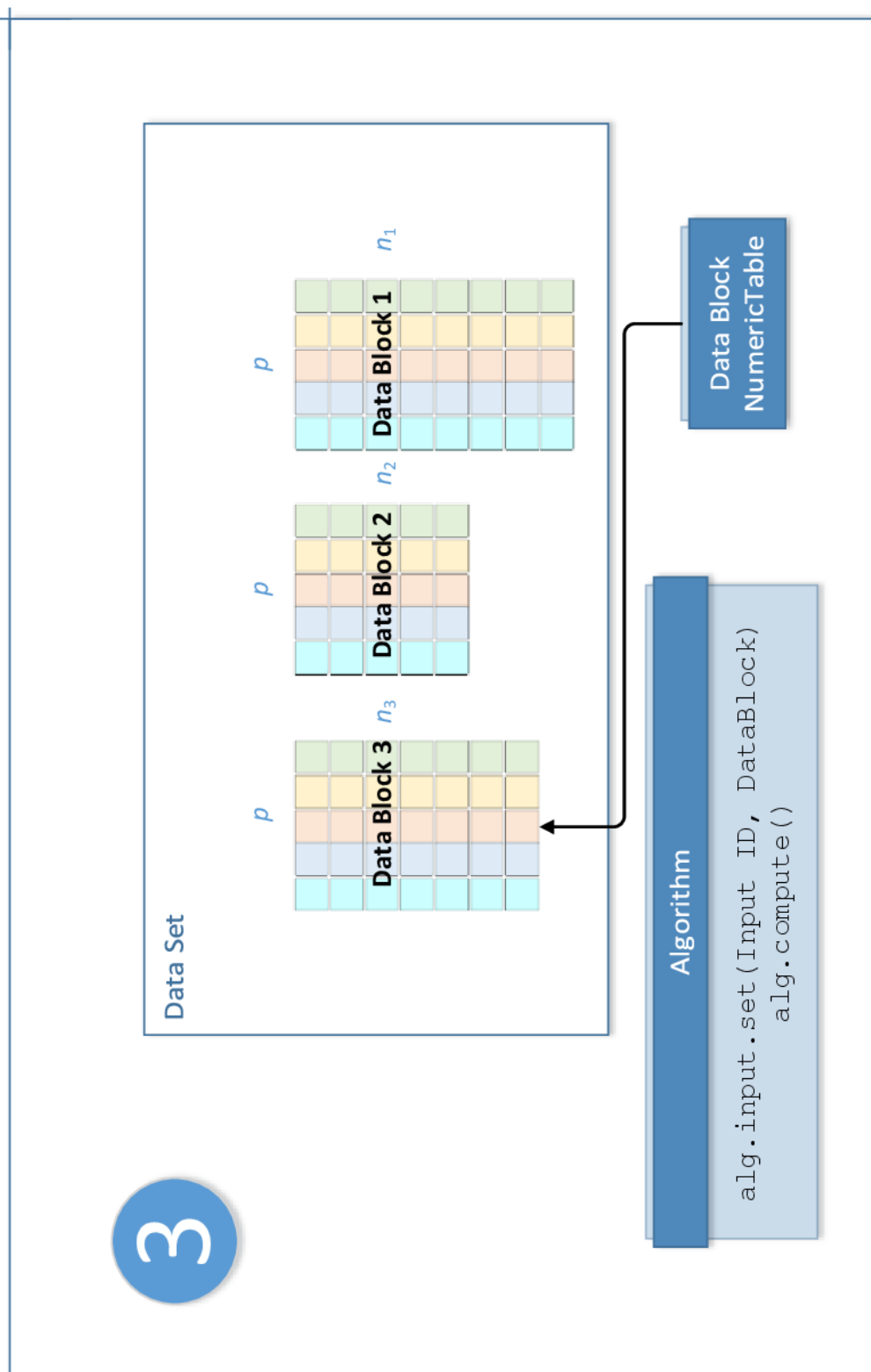
Online Processing

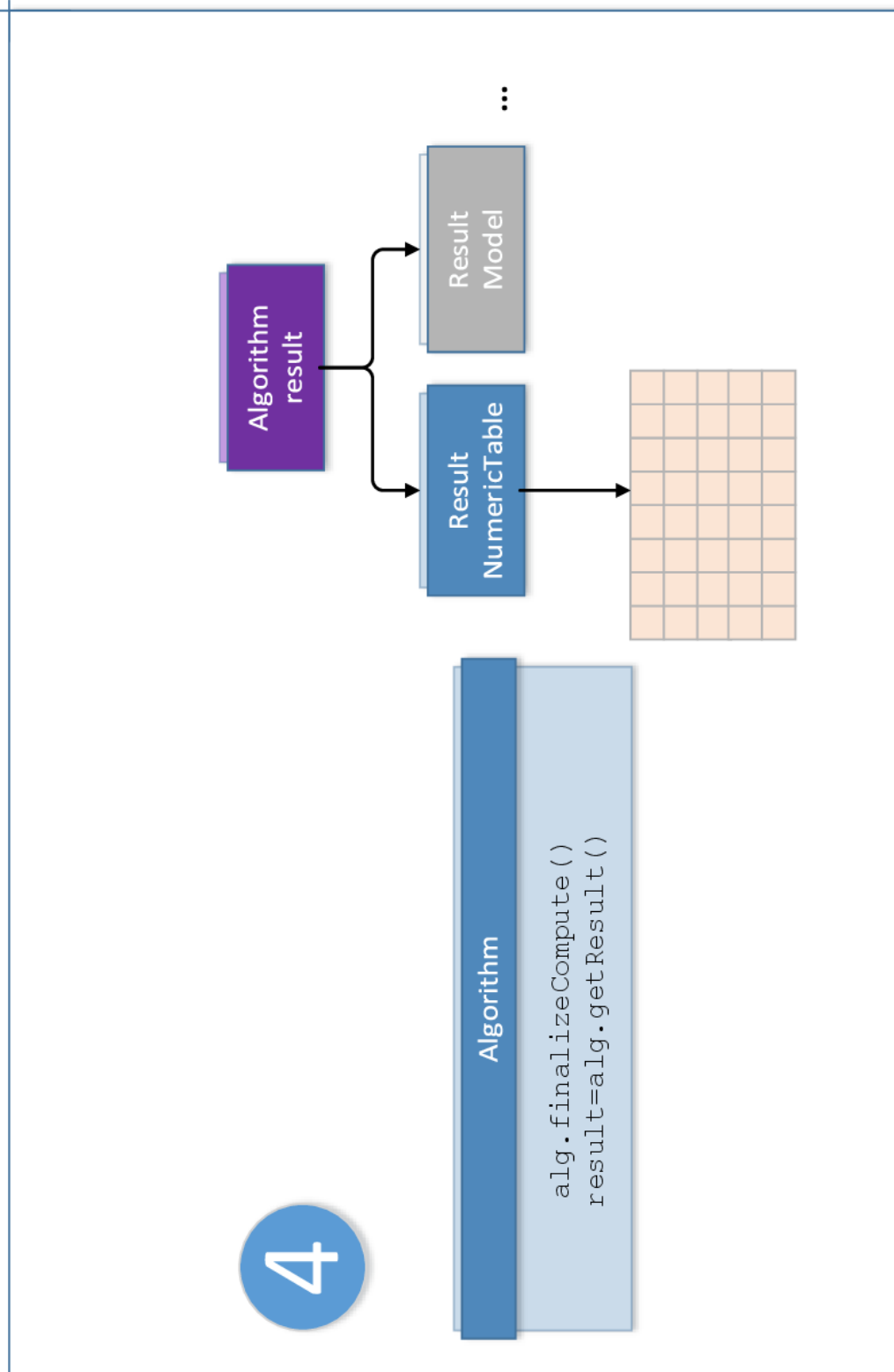
Some Intel DAAL algorithms enable processing of data sets in blocks. In the online processing mode, the `compute()`, and `finalizeCompute()` methods of a particular algorithm class are used. This computation mode assumes that the data arrives in blocks $i = 1, 2, 3, \dots, nblocks$. Call the `compute()` method each time new input becomes available. When the last block of data arrives, call the `finalizeCompute()` method to produce final results. If the input data arrives in an asynchronous mode, you can use the `getStatus()` method for a given data source to check whether a new block of data is available for load.

The following diagram illustrates the computation schema for online processing:







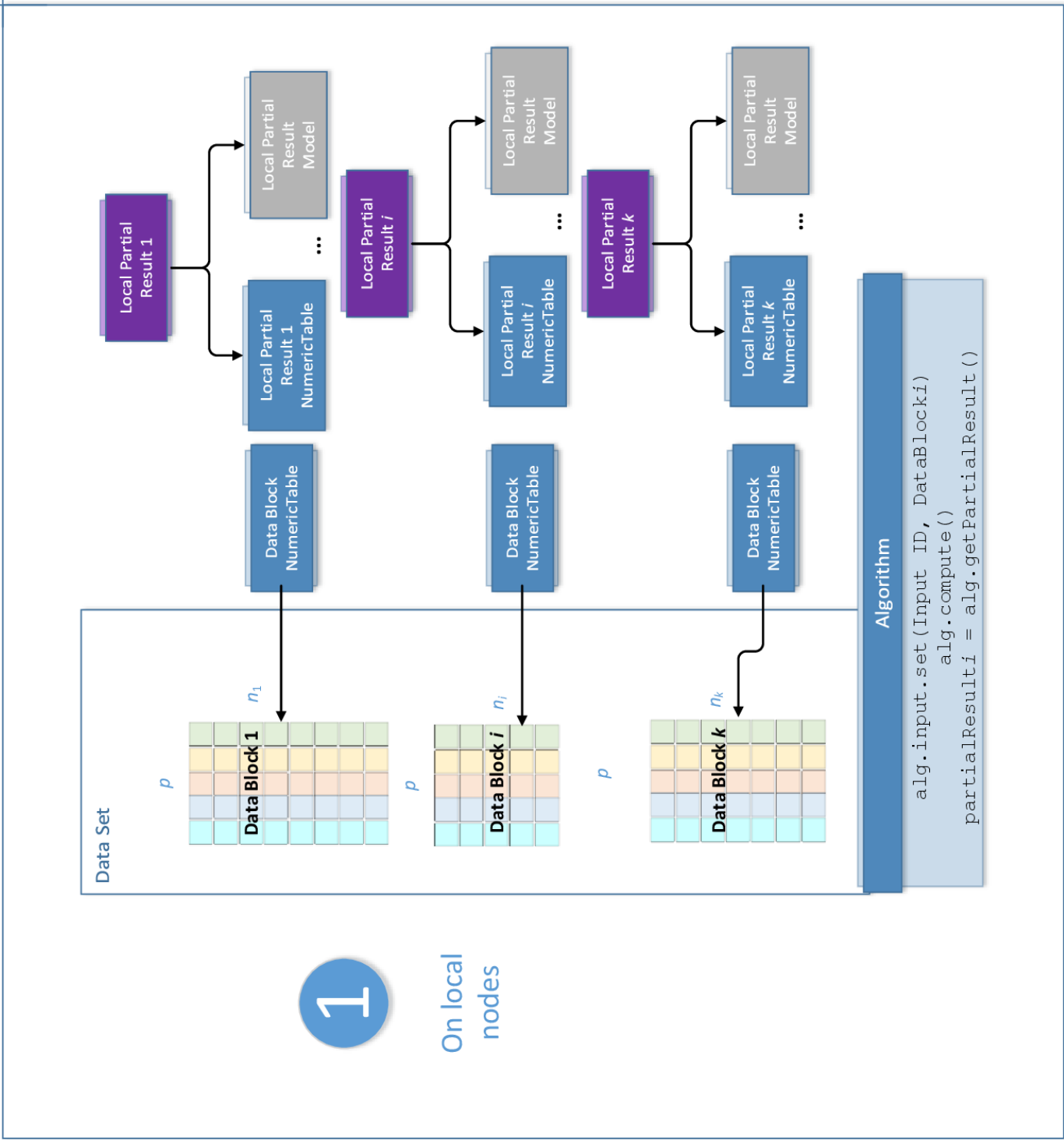


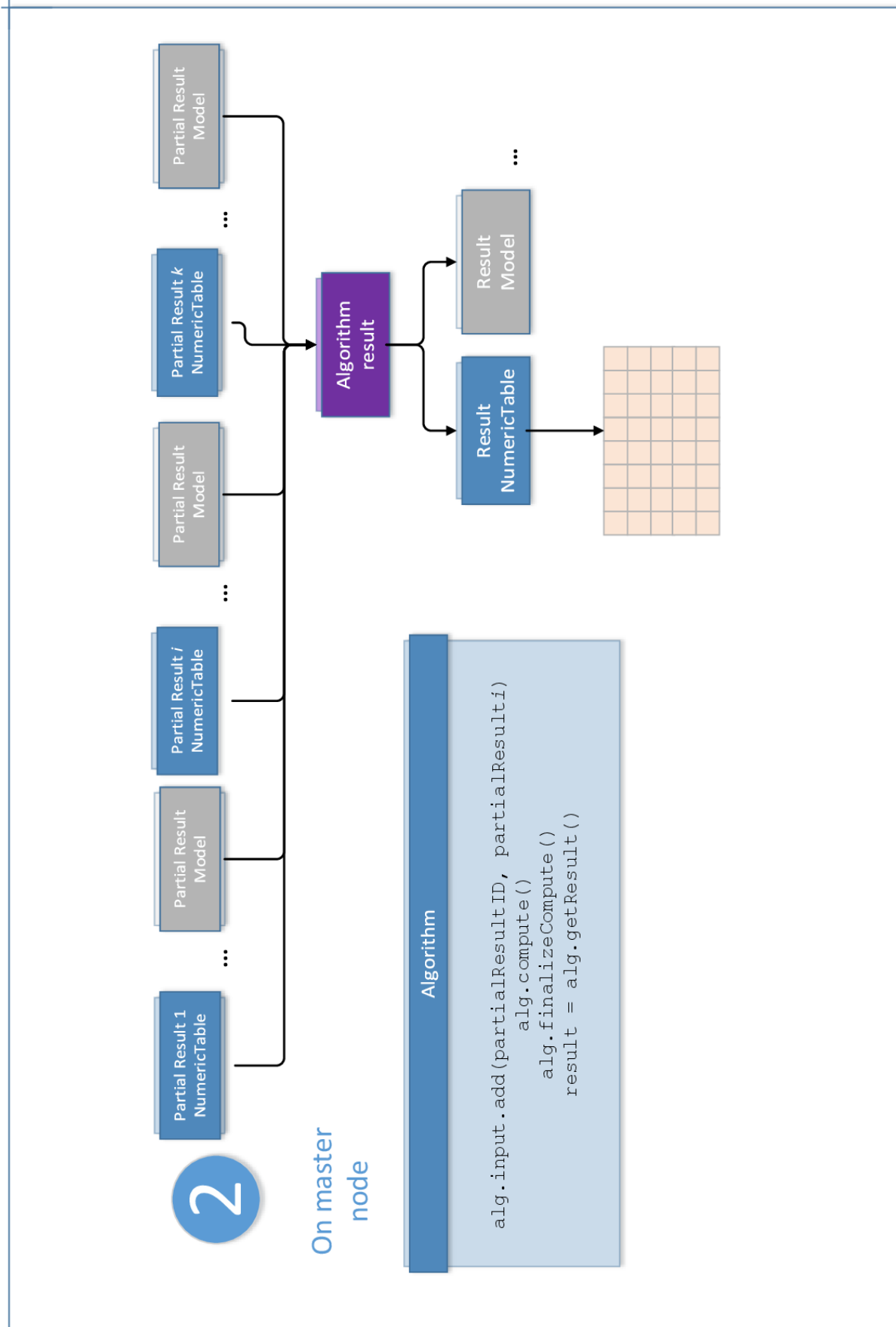
Distributed Processing

Some Intel DAAL algorithms enable processing of data sets distributed across several devices. In distributed processing mode, the `compute()`, and `finalizeCompute()` methods of a particular algorithm class are used. This computation mode assumes that the data set is split in *nblocks* blocks across computation nodes.

Computation is done in several steps. You need to define the computation step for an algorithm by providing the `computeStep` value to the constructor during initialization of the algorithm. Use the `compute()` method on each computation node to compute partial results. Use the `input.add()` method on the master node to add pointers to partial results processed on each computation node. When the last partial result arrives, call the `compute()` method followed by `finalizeCompute()` to produce final results. If the input data arrives in an asynchronous mode, you can use the `getStatus()` method for a given data source to check whether a new block of data is available for load.

The computation schema is algorithm-specific. The following diagram illustrates a typical computation schema for distribute processing:





For the algorithm-specific computation schema, refer to the Distributed Processing section in the description of an appropriate algorithm.

Distributed algorithms in Intel DAAL are abstracted from underlying cross-device communication technology, which enables use of the library in a variety of multi-device computing and data transfer scenarios. They include but are not limited to MPI* based cluster environments, Hadoop*/Spark* based cluster environments, low-level data exchange protocols, and more.

You can find typical use cases for MPI*, Hadoop*, and Spark* at [https://software.intel.com/en-us/product-code-samples?field_software_product_tid\[\]=79775](https://software.intel.com/en-us/product-code-samples?field_software_product_tid[]=79775).

Analysis

Intel DAAL analysis algorithms are intended to uncover the underlying structure of a data set and to characterize it by a set of quantitative measures, such as statistical moments, correlations coefficients, and so on. Some of the analysis algorithms also simplify the data: they transform, aggregate, or decompose that data in such a way that the transformed (aggregated or decomposed) data becomes easier to understand and characterize.

Moments of Low Order

Moments are basic quantitative measures of data set characteristics such as location and dispersion. Intel DAAL computes the following low order characteristics: minimums/maximums, sums, means, sums of squares, sums of squared differences from the means, second order raw moments, variances, standard deviations, and variations.

Details

Given a set X of n feature vectors $x_1 = (x_{11}, \dots, x_{1p})$, ..., $x_n = (x_{n1}, \dots, x_{np})$ of dimension p , the problem is to compute the following sample characteristics for each feature in the data set:

Statistic	Definition
Minimum	$\min(j) = \min_i x_{ij}$
Maximum	$\max(j) = \max_i x_{ij}$
Sum	$s(j) = \sum_i x_{ij}$
Sum of squares	$s_2(j) = \sum_i x_{ij}^2$
Means	$m(j) = \frac{s(j)}{n}$
Second order raw moment	$a_2(j) = \frac{s_2(j)}{n}$
Sum of squared difference from the means	$SDM(j) = \sum_i (x_{ij} - m(j))^2$
Variance	$k_2(j) = \frac{SDM(j)}{n-1}$
Standard deviation	$stddev(j) = \sqrt{k_2(j)}$

Statistic	Definition
Variation coefficient	$V(j) = \frac{stddev(j)}{m(j)}$

Batch Processing

Algorithm Input

The low order moments algorithm accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
<code>data</code>	Pointer to the numeric table of size $n \times p$ to compute moments for. While the input for <code>defaultDense</code> , <code>singlePassDense</code> , or <code>sumDense</code> method can be an object of any class derived from <code>NumericTable</code> , the input for <code>fastCSR</code> , <code>singlePassCSR</code> , or <code>sumCSR</code> method can only be an object of the <code>CSRNumericTable</code> class.

Algorithm Parameters

The low order moments algorithm has the following parameters:

Parameter	Default Value	Description
<code>algorithmFPTYPE</code>	<code>double</code>	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<code>method</code>	<code>defaultDense</code>	<p>Available methods for computation of low order moments:</p> <ul style="list-style-type: none"> <code>defaultDense</code> - default performance-oriented method <code>singlePassDense</code> - implementation of the single-pass algorithm proposed by D.H.D. West <code>sumDense</code> - implementation of the algorithm in the cases where the basic statistics associated with the numeric table are pre-computed sums; returns an error if pre-computed sums are not defined <code>fastCSR</code> - performance-oriented method for CSR numeric tables <code>singlePassCSR</code> - implementation of the single-pass algorithm proposed by D.H.D. West; optimized for CSR numeric tables <code>sumCSR</code> - implementation of the algorithm in the cases where the basic statistics associated with the numeric table are pre-computed sums; optimized for CSR numeric tables; returns an error if pre-computed sums are not defined

Parameter	Default Value	Description
<code>estimatesToCompute</code>	<code>estimatesAll</code>	Estimates to be computed by the algorithm: <ul style="list-style-type: none"> <code>estimatesAll</code> - all supported moments <code>estimatesMinMax</code> - minimum and maximum <code>estimatesMeanVariance</code> - mean and variance

Algorithm Output

The low order moments algorithm calculates the results described in the following table. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Each result is a pointer to the $1 \times p$ numeric table that contains characteristics for each feature in the data set. By default, the tables are objects of the `HomogenNumericTable` class, but you can define each table as an object of any class derived from `NumericTable` except `PackedSymmetricMatrix`, `PackedTriangularMatrix`, and `CSRNumericTable`.

Result ID	Characteristic
<code>minimum</code>	Minimums.
<code>maximum</code>	Maximums.
<code>sum</code>	Sums.
<code>sumSquares</code>	Sums of squares.
<code>sumSquaresCentered</code>	Sums of squared differences from the means.
<code>mean</code>	Estimates for the means.
<code>secondOrderRawMoment</code>	Estimates for the second order raw moments.
<code>variance</code>	Estimates for the variances.
<code>standardDeviation</code>	Estimates for the standard deviations.
<code>variation</code>	Estimates for the variations.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Examples

Refer to the following examples in your Intel DAAL directory:

C++:

- `./examples/cpp/source/moments/low_order_moments_dense_batch.cpp`
- `./examples/cpp/source/moments/low_order_moments_csr_batch.cpp`

Java*:

- `./examples/java/source/com/intel/daal/examples/moments/`
`LowOrderMomentsDenseBatch.java`
- `./examples/java/source/com/intel/daal/examples/moments/LowOrderMomentCSRBatch.java`

Python*:

- `./examples/python/source/moments/low_order_moments_dense_batch.py`
- `./examples/python/source/moments/low_order_moments_csr_batch.py`

See Also

[Handling Errors](#)

Online Processing

Online processing computation mode assumes that data arrives in blocks $i = 1, 2, 3, \dots nblocks$.

Computation of low order moments in the online processing mode follows the general computation schema for online processing described in [Algorithms](#).

Algorithm Input

The low order moments algorithm accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
<code>data</code>	Pointer to the numeric table of size $n_i \times p$ that represents the current data block. While the input for <code>defaultDense</code> , <code>singlePassDense</code> , or <code>sumDense</code> method can be an object of any class derived from <code>NumericTable</code> , the input for <code>fastCSR</code> , <code>singlePassCSR</code> , or <code>sumCSR</code> method can only be an object of the <code>CSRNumericTable</code> class.

Algorithm Parameters

The low order moments algorithm has the following parameters:

Parameter	Default Value	Description
<code>algorithmFPType</code>	<code>double</code>	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<code>method</code>	<code>defaultDense</code>	Available methods for computation of low order moments: <ul style="list-style-type: none"> • <code>defaultDense</code> - default performance-oriented method • <code>singlePassDense</code> - implementation of the single-pass algorithm proposed by D.H.D. West

Parameter	Default Value	Description
		<ul style="list-style-type: none"> <code>sumDense</code> - implementation of the algorithm in the cases where the basic statistics associated with the numeric table are pre-computed sums; returns an error if pre-computed sums are not defined <code>fastCSR</code> - performance-oriented method for CSR numeric tables <code>singlePassCSR</code> - implementation of the single-pass algorithm proposed by D.H.D. West; optimized for CSR numeric tables <code>sumCSR</code> - implementation of the algorithm in the cases where the basic statistics associated with the numeric table are pre-computed sums; optimized for CSR numeric tables; returns an error if pre-computed sums are not defined
<i>initialization Procedure</i>	Not applicable	<p>The procedure for setting initial parameters of the algorithm in the online processing mode. By default, the algorithm does the following initialization:</p> <ul style="list-style-type: none"> Sets <code>nObservations</code>, <code>partialSum</code>, and <code>partialSumSquares</code> to zero Sets <code>partialMinimum</code> and <code>partialMaximum</code> to the first row of the input table.
<i>estimatesToCompute</i>	<code>estimatesAll</code>	<p>Estimates to be computed by the algorithm:</p> <ul style="list-style-type: none"> <code>estimatesAll</code> - all supported moments <code>estimatesMinMax</code> - minimum and maximum <code>estimatesMeanVariance</code> - mean and variance

Partial Results

The low order moments algorithm in the online processing mode calculates partial results described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
<code>nObservations</code>	Pointer 1 x 1 numeric table that contains the number of rows processed so far. By default, this result is an object of the <code>HomogenNumericTable</code> class, but you can define the result as an object of any class derived from <code>NumericTable</code> except <code>CSRNumericTable</code> .

Partial characteristics computed so far, each in a 1 x *p* numeric table. By default, each table is an object of the `HomogenNumericTable` class, but you can define the tables as objects of any class derived from `NumericTable` except `PackedSymmetricMatrix`, `PackedTriangularMatrix`, and `CSRNumericTable`.

Result ID	Result
<code>partialMinimum</code>	Partial minimums.
<code>partialMaximum</code>	Partial maximums.
<code>partialSum</code>	Partial sums.
<code>partialSumSquares</code>	Partial sums of squares.
<code>partialSumSquaresCentered</code>	Partial sums of squared differences from the means.

Algorithm Output

The low order moments algorithm calculates the results described in the following table. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Each result is a pointer to the $1 \times p$ numeric table that contains characteristics for each feature in the data set. By default, the tables are objects of the `HomogenNumericTable` class, but you can define each table as an object of any class derived from `NumericTable` except `PackedSymmetricMatrix`, `PackedTriangularMatrix`, and `CSRNumericTable`.

Result ID	Characteristic
<code>minimum</code>	Minimums.
<code>maximum</code>	Maximums.
<code>sum</code>	Sums.
<code>sumSquares</code>	Sums of squares.
<code>sumSquaresCentered</code>	Sums of squared differences from the means.
<code>mean</code>	Estimates for the means.
<code>secondOrderRawMoment</code>	Estimates for the second order raw moments.
<code>variance</code>	Estimates for the variances.
<code>standartDeviation</code>	Estimates for the standard deviations.
<code>variation</code>	Estimates for the variations.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Examples

Refer to the following examples in your Intel DAAL directory:

C++:

- `./examples/cpp/source/moments/low_order_moments_dense_online.cpp`
- `./examples/cpp/source/moments/low_order_moments_csr_online.cpp`

Java*:

- `./examples/java/source/com/intel/daal/examples/moments/LowOrderMomentsDenseOnline.java`
- `./examples/java/source/com/intel/daal/examples/moments/LowOrderMomentsCSROnline.java`

Python*:

- `./examples/python/source/moments/low_order_moments_dense_online.py`
- `./examples/python/source/moments/low_order_moments_csr_online.py`

See Also

[Handling Errors](#)

Distributed Processing

This mode assumes that the data set is split into *nblocks* blocks across computation nodes.

Algorithm Parameters

The low order moments algorithm in the distributed processing mode has the following parameters:

Parameter	Default Value	Description
<i>computeStep</i>	Not applicable	The parameter required to initialize the algorithm. Can be: <ul style="list-style-type: none"> • <code>step1Local</code> - the first step, performed on local nodes • <code>step2Master</code> - the second step, performed on a master node
<i>algorithmFPType</i>	double	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<i>method</i>	defaultDense	Available methods for computation of low order moments: <ul style="list-style-type: none"> • <code>defaultDense</code> - default performance-oriented method • <code>singlePassDense</code> - implementation of the single-pass algorithm proposed by D.H.D. West • <code>sumDense</code> - implementation of the algorithm in the cases where the basic statistics associated with the numeric table are pre-computed sums; returns an error if pre-computed sums are not defined • <code>fastCSR</code> - performance-oriented method for CSR numeric tables • <code>singlePassCSR</code> - implementation of the single-pass algorithm proposed by D.H.D. West; optimized for CSR numeric tables

Parameter	Default Value	Description
		<ul style="list-style-type: none"> sumCSR - implementation of the algorithm in the cases where the basic statistics associated with the numeric table are pre-computed sums; optimized for CSR numeric tables; returns an error if pre-computed sums are not defined
<code>estimatesToCompute</code>	<code>estimatesAll</code>	<p>Estimates to be computed by the algorithm:</p> <ul style="list-style-type: none"> <code>estimatesAll</code> - all supported moments <code>estimatesMinMax</code> - minimum and maximum <code>estimatesMeanVariance</code> - mean and variance

Computation of low order moments follows the general schema described in [Algorithms](#):

Step 1 - on Local Nodes

In this step, the low order moments algorithm accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
<code>data</code>	<p>Pointer to the $n_i \times p$ numeric table that represents the i-th data block on the local node.</p> <p>While the input for <code>defaultDense</code>, <code>singlePassDense</code>, or <code>sumDense</code> method can be an object of any class derived from <code>NumericTable</code>, the input for <code>fastCSR</code>, <code>singlePassCSR</code>, or <code>sumCSR</code> method can only be an object of the <code>CSRNumericTable</code> class.</p>

In this step, the low order moments algorithm calculates the results described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
<code>nObservations</code>	<p>Pointer 1 x 1 numeric table that contains the number of observations processed so far on the local node. By default, this result is an object of the <code>HomogenNumericTable</code> class, but you can define the result as an object of any class derived from <code>NumericTable</code> except <code>CSRNumericTable</code>.</p>
<code>partialMinimum</code>	Partial minimums.
<code>partialMaximum</code>	Partial maximums.
<code>partialSum</code>	Partial sums.

Partial characteristics computed so far on the local node, each in a $1 \times p$ numeric table. By default, each table is an object of the `HomogenNumericTable` class, but you can define the tables as objects of any class derived from `NumericTable` except `PackedSymmetricMatrix`, `PackedTriangularMatrix`, and `CSRNumericTable`.

Result ID	Result
<code>partialSumSquares</code>	Partial sums of squares.
<code>partialSumSquaresCentered</code>	Partial sums of squared differences from the means.

Step 2 - on Master Node

In this step, the low order moments algorithm accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
<code>partialResults</code>	A collection that contains numeric tables with partial results computed in Step 1 on local nodes (six numeric tables from each local node). These numeric tables can be objects of any class derived from the <code>NumericTable</code> class except <code>PackedSymmetricMatrix</code> and <code>PackedTriangularMatrix</code> .

In this step, the low order moments algorithm calculates the results described in the following table. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Each result is a pointer to the $1 \times p$ numeric table that contains characteristics for each feature in the data set. By default, the tables are objects of the `HomogenNumericTable` class, but you can define each table as an object of any class derived from `NumericTable` except `PackedSymmetricMatrix`, `PackedTriangularMatrix`, and `CSRNumericTable`.

Result ID	Characteristic
<code>minimum</code>	Minimums.
<code>maximum</code>	Maximums.
<code>sum</code>	Sums.
<code>sumSquares</code>	Sums of squares.
<code>sumSquaresCentered</code>	Sums of squared differences from the means.
<code>mean</code>	Estimates for the means.
<code>secondOrderRawMoment</code>	Estimates for the second order raw moments.
<code>variance</code>	Estimates for the variances.
<code>standardDeviation</code>	Estimates for the standard deviations.
<code>variation</code>	Estimates for the variations.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Optimization Notice

Notice revision #20110804

Examples

Refer to the following examples in your Intel DAAL directory:

C++:

- `./examples/cpp/source/moments/low_order_moments_dense_distributed.cpp`
- `./examples/cpp/source/moments/low_order_moments_csr_distributed.cpp`

Java*:

- `./examples/java/source/com/intel/daal/examples/moments/LowOrderMomentsDenseDistributed.java`
- `./examples/java/source/com/intel/daal/examples/moments/LowOrderMomentsCSR Distributed.java`

Python*:

- `./examples/python/source/moments/low_order_moments_dense_distributed.py`
- `./examples/python/source/moments/low_order_moments_csr_distributed.py`

See Also

[Handling Errors](#)

Performance Considerations

To get the best overall performance when computing low order moments:

- If input data is homogeneous, provide the input data and store results in homogeneous numeric tables of the same type as specified in the `algorithmFPTType` class template parameter.
- If input data is non-homogeneous, use AOS layout rather than SOA layout.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Quantile

Quantile is an algorithm to analyze the distribution of observations. Quantiles are the values that divide the distribution so that a given portion of observations is below the quantile.

Details

Given a set X of p features $x_1 = (x_{11}, \dots, x_{n1})$, ..., $x_p = (x_{1p}, \dots, x_{np})$ and the quantile orders $\beta = (\beta_1, \dots, \beta_m)$, the problem is to compute z_{ik} such that $P\{\xi_i \leq z_{ik}\} \geq \beta_k$ and $P\{\xi_i > z_{ik}\} \leq 1 - \beta_k$, where

- $x_i = (x_{1i}, \dots, x_{ni})$ are observations of a random variable ξ_i that represents the i -th feature

- P is the probability measure
- $i = 1, \dots, p$
- $k = 1, \dots, m$

Batch Processing

Algorithm Input

The quantile algorithm accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
<code>data</code>	Pointer to the $n \times p$ numeric table that contains the input data set. This table can be an object of any class derived from <code>NumericTable</code> .

Algorithm Parameters

The quantile algorithm has the following parameters:

Parameter	Default Value	Description
<code>algorithmFPType</code>	<code>double</code>	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<code>method</code>	<code>defaultDense</code>	Performance-oriented computation method, the only method supported by the algorithm.
<code>quantileOrders</code>	<code>0.5</code>	The $1 \times m$ numeric table with quantile orders.

Algorithm Output

The quantile algorithm calculates the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
<code>quantiles</code>	Pointer to the $p \times m$ numeric table with the quantiles. By default, this result is an object of the <code>HomogenNumericTable</code> class, but you can define the result as an object of any class derived from <code>NumericTable</code> except <code>PackedSymmetricMatrix</code> , <code>PackedTriangularMatrix</code> , and <code>CSRNumericTable</code> .

Examples

Refer to the following examples in your Intel DAAL directory:

C++: `./examples/cpp/source/quantiles/quantiles_batch.cpp`

Java*: `./examples/java/source/com/intel/daal/examples/quantiles/QuantilesBatch.java`

Python*: `./examples/python/source/quantiles/quantiles_batch.py`

Correlation and Variance-Covariance Matrices

Variance-covariance and correlation matrices are among the most important quantitative measures of a data set that characterize statistical relationships involving dependence.

Specifically, the covariance measures the extent to which variables "fluctuate together" (that is, co-vary). The correlation is the covariance normalized to be between -1 and +1. A positive correlation indicates the extent to which variables increase or decrease simultaneously. A negative correlation indicates the extent to which one variable increases while the other one decreases. Values close to +1 and -1 indicate a high degree of linear dependence between variables.

Details

Given a set X of n feature vectors $x_1 = (x_{11}, \dots, x_{1p})$, ..., $x_n = (x_{n1}, \dots, x_{np})$ of dimension p , the problem is to compute the sample means and variance-covariance matrix or correlation matrix:

Statistic	Definition
Means	$M = (m(1), \dots, m(p))$, where $m(j) = \frac{1}{n} \sum_i x_{ij}$
Variance-covariance matrix	$Cov = (v_{ij})$, where $v_{ij} = \frac{1}{n-1} \sum_{k=1}^n (x_{ki} - m(i))(x_{kj} - m(j)), i = \overline{1, p}, j = \overline{1, p}$
Correlation matrix	$Cor = (c_{ij})$, where $c_{ij} = \frac{v_{ij}}{\sqrt{v_{ii} \cdot v_{jj}}}, i = \overline{1, p}, j = \overline{1, p}$

Batch Processing

Algorithm Input

The correlation and variance-covariance matrices algorithm accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
data	Pointer to the $n \times p$ numeric table for which the variance-covariance or correlation matrix C is computed. While the input for <code>defaultDense</code> , <code>singlePassDense</code> , or <code>sumDense</code> method can be an object of any class derived from <code>NumericTable</code> , the input for <code>fastCSR</code> , <code>singlePassCSR</code> , or <code>sumCSR</code> method can only be an object of the <code>CSRNumericTable</code> class.

Algorithm Parameters

The correlation and variance-covariance matrices algorithm has the following parameters:

Parameter	Default Value	Description
<code>algorithmFPTYPE</code>	double	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .

Parameter	Default Value	Description
<i>method</i>	<code>defaultDense</code>	<p>Available methods for computation of correlation and variance-covariance matrices:</p> <ul style="list-style-type: none"> <code>defaultDense</code> - default performance-oriented method <code>singlePassDense</code> - implementation of the single-pass algorithm proposed by D.H.D. West <code>sumDense</code> - implementation of the algorithm in the cases where the basic statistics associated with the numeric table are pre-computed sums; returns an error if pre-computed sums are not defined <code>fastCSR</code> - performance-oriented method for CSR numeric tables <code>singlePassCSR</code> - implementation of the single-pass algorithm proposed by D.H.D. West; optimized for CSR numeric tables <code>sumCSR</code> - implementation of the algorithm in the cases where the basic statistics associated with the numeric table are pre-computed sums; optimized for CSR numeric tables; returns an error if pre-computed sums are not defined
<i>outputMatrixType</i>	<code>covarianceMatrix</code>	<p>The type of the output matrix. Can be:</p> <ul style="list-style-type: none"> <code>covarianceMatrix</code> - variance-covariance matrix <code>correlationMatrix</code> - correlation matrix

Algorithm Output

The correlation and variance-covariance matrices algorithm calculates the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
<code>covariance</code>	Use when <code>outputMatrixType=covarianceMatrix</code> . Pointer to the numeric table with the $p \times p$ variance-covariance matrix. By default, this result is an object of the <code>HomogenNumericTable</code> class, but you can define the result as an object of any class derived from <code>NumericTable</code> except <code>PackedTriangularMatrix</code> and <code>CSRNumericTable</code> .
<code>correlation</code>	Use when <code>outputMatrixType=correlationMatrix</code> . Pointer to the numeric table with the $p \times p$ correlation matrix. By default, this result is an object of the <code>HomogenNumericTable</code> class, but you can define the result as an object of any class derived from <code>NumericTable</code> except <code>PackedTriangularMatrix</code> and <code>CSRNumericTable</code> .

Result ID	Result
mean	Pointer to the $1 \times p$ numeric table with means. By default, this result is an object of the <code>HomogenNumericTable</code> class, but you can define the result as an object of any class derived from <code>NumericTable</code> except <code>PackedTriangularMatrix</code> , <code>PackedSymmetricMatrix</code> , and <code>CSRNumericTable</code> .

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Examples

Refer to the following examples in your Intel DAAL directory:

C++:

- `./examples/cpp/source/covariance/covariance_dense_batch.cpp`
- `./examples/cpp/source/covariance/covariance_csr_batch.cpp`

Java*:

- `./examples/java/source/com/intel/daal/examples/covariance/CovarianceDenseBatch.java`
- `./examples/java/source/com/intel/daal/examples/covariance/CovarianceCSRBatch.java`

Python*:

- `./examples/python/source/covariance/covariance_dense_batch.py`
- `./examples/python/source/covariance/covariance_csr_batch.py`

See Also

[Handling Errors](#)

Online Processing

Online processing computation mode assumes that data arrives in blocks $i = 1, 2, 3, \dots nblocks$.

Computation of correlation and variance-covariance matrices in the online processing mode follows the general computation schema for online processing described in [Algorithms](#).

Algorithm Input

The correlation and variance-covariance matrices algorithm in the online processing mode accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
<code>data</code>	Pointer to the numeric table of size $n_i \times p$ that represents the current data block. While the input for <code>defaultDense</code> , <code>singlePassDense</code> , or <code>sumDense</code> method can be an object of any class derived from <code>NumericTable</code> , the input for <code>fastCSR</code> , <code>singlePassCSR</code> , or <code>sumCSR</code> method can only be an object of the <code>CSRNumericTable</code> class.

Algorithm Parameters

The correlation and variance-covariance matrices algorithm has the following parameters in the online processing mode:

Parameter	Default Value	Description
<code>algorithmFPType</code>	<code>double</code>	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<code>method</code>	<code>defaultDense</code>	<p>Available methods for computation of correlation and variance-covariance matrices:</p> <ul style="list-style-type: none"> <code>defaultDense</code> - default performance-oriented method <code>singlePassDense</code> - implementation of the single-pass algorithm proposed by D.H.D. West <code>sumDense</code> - implementation of the algorithm in the cases where the basic statistics associated with the numeric table are pre-computed sums; returns an error if pre-computed sums are not defined <code>fastCSR</code> - performance-oriented method for CSR numeric tables <code>singlePassCSR</code> - implementation of the single-pass algorithm proposed by D.H.D. West; optimized for CSR numeric tables <code>sumCSR</code> - implementation of the algorithm in the cases where the basic statistics associated with the numeric table are pre-computed sums; optimized for CSR numeric tables; returns an error if pre-computed sums are not defined
<code>outputMatrixType</code>	<code>covarianceMatrix</code>	<p>The type of the output matrix. Can be:</p> <ul style="list-style-type: none"> <code>covarianceMatrix</code> - variance-covariance matrix <code>correlationMatrix</code> - correlation matrix

Parameter	Default Value	Description
<i>initialization Procedure</i>	Not applicable	The procedure for setting initial parameters of the algorithm in the online processing mode. By default, the algorithm sets the <code>nObservations</code> , <code>sum</code> , and <code>crossProduct</code> parameters to zero.

Partial Results

The correlation and variance-covariance matrices algorithm in the online processing mode calculates partial results described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
<code>nObservations</code>	Pointer 1 x 1 numeric table that contains the number of observations processed so far. By default, this result is an object of the <code>HomogenNumericTable</code> class, but you can define the result as an object of any class derived from <code>NumericTable</code> except <code>CSRNumericTable</code> .
<code>crossProduct</code>	Pointer $p \times p$ numeric table with the cross-product matrix computed so far. By default, this table is an object of the <code>HomogenNumericTable</code> class, but you can define the result as an object of any class derived from <code>NumericTable</code> except <code>PackedSymmetricMatrix</code> , <code>PackedTriangularMatrix</code> , and <code>CSRNumericTable</code> .
<code>sum</code>	Pointer 1 x p numeric table with partial sums computed so far. By default, this table is an object of the <code>HomogenNumericTable</code> class, but you can define the result as an object of any class derived from <code>NumericTable</code> except <code>PackedSymmetricMatrix</code> , <code>PackedTriangularMatrix</code> , and <code>CSRNumericTable</code> .

Algorithm Output

The correlation and variance-covariance matrices algorithm calculates the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
<code>covariance</code>	Use when <code>outputMatrixType=covarianceMatrix</code> . Pointer to the numeric table with the $p \times p$ variance-covariance matrix. By default, this result is an object of the <code>HomogenNumericTable</code> class, but you can define the result as an object of any class derived from <code>NumericTable</code> except <code>PackedTriangularMatrix</code> and <code>CSRNumericTable</code> .
<code>correlation</code>	Use when <code>outputMatrixType=correlationMatrix</code> . Pointer to the numeric table with the $p \times p$ correlation matrix. By default, this result is an object of the <code>HomogenNumericTable</code> class, but you can define the result as an object of any class derived from <code>NumericTable</code> except <code>PackedTriangularMatrix</code> and <code>CSRNumericTable</code> .

Result ID	Result
mean	Pointer to the $1 \times p$ numeric table with means. By default, this result is an object of the <code>HomogenNumericTable</code> class, but you can define the result as an object of any class derived from <code>NumericTable</code> except <code>PackedTriangularMatrix</code> , <code>PackedSymmetricMatrix</code> , and <code>CSRNumericTable</code> .

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Examples

Refer to the following examples in your Intel DAAL directory:

C++:

- `./examples/cpp/source/covariance/covariance_dense_online.cpp`
- `./examples/cpp/source/covariance/covariance_csr_online.cpp`

Java*:

- `./examples/java/source/com/intel/daal/examples/covariance/CovarianceDenseOnline.java`
- `./examples/java/source/com/intel/daal/examples/covariance/CovarianceCSROnline.java`

Python*:

- `./examples/python/source/covariance/covariance_dense_online.py`
- `./examples/python/source/covariance/covariance_csr_online.py`

See Also

[Handling Errors](#)

Distributed Processing

This mode assumes that the data set is split into *nblocks* blocks across computation nodes.

Algorithm Parameters

The correlation and variance-covariance matrices algorithm in the distributed processing mode has the following parameters:

Parameter	Default Value	Description
<code>computeStep</code>	Not applicable	The parameter required to initialize the algorithm. Can be: <ul style="list-style-type: none"> • <code>step1Local</code> - the first step, performed on local nodes

Parameter	Default Value	Description
		<ul style="list-style-type: none"> <code>step2Master</code> - the second step, performed on a master node
<code>algorithmFPType</code>	<code>double</code>	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<code>method</code>	<code>defaultDense</code>	<p>Available methods for computation of correlation and variance-covariance matrices:</p> <ul style="list-style-type: none"> <code>defaultDense</code> - default performance-oriented method <code>singlePassDense</code> - implementation of the single-pass algorithm proposed by D.H.D. West <code>sumDense</code> - implementation of the algorithm in the cases where the basic statistics associated with the numeric table are pre-computed sums; returns an error if pre-computed sums are not defined <code>fastCSR</code> - performance-oriented method for CSR numeric tables <code>singlePassCSR</code> - implementation of the single-pass algorithm proposed by D.H.D. West; optimized for CSR numeric tables <code>sumCSR</code> - implementation of the algorithm in the cases where the basic statistics associated with the numeric table are pre-computed sums; optimized for CSR numeric tables; returns an error if pre-computed sums are not defined
<code>outputMatrixType</code>	<code>covarianceMatrix</code>	<p>The type of the output matrix. Can be:</p> <ul style="list-style-type: none"> <code>covarianceMatrix</code> - variance-covariance matrix <code>correlationMatrix</code> - correlation matrix

Computation of correlation and variance-covariance matrices follows the general schema described in [Algorithms](#):

Step 1 - on Local Nodes

In this step, the correlation and variance-covariance matrices algorithm accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
<code>data</code>	Pointer to the $n_i \times p$ numeric table that represents the i -th data block on the local node.

Input ID	Input
	While the input for <code>defaultDense</code> , <code>singlePassDense</code> , or <code>sumDense</code> method can be an object of any class derived from <code>NumericTable</code> , the input for <code>fastCSR</code> , <code>singlePassCSR</code> , or <code>sumCSR</code> method can only be an object of the <code>CSRNumericTable</code> class.

In this step, the correlation and variance-covariance matrices algorithm calculates the results described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
<code>nObservations</code>	Pointer 1 x 1 numeric table that contains the number of observations processed so far on the local node. By default, this result is an object of the <code>HomogenNumericTable</code> class, but you can define the result as an object of any class derived from <code>NumericTable</code> except <code>CSRNumericTable</code> .
<code>crossProduct</code>	Pointer $p \times p$ numeric table with the cross-product matrix computed so far on the local node. By default, this table is an object of the <code>HomogenNumericTable</code> class, but you can define the result as an object of any class derived from <code>NumericTable</code> except <code>CSRNumericTable</code> .
<code>sum</code>	Pointer 1 x p numeric table with partial sums computed so far on the local node. By default, this table is an object of the <code>HomogenNumericTable</code> class, but you can define the result as an object of any class derived from <code>NumericTable</code> except <code>PackedSymmetricMatrix</code> , <code>PackedTriangularMatrix</code> , and <code>CSRNumericTable</code> .

Step 2 - on Master Node

In this step, the correlation and variance-covariance matrices algorithm accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
<code>partialResults</code>	A collection that contains results computed in Step 1 on local nodes (<code>nObservations</code> , <code>crossProduct</code> , and <code>sum</code>). The collection can contain objects of any class derived from the <code>NumericTable</code> class except <code>PackedSymmetricMatrix</code> and <code>PackedTriangularMatrix</code> .

In this step, the correlation and variance-covariance matrices algorithm calculates the results described in the following table. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
<code>covariance</code>	Use when <code>outputMatrixType=covarianceMatrix</code> . Pointer to the numeric table with the $p \times p$ variance-covariance matrix. By default, this result is an object of the <code>HomogenNumericTable</code>

Result ID	Result
	class, but you can define the result as an object of any class derived from <code>NumericTable</code> except <code>PackedTriangularMatrix</code> and <code>CSRNumericTable</code> .
correlation	Use when <code>outputMatrixType=correlationMatrix</code> . Pointer to the numeric table with the $p \times p$ correlation matrix. By default, this result is an object of the <code>HomogenNumericTable</code> class, but you can define the result as an object of any class derived from <code>NumericTable</code> except <code>PackedTriangularMatrix</code> and <code>CSRNumericTable</code> .
mean	Pointer to the $1 \times p$ numeric table with means. By default, this result is an object of the <code>HomogenNumericTable</code> class, but you can define the result as an object of any class derived from <code>NumericTable</code> except <code>PackedTriangularMatrix</code> , <code>PackedSymmetricMatrix</code> , and <code>CSRNumericTable</code> .

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Examples

Refer to the following examples in your Intel DAAL directory:

C++:

- `./examples/cpp/source/covariance/covariance_dense_distributed.cpp`
- `./examples/cpp/source/covariance/covariance_csr_distributed.cpp`

Java*:

- `./examples/java/source/com/intel/daal/examples/covariance/CovarianceDenseDistributed.java`
- `./examples/java/source/com/intel/daal/examples/covariance/CovarianceCSR Distributed.java`

Python*:

- `./examples/python/source/covariance/covariance_dense_distributed.py`
- `./examples/python/source/covariance/covariance_csr_distributed.py`

See Also

[Handling Errors](#)

Performance Considerations

To get the best overall performance when computing correlation or variance-covariance matrices:

- If input data is homogeneous, provide the input data and store results in homogeneous numeric tables of the same type as specified in the `algorithmFPType` class template parameter.
- If input data is non-homogeneous, use AOS layout rather than SOA layout.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Cosine Distance Matrix

Cosine distances are a useful measure of similarities between feature vectors.

Details

Given n feature vectors $x_1 = (x_{11}, \dots, x_{1p})$, ..., $x_n = (x_{n1}, \dots, x_{np})$ of dimension p , the problem is to compute the symmetric n -by- n matrix D_{\cos} of distances between feature vectors:

$D_{\cos} = (d_{ij})$, where

$$d_{ij} = 1 - \frac{\sum_{k=1}^p x_{ik} \cdot x_{jk}}{\sqrt{\sum_{k=1}^p x_{ik}^2} \sqrt{\sum_{k=1}^p x_{jk}^2}}, \quad i = \overline{1, n}, j = \overline{1, n}$$

Batch Processing

Algorithm Input

The cosine distance matrix algorithm accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
data	Pointer to the $n \times p$ numeric table for which the distance is computed. The input can be an object of any class derived from <code>NumericTable</code> .

Algorithm Parameters

The cosine distance matrix algorithm has the following parameters:

Parameter	Default Value	Description
<code>algorithmFPType</code>	double	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<code>method</code>	defaultDense	Performance-oriented computation method, the only method supported by the algorithm.

Algorithm Output

The cosine distance matrix algorithm calculates the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
cosineDistance	Pointer to the numeric table that represents the $n \times n$ symmetric distance matrix D_{cos} . By default, the result is an object of the <code>PackedSymmetricMatrix</code> class with the <code>lowerPackedSymmetricMatrix</code> layout. However, you can define the result as an object of any class derived from <code>NumericTable</code> except <code>PackedTriangularMatrix</code> and <code>CSRNumericTable</code> .

Examples

Refer to the following examples in your Intel DAAL directory:

C++: `./examples/cpp/source/distance/cosine_distance_batch.cpp`

Java*: `./examples/java/source/com/intel/daal/examples/distance/CosineDistanceBatch.java`

Python*: `./examples/python/source/distance/cosine_distance_batch.py`

Performance Considerations

To get the best overall performance when computing the cosine distance matrix:

- If input data is homogeneous, provide the input data and store results in homogeneous numeric tables of the same type as specified in the `algorithmFPType` class template parameter.
- If input data is non-homogeneous, use AOS layout rather than SOA layout.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Correlation Distance Matrix

Correlation distances are a useful measure of similarities between feature vectors.

Details

Given n feature vectors $x_1 = (x_{11}, \dots, x_{1p})$, ..., $x_n = (x_{n1}, \dots, x_{np})$ of dimension p , the problem is to compute the symmetric n -by- n matrix D_{cor} of distances between feature vectors:

$D_{cor} = (d_{ij})$, where

$$d_{ij} = 1 - \frac{\sum_{k=1}^p (x_{ik} - \bar{x}_i)(x_{jk} - \bar{x}_j)}{\sqrt{\sum_{k=1}^p (x_{ik} - \bar{x}_i)^2} \sqrt{\sum_{k=1}^p (x_{jk} - \bar{x}_j)^2}},$$

$$\bar{x}_i = \frac{1}{p} \sum_{k=1}^p x_{ik}, \bar{x}_j = \frac{1}{p} \sum_{k=1}^p x_{jk}, i = \overline{1, n}, j = \overline{1, n}$$

Batch Processing

Algorithm Input

The correlation distance matrix algorithm accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
<code>data</code>	Pointer to the $n \times p$ numeric table for which the distance is computed. The input can be an object of any class derived from <code>NumericTable</code> .

Algorithm Parameters

The correlation distance matrix algorithm has the following parameters:

Parameter	Default Value	Description
<code>algorithmFPType</code>	<code>double</code>	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<code>method</code>	<code>defaultDense</code>	Performance-oriented computation method, the only method supported by the algorithm.

Algorithm Output

The correlation distance matrix algorithm calculates the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
<code>correlationDistance</code>	Pointer to the numeric table that represents the $n \times n$ symmetric distance matrix D_{cor} . By default, the result is an object of the <code>PackedSymmetricMatrix</code> class with the <code>lowerPackedSymmetricMatrix</code> layout. However, you can define the result as an object of any class derived from <code>NumericTable</code> except <code>PackedTriangularMatrix</code> and <code>CSRNumericTable</code> .

Examples

Refer to the following examples in your Intel DAAL directory:

C++: `./examples/cpp/source/distance/correlation_distance_batch.cpp`

Java*: `./examples/java/source/com/intel/daal/examples/distance/CorrelationDistanceBatch.java`

Python*: `./examples/python/source/distance/correlation_distance_batch.py`

Performance Considerations

To get the best overall performance when computing the correlation distance matrix:

- If input data is homogeneous, provide the input data and store results in homogeneous numeric tables of the same type as specified in the `algorithmFPType` class template parameter.
- If input data is non-homogeneous, use AOS layout rather than SOA layout.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

K-Means Clustering

K-Means is among the most popular and simplest clustering methods. It is intended to partition a data set into a small number of clusters such that feature vectors within a cluster have greater similarity with one another than with feature vectors from other clusters. Each cluster is characterized by a representative point, called a centroid, and a cluster radius.

In other words, the clustering methods enable reducing the problem of analysis of the entire data set to the analysis of clusters.

There are numerous ways to define the measure of similarity and centroids. For K-Means, the centroid is defined as the mean of feature vectors within the cluster.

Details

Given the set $X = \{ x_1 = (x_{11}, \dots, x_{1p}), \dots, x_n = (x_{n1}, \dots, x_{np}) \}$ of np -dimensional feature vectors and a positive integer k , the problem is to find a set $C = \{ c_1, \dots, c_k \}$ of kp -dimensional vectors that minimize the objective function (overall error)

$$\Phi_X(C) = \sum_{x_i \in X} d^2(x_i, C),$$

where $d^2(x_i, C)$ is the distance from x_i to the closest center in C , such as the Euclidean distance.

The vectors c_1, \dots, c_k are called centroids. To start computations, the algorithm requires initial values of centroids.

Centroid Initialization

Centroids initialization can be done using these methods:

- Choice of first k feature vectors from the data set X
- Random choice of k feature vectors from the data set using the following simple random sampling draw-by-draw algorithm. The algorithm does the following:
 1. Chooses one of the feature vectors x_i from X with equal probability
 2. Excludes x_i from X and adds it to the current set of centers
 3. Resumes from step 1 until the set of centers reaches the desired size k
- K-Means++ algorithm [Arthur2007], which selects centers with the probability proportional to their contribution to the overall error $\Phi_X(C)$ according to the following scheme:
 1. Chooses one of the feature vectors x_i from X with equal probability
 2. Excludes x_i from X and adds it to the current set of centers C
 3. For each feature vector x_j in X calculates its minimal distance $d(x_j, C)$ from the current set of centers C
 4. Chooses one of the feature vectors x_i from X with the probability $\frac{d^2(x_i, C)}{\Phi_X(C)}$

5. Resumes from step 2 until the set of centers C reaches the desired size k .
- Parallel K-Means++ algorithm [Bahmani2012] that does the following:
 1. Chooses one of the feature vectors x_i from X with equal probability
 2. Excludes x_i from X and adds it to the current set of centers C
 3. Repeats $nRounds$ times:
 - i. For each feature vector x_i from X calculates its minimal distance $d(x_i, C)$ from the current set of centers C
 - ii. Chooses $L = oversamplingFactor * k$ feature vectors x_i from X with the probability $\frac{d^2(x_i, C)}{\Phi_X(C)}$
 - iii. Excludes x_i vectors chosen in the previous step from X and adds them to the current set of centers C
 4. For $c_i \in C$ sets w_i to the ratings, the number of points in X closer to c_i than to any other point in C
 5. Applies K-Means++ algorithm with weights w_i to the points in C , which means that the following probability is used in step 4:

$$\frac{w_i d^2(x_i, C)}{\sum_{x_i \in X} w_i d^2(x_i, C)}$$

The algorithm parameters define the number of candidates L selected in each round and number of rounds:

- Choose *oversamplingFactor* to make $L = O(k)$.
- Choose $nRounds$ as $O(\log \Phi_X(C))$, where $\Phi_X(C)$ is the estimation of the goal function when the first center is chosen. [Bahmani2012] recommends to set $nRounds$ to a constant value not greater than 8.

Computation

Computation of the goal function includes computation of the Euclidean distance between vectors $||x_j - m_i||$. The algorithm uses the following modification of the Euclidean distance between feature vectors a and b : $d(a, b) = d_1(a, b) + d_2(a, b)$, where d_1 is computed for continuous features as

$$d_1(a, b) = \sqrt{\sum_{k=1}^{p_1} (a_k - b_k)^2}$$

and d_2 is computed for binary categorical features as

$$d_2(a, b) = \gamma \sqrt{\sum_{k=1}^{p_2} (a_k - b_k)^2}$$

In these equations, γ weighs the impact of binary categorical features on the clustering, p_1 is the number of continuous features, and p_2 is the number of binary categorical features. Note that the algorithm does not support non-binary categorical features.

The K-Means clustering algorithm computes centroids using Lloyd's method [Lloyd82]. For each feature vector x_1, \dots, x_n , you can also compute the index of the cluster that contains the feature vector.

Initialization

The K-Means clustering algorithm requires initialization of centroids as an explicit step. Initialization flow depends by the computation mode. Skip this step if you already calculated initial centroids.

Batch Processing

Input

Centroid initialization for K-Means clustering accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
<code>data</code>	Pointer to the $n \times p$ numeric table with the data to be clustered. The input can be an object of any class derived from <code>NumericTable</code> .

Parameters

The following table lists parameters of centroid initialization for K-Means clustering, which depend on the initialization method parameter *method*.

Parameter	<i>method</i>	Default Value	Description
<code>algorithmFPType</code>	any	double	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<code>method</code>	Not applicable		<p>Available initialization methods for K-Means clustering:</p> <ul style="list-style-type: none"> <code>defaultDense</code> - uses first <i>nClusters</i> points as initial clusters <code>deterministicCSR</code> - uses first <i>nClusters</i> points as initial clusters for data in a CSR numeric table <code>randomDense</code> - uses random <i>nClusters</i> points as initial clusters <code>randomCSR</code> - uses random <i>nClusters</i> points as initial clusters for data in a CSR numeric table <code>defaultDense</code> <ul style="list-style-type: none"> <code>plusPlusDense</code> - uses K-Means++ algorithm [Arthur2007] <code>plusPlusCSR</code> - uses K-Means++ algorithm for data in a CSR numeric table <code>parallelPlusDense</code> - uses parallel K-Means++ algorithm [Bahmani2012] <code>parallelPlusCSR</code> - uses parallel K-Means++ algorithm for data in a CSR numeric table <p>For more details, see the algorithm description.</p>
<code>nClusters</code>	any	Not applicable	The number of clusters. Required.
<code>seed</code>	any	777	The seed for generating random numbers.

Parameter	method	Default Value	Description
<i>oversamplingFactor</i>	<code>parallelPlusDense,</code> <code>parallelPlusCSR</code>	0.5	A fraction of <i>nClusters</i> in each of <i>nRounds</i> of parallel K-Means++. $L = nClusters * oversamplingFactor$ points are sampled in a round. For details, see [Bahmani2012], section 3.3.
<i>nRounds</i>	<code>parallelPlusDense,</code> <code>parallelPlusCSR</code>	5	The number of rounds for parallel K-Means++. ($L * nRounds$) must be greater than <i>nClusters</i> . For details, see [Bahmani2012], section 3.3.

Output

Centroid initialization for K-Means clustering calculates the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
<code>centroids</code>	Pointer to the <i>nClusters</i> x <i>p</i> numeric table with the cluster centroids. By default, this result is an object of the <code>HomogenNumericTable</code> class, but you can define the result as an object of any class derived from <code>NumericTable</code> except <code>PackedTriangularMatrix</code> , <code>PackedSymmetricMatrix</code> , and <code>CSRNumericTable</code> .

Distributed Processing

This mode assumes that the data set is split into *nblocks* blocks across computation nodes.

Parameters

Centroid initialization for K-Means clustering in the distributed processing mode has the following parameters:

Parameter	method	Default Value	Description
<i>computeStep</i>	any	Not applicable	The parameter required to initialize the algorithm. Can be: <ul style="list-style-type: none"> <code>step1Local</code> - the first step, performed on local nodes. Applicable for all methods. <code>step2Master</code> - the second step, performed on a master node. Applicable for <code>deterministic</code> and <code>random</code> methods only. <code>step2Local</code> - the second step, performed on local nodes. Applicable for <code>plusPlus</code> and <code>parallelPlus</code> methods only.

Parameter	<i>method</i>	Default Value	Description
			<ul style="list-style-type: none"> • <code>step3Master</code> - the third step, performed on a master node. Applicable for <code>plusPlus</code> and <code>parallelPlus</code> methods only. • <code>step4Local</code> - the forth step, performed on local nodes. Applicable for <code>plusPlus</code> and <code>parallelPlus</code> methods only. • <code>step5Master</code> - the fifth step, performed on a master node. Applicable for <code>plusPlus</code> and <code>parallelPlus</code> methods only.
<i>algorithmFPType</i>	<code>any</code>	<code>double</code>	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<i>method</i>	Not applicable	<code>defaultDense</code>	<p>Available initialization methods for K-Means clustering:</p> <ul style="list-style-type: none"> • <code>defaultDense</code> - uses first <i>nClusters</i> feature vectors as initial centroids • <code>deterministicCSR</code> - uses first <i>nClusters</i> feature vectors as initial centroids for data in a CSR numeric table • <code>randomDense</code> - uses random <i>nClusters</i> feature vectors as initial centroids • <code>randomCSR</code> - uses random <i>nClusters</i> feature vectors as initial centroids for data in a CSR numeric table • <code>plusPlusDense</code> - uses K-Means++ algorithm [Arthur2007] • <code>plusPlusCSR</code> - uses K-Means++ algorithm for data in a CSR numeric table • <code>parallelPlusDense</code> - uses parallel K-Means++ algorithm [Bahmani2012] • <code>parallelPlusCSR</code> - uses parallel K-Means++ algorithm for data in a CSR numeric table <p>For more details, see the algorithm description.</p>

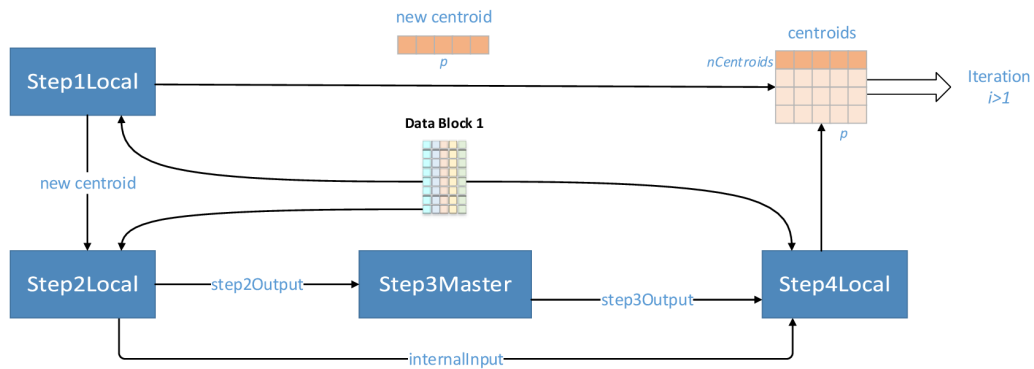
Parameter	method	Default Value	Description
<i>nClusters</i>	any	Not applicable	The number of centroids. Required.
<i>nRowsTotal</i>	any	0	The total number of rows in all input data sets on all nodes. Required in the distributed processing mode in the first step.
<i>seed</i>	any	777	The seed for generating random numbers.
<i>offset</i>		Not applicable	Offset in the total data set specifying the start of a block stored on a given local node. Required.
<i>oversamplingFactor</i>	parallelPlusDense, parallelPlusCSR	0.5	A fraction of <i>nClusters</i> in each of <i>nRounds</i> of parallel K-Means++. $L = nClusters * oversamplingFactor$ points are sampled in a round. For details, see [Bahmani2012], section 3.3.
<i>nRounds</i>	parallelPlusDense, parallelPlusCSR	5	The number of rounds for parallel K-Means++. ($L * nRounds$) must be greater than <i>nClusters</i> . For details, see [Bahmani2012], section 3.3.
<i>firstIteration</i>	plusPlusDense, plusPlusCSR, parallelPlusDense, parallelPlusCSR	false	Set to true if step2Local is called for the first time.
<i>outputForStep5Required</i>	parallelPlusDense, parallelPlusCSR	false	Set to true if step4Local is called on the last iteration of the Step 2 - Step 4 loop.

Centroid initialization for K-Means clustering follows the general schema described in [Algorithms](#).

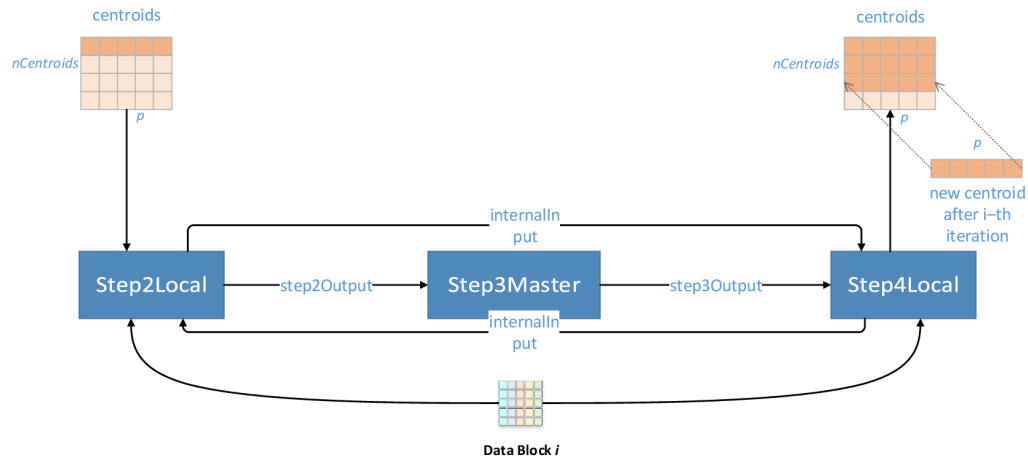
plusPlus methods

General scheme

Iteration 1



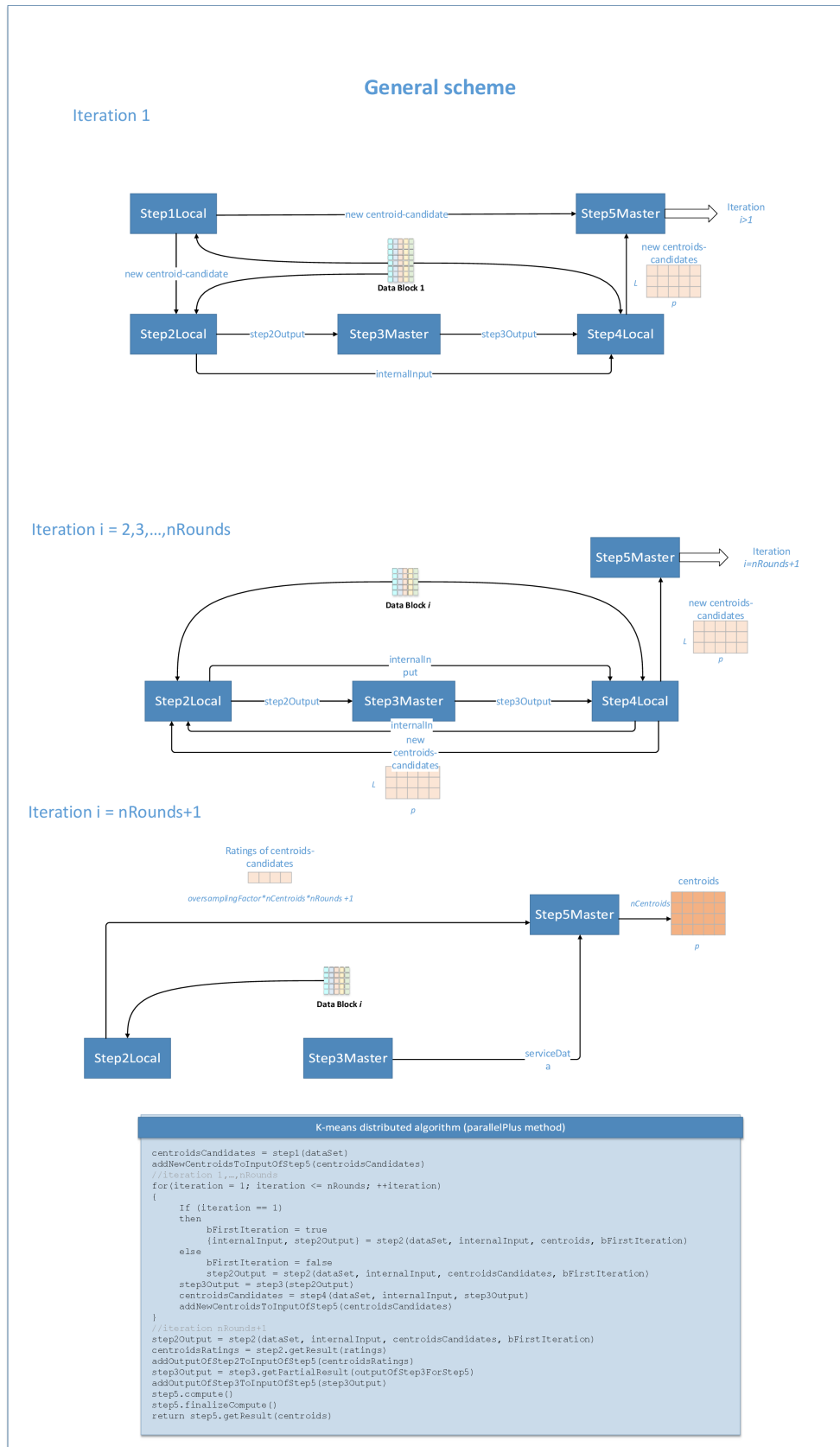
Iteration $i = 2, 3 \dots nCentroids$



K-means distributed initialization algorithm (plusPlus method)

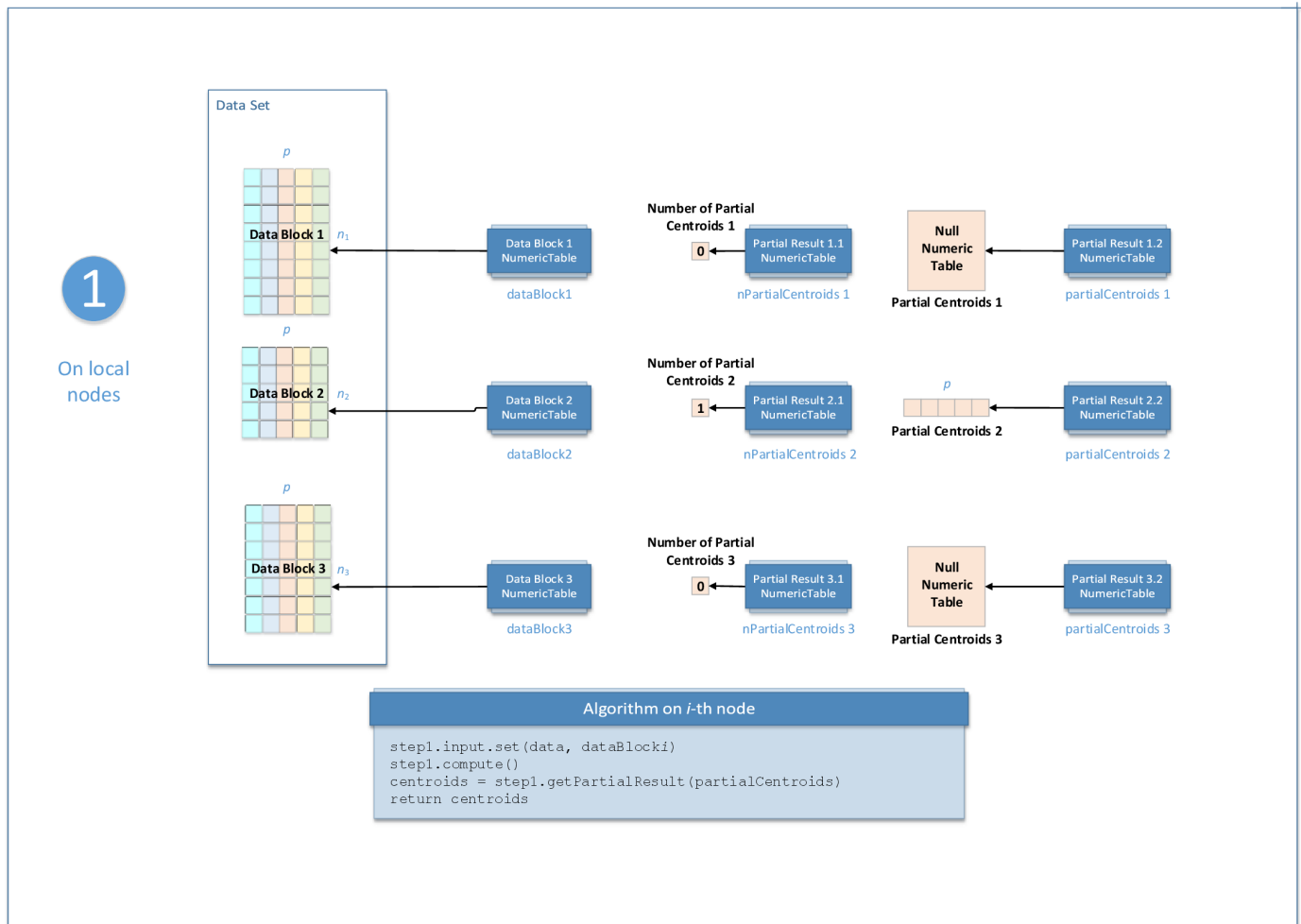
```
centroids = step1(dataSet)//iteration 1
outputCentroidsTable.addNewCentroids(centroids)
for(iteration = 2; iteration <= nCentroids; ++iteration)//iterations 2,3,...,nCentroids
{
    If (iteration == 2)
    then
        bFirstIteration = true
        {internalInput, step2Output} = step2(dataSet, centroids, bFirstIteration)
    else
        bFirstIteration = false
        step2Output = step2(dataSet, internalInput, centroids, bFirstIteration)
        step3Output = step3(step2Output)
        centroids = step4(dataSet, internalInput, step3Output)
        outputCentroidsTable.addNewCentroids(centroids)
}
return outputCentroidsTable
```

parallelPlus methods

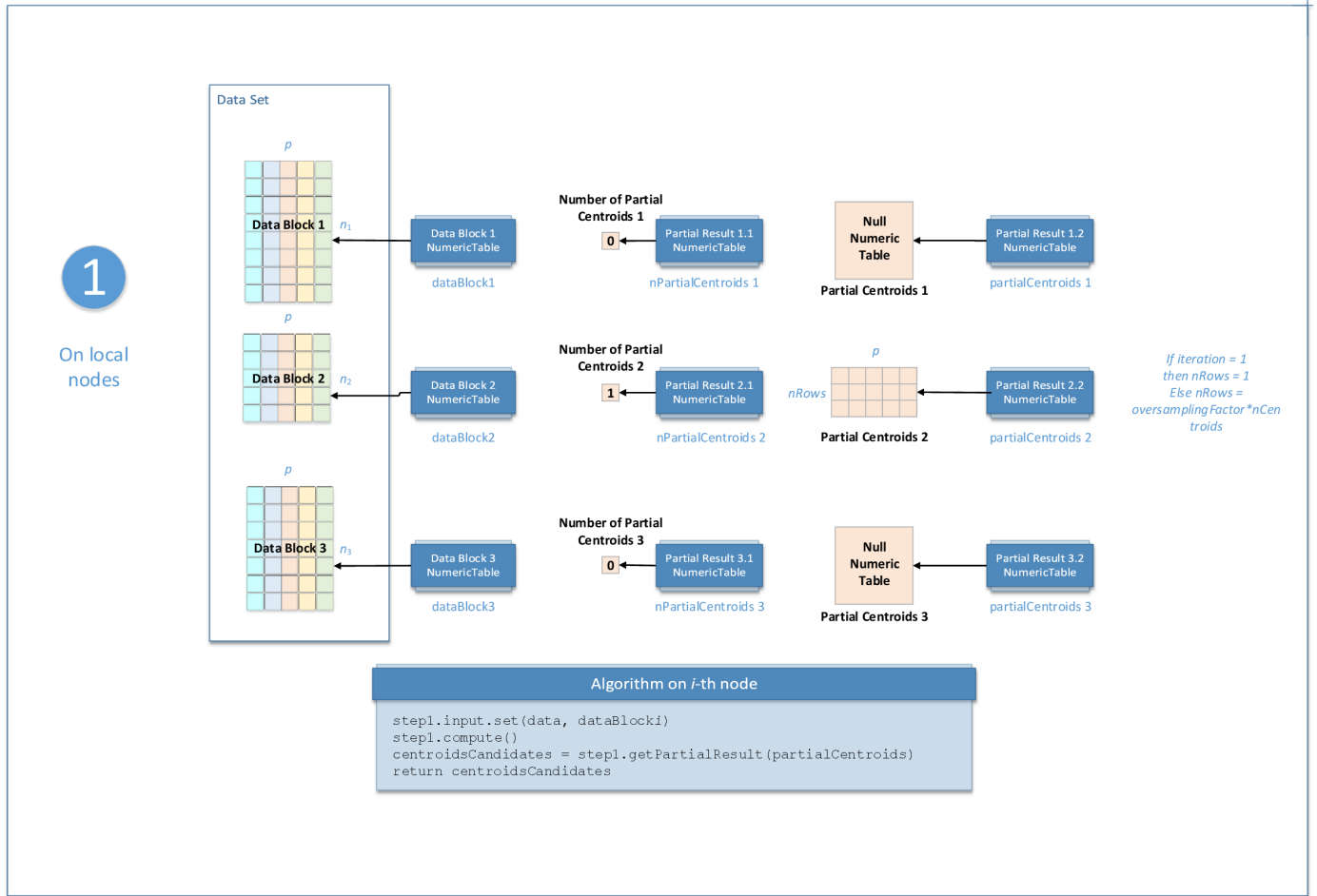


Step 1 - on Local Nodes (deterministic, random, plusPlus, and parallelPlus methods)

plusPlus methods



parallelPlus methods



In this step, centroid initialization for K-Means clustering accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
<code>data</code>	Pointer to the $n_i \times p$ numeric table that represents the i -th data block on the local node. While the input for <code>defaultDense</code> , <code>randomDense</code> , <code>plusplusDense</code> , or <code>parallelPlusDense</code> method can be an object of any class derived from <code>NumericTable</code> , the input for <code>deterministicCSR</code> , <code>randomCSR</code> , <code>plusplusCSR</code> , or <code>parallelPlusCSR</code> method can only be an object of the <code>CSRNumericTable</code> class.

In this step, centroid initialization for K-Means clustering calculates the results described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
<code>nPartialClusters</code>	Pointer to the 1×1 numeric table that contains the number of centroids computed on the local node. By default, this result is an object of the <code>HomogenNumericTable</code> class, but you can define the result as an object of any class derived from <code>NumericTable</code> except <code>CSRNumericTable</code> , <code>PackedTriangularMatrix</code> , and <code>PackedSymmetricMatrix</code> .

Result ID	Result
<code>partialClusters</code>	Pointer to the <code>nClusters</code> x <code>p</code> numeric table with the centroids computed on the local node. By default, this result is an object of the <code>HomogenNumericTable</code> class, but you can define the result as an object of any class derived from <code>NumericTable</code> except <code>PackedTriangularMatrix</code> , <code>PackedSymmetricMatrix</code> , and <code>CSRNumericTable</code> .

Step 2 - on Master Node (deterministic and random methods)

This step is applicable for `deterministic` and `random` methods only. Centroid initialization for K-Means clustering accepts the input from each local node described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

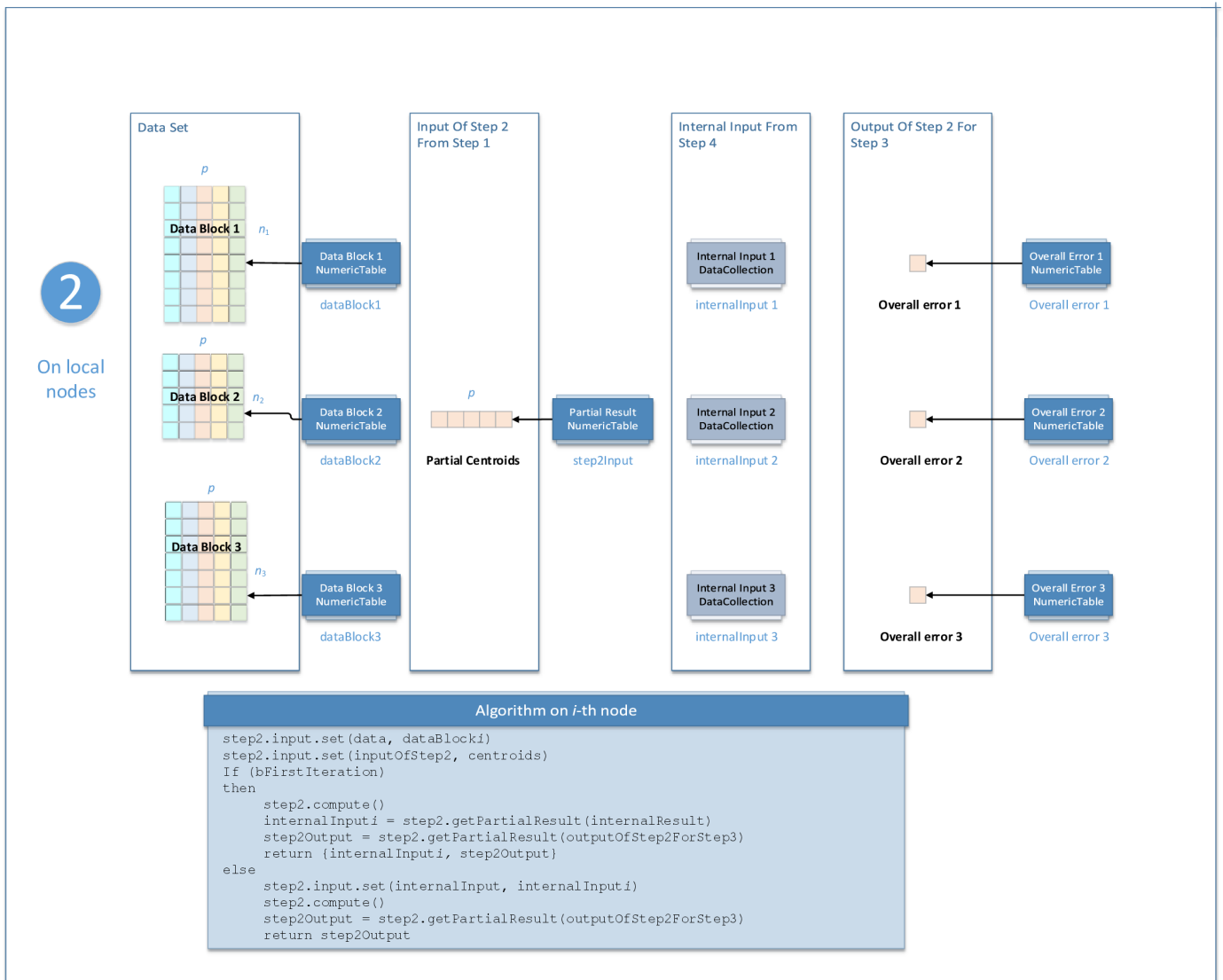
Input ID	Input
<code>partialResults</code>	A collection that contains results computed in Step 1 on local nodes (two numeric tables from each local node).

In this step, centroid initialization for K-Means clustering calculates the results described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

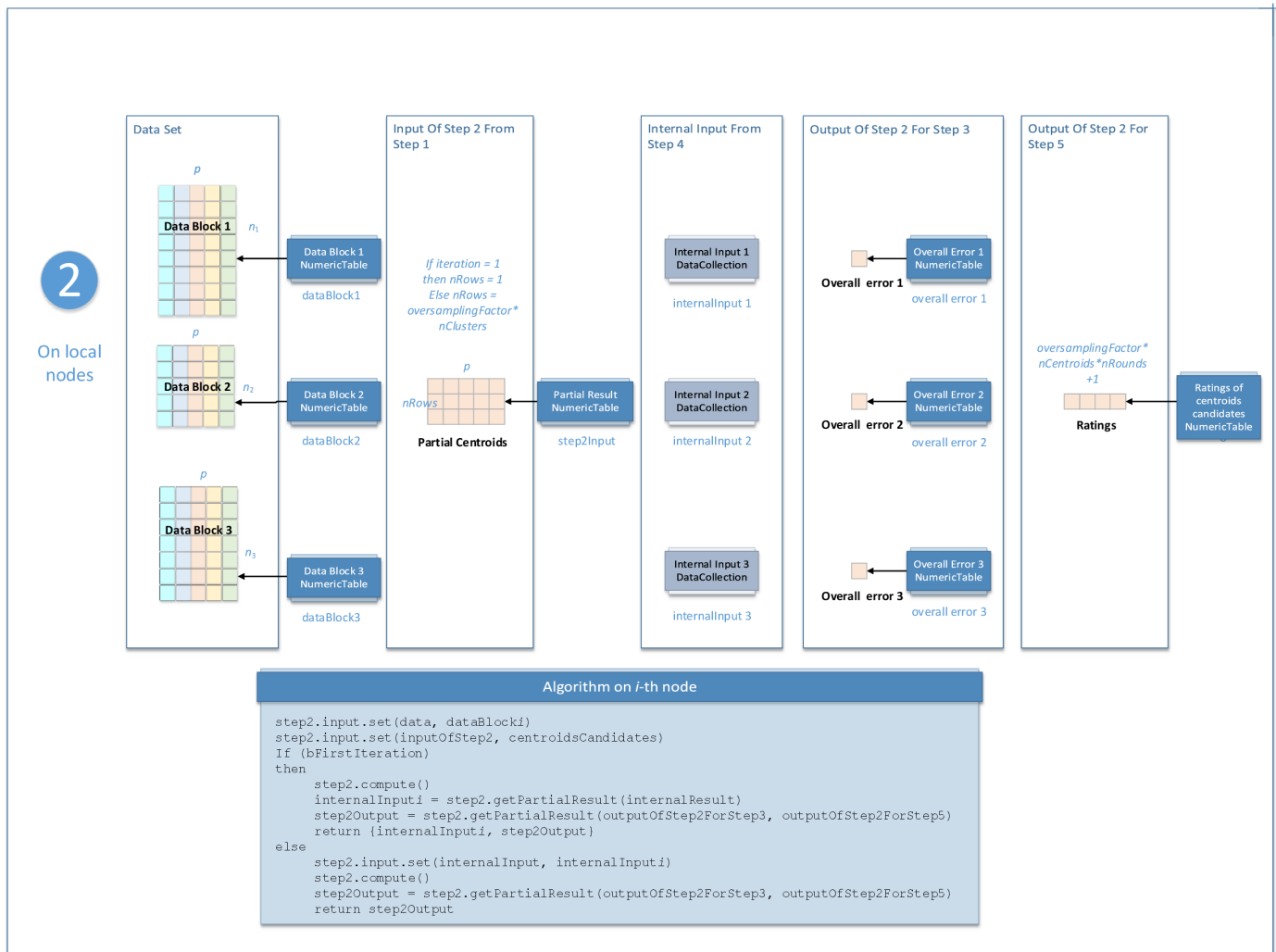
Result ID	Result
<code>centroids</code>	Pointer to the <code>nClusters</code> x <code>p</code> numeric table with centroids. By default, this result is an object of the <code>HomogenNumericTable</code> class, but you can define the result as an object of any class derived from <code>NumericTable</code> except <code>PackedTriangularMatrix</code> , <code>PackedSymmetricMatrix</code> , and <code>CSRNumericTable</code> .

Step 2 - on Local Nodes (`plusPlus` and `parallelPlus` methods)

plusPlus methods



parallelPlus methods



This step is applicable for `plusPlus` and `parallelPlus` methods only. Centroid initialization for K-Means clustering accepts the input from each local node described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
<code>data</code>	Pointer to the $n_i \times p$ numeric table that represents the i -th data block on the local node. While the input for <code>defaultDense</code> , <code>randomDense</code> , <code>plusPlusDense</code> , or <code>parallelPlusDense</code> method can be an object of any class derived from <code>NumericTable</code> , the input for <code>deterministicCSR</code> , <code>randomCSR</code> , <code>plusPlusCSR</code> , or <code>parallelPlusCSR</code> method can only be an object of the <code>CSRNumericTable</code> class.
<code>inputOfStep2</code>	<p>Pointer to the $m \times p$ numeric table with the centroids calculated in the previous steps (Step 1 or Step 4).</p> <p>The value of m is defined by the method and iteration of the algorithm:</p> <ul style="list-style-type: none"> <code>plusPlus</code> method: $m = 1$ <code>parallelPlus</code> method: <ul style="list-style-type: none"> $m = 1$ for the first iteration of the Step 2 - Step 4 loop

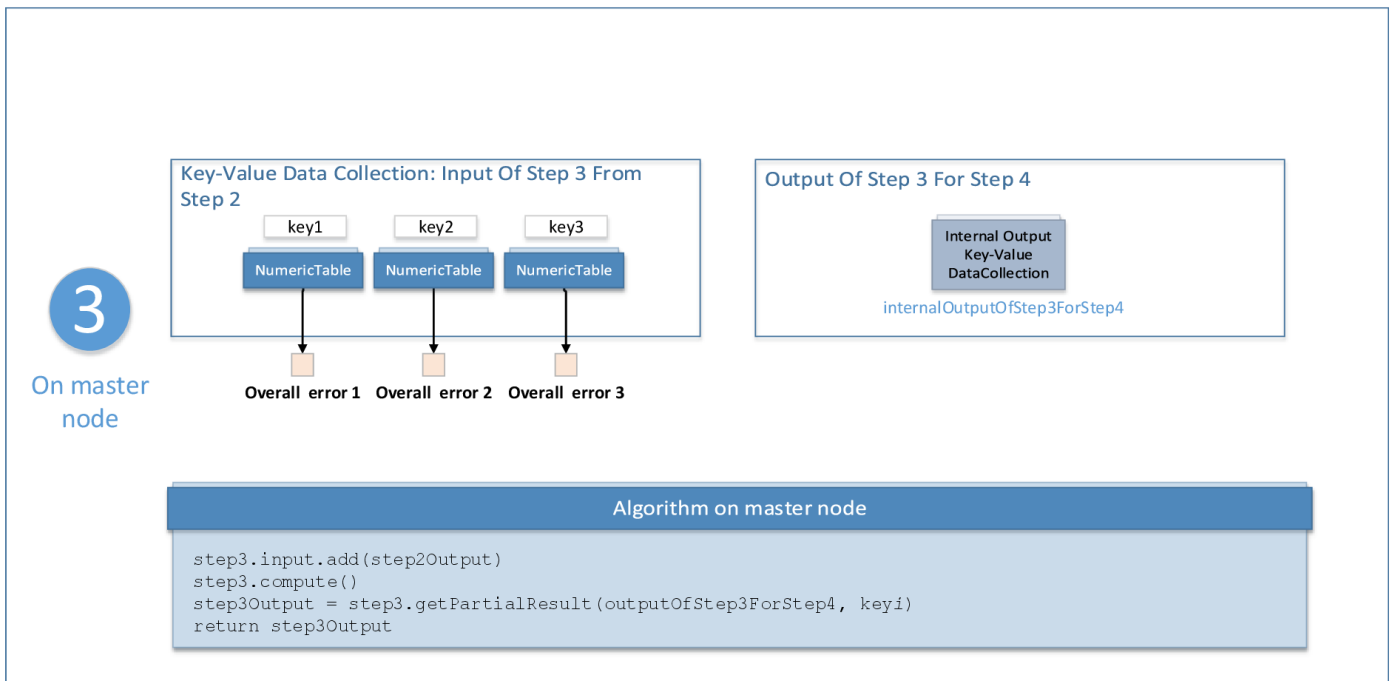
Input ID	Input
	<ul style="list-style-type: none"> $m = L = nClusters * oversamplingFactor$ for other iterations <p>This input can be an object of any class derived from <code>NumericTable</code>, except <code>CSRNumericTable</code>, <code>PackedTriangularMatrix</code>, and <code>PackedSymmetricMatrix</code>.</p>
<code>internalInput</code>	<p>Pointer to the <code>DataCollection</code> object with the internal data of the distributed algorithm used by its local nodes in Step 2 and Step 4. The <code>DataCollection</code> is created in Step 2 when <code>firstIteration</code> is set to <code>true</code>, and then the <code>DataCollection</code> should be set from the partial result as an input for next local steps (Step 2 and Step 4).</p>

In this step, centroid initialization for K-Means clustering calculates the results described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

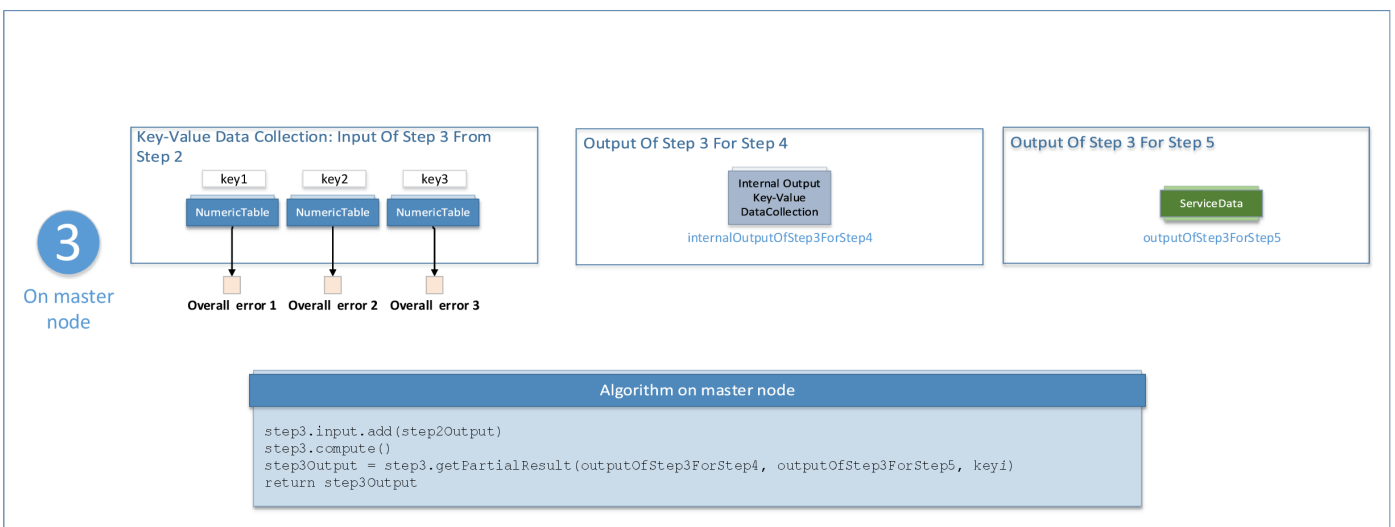
Result ID	Result
<code>outputOfStep2ForStep3</code>	<p>Pointer to the 1 x 1 numeric table that contains the overall error accumulated on the node. For a description of the overall error, see K-Means Clustering Details.</p> <p>By default, this result is an object of the <code>HomogenNumericTable</code> class, but you can define the result as an object of any class derived from <code>NumericTable</code> except <code>PackedTriangularMatrix</code>, <code>PackedSymmetricMatrix</code>, and <code>CSRNumericTable</code>.</p>
<code>outputOfStep2ForStep5</code>	<p>Applicable for <code>parallelPlus</code> methods only and calculated when <code>outputForStep5Required</code> is set to <code>true</code>. Pointer to the 1 x m numeric table with the ratings of centroid candidates computed on the previous steps and $m = oversamplingFactor * nClusters * nRounds + 1$. For a description of ratings, see K-Means Clustering Details.</p> <p>By default, this result is an object of the <code>HomogenNumericTable</code> class, but you can define the result as an object of any class derived from <code>NumericTable</code> except <code>PackedTriangularMatrix</code>, <code>PackedSymmetricMatrix</code>, and <code>CSRNumericTable</code>.</p>

Step 3 - on Master Node ([plusPlus](#) and [parallelPlus](#) methods)

[plusPlus](#) methods



parallelPlus methods



This step is applicable for `plusPlus` and `parallelPlus` methods only. Centroid initialization for K-Means clustering accepts the input from each local node described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

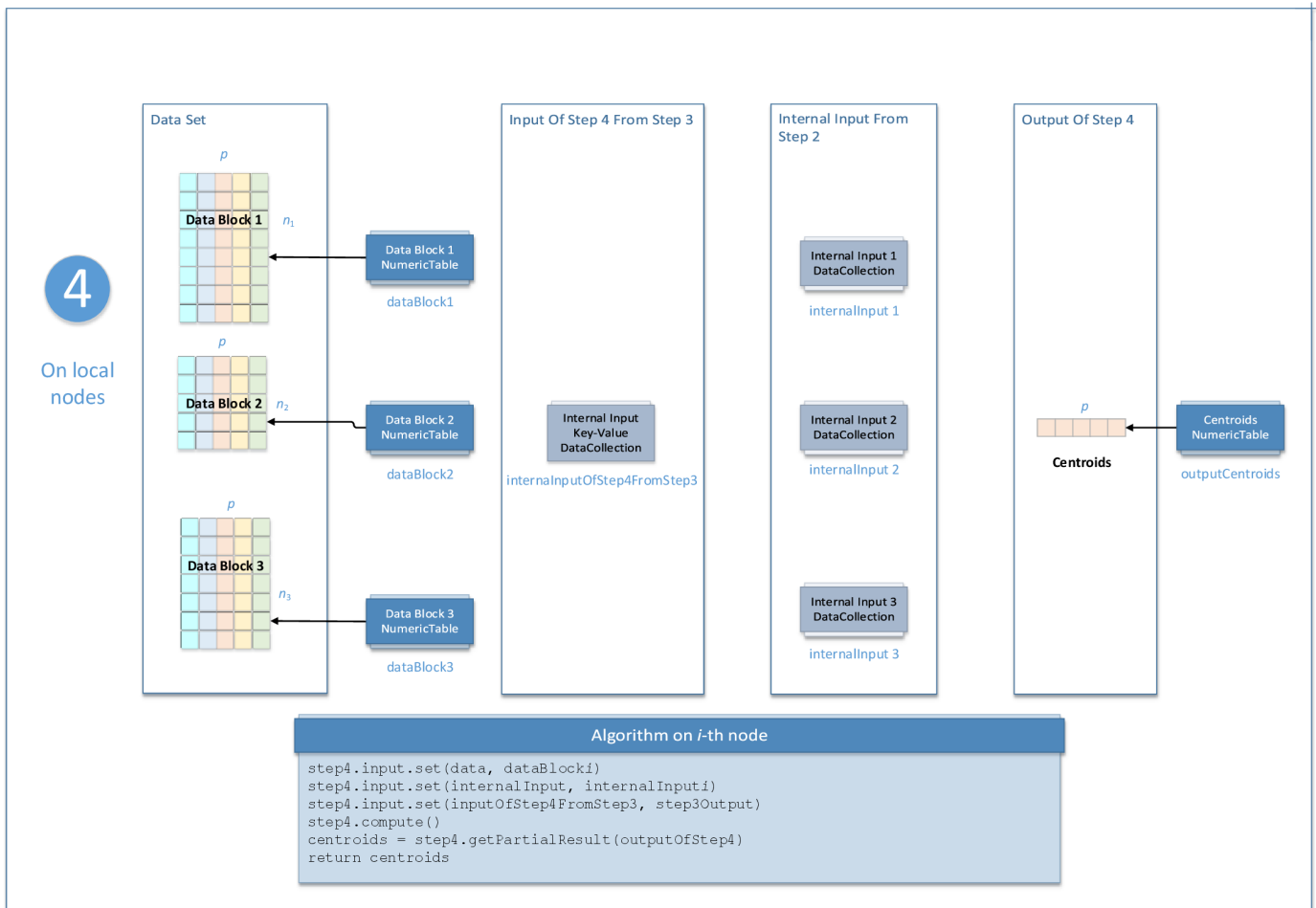
Input ID	Input
<code>inputOfStep3FromStep2</code>	A key-value data collection that maps parts of the accumulated error to the local nodes: i -th element of this collection is a numeric table that contains overall error accumulated on the i -th node.

In this step, centroid initialization for K-Means clustering calculates the results described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

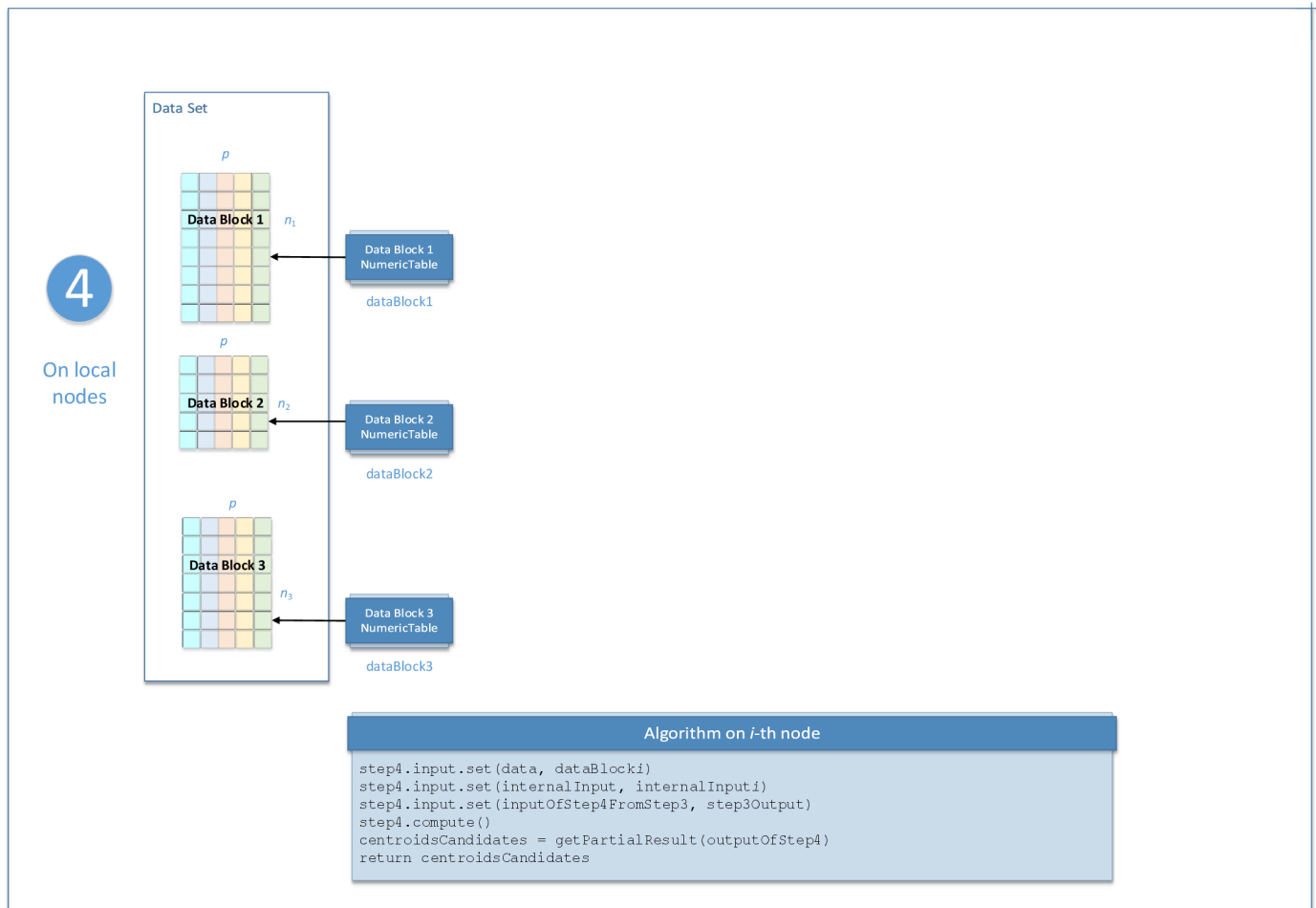
Result ID	Result
outputOfStep3ForStep4	A key-value data collection that maps the input from Step 4 to local nodes: i -th element of this collection is a numeric table that contains the input from Step 4 on the i -th node. Note that Step 3 may produce no input for Step 4 on some local nodes, which means the collection may not contain the i -th node entry. The single element of this numeric table $v \leq \Phi_X(C)$, where the overall error $\Phi_X(C)$ calculated on the node. For a description of the overall error, see K-Means Clustering Details . This value defines the probability to sample a new centroid on the i -th node.
outputOfStep3ForStep5	Applicable for <code>parallelPlus</code> methods only. Pointer to the service data to be used in Step 5 .

Step 4 - on Local Nodes (`plusPlus` and `parallelPlus` methods)

`plusPlus` methods



`parallelPlus` methods



This step is applicable for `plusPlus` and `parallelPlus` methods only. Centroid initialization for K-Means clustering accepts the input from each local node described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

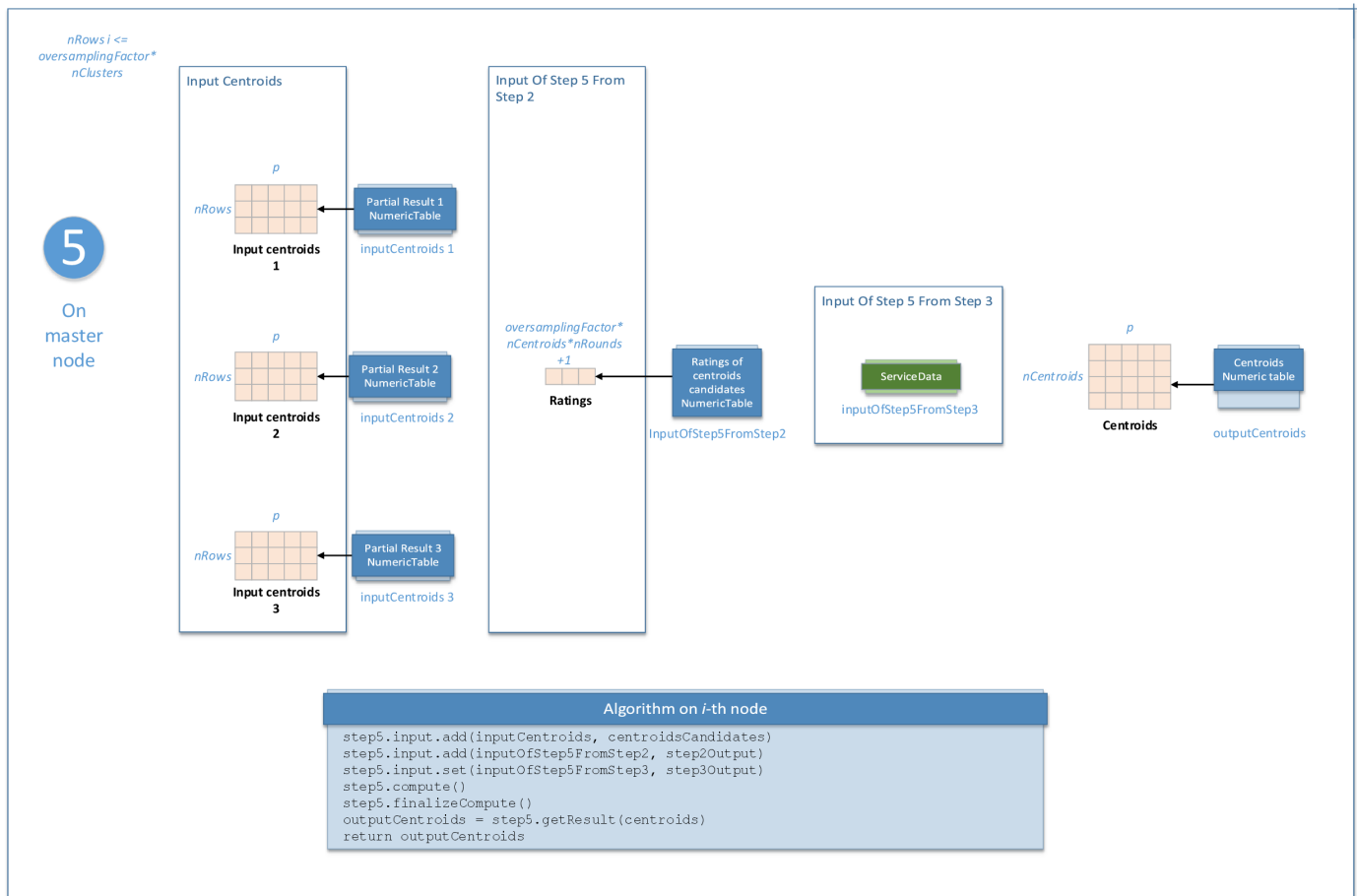
Input ID	Input
<code>data</code>	Pointer to the $n_i \times p$ numeric table that represents the i -th data block on the local node. While the input for <code>defaultDense</code> , <code>randomDense</code> , <code>plusPlusDense</code> , or <code>parallelPlusDense</code> method can be an object of any class derived from <code>NumericTable</code> , the input for <code>deterministicCSR</code> , <code>randomCSR</code> , <code>plusPlusCSR</code> , or <code>parallelPlusCSR</code> method can only be an object of the <code>CSRNumericTable</code> class.
<code>inputOfStep4FromStep3</code>	<p>Pointer to the $l \times m$ numeric table with the values calculated in Step 3.</p> <p>The value of m is defined by the method of the algorithm:</p> <ul style="list-style-type: none"> <code>plusPlus</code> method: $m = 1$ <code>parallelPlus</code> method: $m \leq L$, $L = nClusters * oversamplingFactor$

Input ID	Input
	This input can be an object of any class derived from <code>NumericTable</code> , except <code>CSRNumericTable</code> , <code>PackedTriangularMatrix</code> , and <code>PackedSymmetricMatrix</code> .
<code>internalInput</code>	Pointer to the <code>DataCollection</code> object with the internal data of the distributed algorithm used by its local nodes in Step 2 and Step 4 . The <code>DataCollection</code> is created in Step 2 when <code>firstIteration</code> is set to <code>true</code> , and then the <code>DataCollection</code> should be set from the partial result as the input for next local steps (Step 2 and Step 4).

In this step, centroid initialization for K-Means clustering calculates the results described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
<code>outputOfStep4</code>	Pointer to the $m \times p$ numeric table that contains centroids computed on this local node, where m equals to the one in <code>inputOfStep4FromStep3</code> . By default, this result is an object of the <code>HomogenNumericTable</code> class, but you can define the result as an object of any class derived from <code>NumericTable</code> except <code>CSRNumericTable</code> , <code>PackedTriangularMatrix</code> , and <code>PackedSymmetricMatrix</code> .

Step 5 - on Master Node ([parallelPlus](#) methods)



This step is applicable for `parallelPlus` methods only. Centroid initialization for K-Means clustering accepts the input from each local node described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
<code>inputCentroids</code>	<p>A data collection with the centroids calculated in Step 1 or Step 4. Each item in the collection is the pointer to $m \times p$ numeric table, where the value of m is defined by the method and the iteration of the algorithm:</p> <ul style="list-style-type: none"> <code>parallelPlus</code> method: <ul style="list-style-type: none"> $m = 1$ for the data added as the output of Step 1 $m \leq L$, $L = nClusters * oversamplingFactor$ for the data added as the output of Step 4 <p>Each numeric table can be an object of any class derived from <code>NumericTable</code>, except <code>CSRNumericTable</code>, <code>PackedTriangularMatrix</code>, and <code>PackedSymmetricMatrix</code>.</p>
<code>inputOfStep5FromStep2</code>	<p>A data collection with the items calculated in Step 2 on local nodes. For a detailed definition, see <code>outputOfStep2ForStep5</code> above.</p>

Input ID	Input
<code>inputOfStep5FromStep3</code>	Pointer to the service data generated as the output of Step 3 on master node. For a detailed definition, see <code>outputOfStep3ForStep5</code> above.

In this step, centroid initialization for K-Means clustering calculates the results described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
<code>centroids</code>	Pointer to the $nClusters \times p$ numeric table with the centroids. By default, this result is an object of the <code>HomogenNumericTable</code> class, but you can define the result as an object of any class derived from <code>NumericTable</code> except <code>PackedTriangularMatrix</code> , <code>PackedSymmetricMatrix</code> , and <code>CSRNumericTable</code> .

Computation

Batch Processing

Algorithm Input

The K-Means clustering algorithm accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
<code>data</code>	Pointer to the $n \times p$ numeric table with the data to be clustered. The input can be an object of any class derived from <code>NumericTable</code> .
<code>inputCentroids</code>	Pointer to the $nClusters \times p$ numeric table with the initial centroids. The input can be an object of any class derived from <code>NumericTable</code> .

Algorithm Parameters

The K-Means clustering algorithm has the following parameters:

Parameter	Default Value	Description
<code>algorithmFPTType</code>	<code>double</code>	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<code>method</code>	<code>defaultDense</code>	Available computation methods for K-Means clustering: <ul style="list-style-type: none"> <code>defaultDense</code> - implementation of Lloyd's algorithm <code>lloydCSR</code> - implementation of Lloyd's algorithm for CSR numeric tables

Parameter	Default Value	Description
<i>nClusters</i>	Not applicable	The number of clusters. Required to initialize the algorithm.
<i>maxIterations</i>	Not applicable	The number of iterations. Required to initialize the algorithm.
<i>accuracyThreshold</i>	0.0	The threshold for termination of the algorithm.
<i>gamma</i>	1.0	The weight to be used in distance calculation for binary categorical features.
<i>distanceType</i>	euclidean	The measure of closeness between points (observations) being clustered. The only distance type supported so far is the Euclidian distance.
<i>assignFlag</i>	true	A flag that enables computation of assignments, that is, assigning cluster indices to respective observations.

Algorithm Output

The K-Means clustering algorithm calculates the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
centroids	Pointer to the <i>nClusters</i> x <i>p</i> numeric table with the cluster centroids. By default, this result is an object of the <code>HomogenNumericTable</code> class, but you can define the result as an object of any class derived from <code>NumericTable</code> except <code>PackedTriangularMatrix</code> , <code>PackedSymmetricMatrix</code> , and <code>CSRNumericTable</code> .
assignments	Use when <i>assignFlag</i> =true. Pointer to the <i>n</i> x 1 numeric table with assignments of cluster indices to feature vectors in the input data. By default, this result is an object of the <code>HomogenNumericTable</code> class, but you can define the result as an object of any class derived from <code>NumericTable</code> except <code>PackedTriangularMatrix</code> , <code>PackedSymmetricMatrix</code> , and <code>CSRNumericTable</code> .
goalFunction	Pointer to the 1 x 1 numeric table with the value of the goal function. By default, this result is an object of the <code>HomogenNumericTable</code> class, but you can define the result as an object of any class derived from <code>NumericTable</code> except <code>CSRNumericTable</code> .
nIterations	Pointer to the 1 x 1 numeric table with the actual number of iterations done by the algorithm. By default, this result is an object of the <code>HomogenNumericTable</code> class, but you can define the result as an object of any class derived from <code>NumericTable</code> except <code>PackedTriangularMatrix</code> , <code>PackedSymmetricMatrix</code> , and <code>CSRNumericTable</code> .

NOTE

You can skip update of centroids and `goalFunction` in the result and compute assignments using original `inputCentroids`. To do this, set `assignFlag` to `true` and `maxIterations` to zero.

Examples

Refer to the following examples in your Intel DAAL directory:

C++:

- `./examples/cpp/source/kmeans/kmeans_dense_batch.cpp`
- `./examples/cpp/source/kmeans/kmeans_csr_batch.cpp`

Java*:

- `./examples/java/source/com/intel/daal/examples/kmeans/KMeansDenseBatch.java`
- `./examples/java/source/com/intel/daal/examples/kmeans/KMeansCSRBatch.java`

Python*:

- `./examples/python/source/kmeans/kmeans_dense_batch.py`
- `./examples/python/source/kmeans/kmeans_csr_batch.py`

Distributed Processing

This mode assumes that the data set is split into `nblocks` blocks across computation nodes.

Algorithm Parameters

The K-Means clustering algorithm in the distributed processing mode has the following parameters:

Parameter	Default Value	Description
<code>computeStep</code>	Not applicable	The parameter required to initialize the algorithm. Can be: <ul style="list-style-type: none"> • <code>step1Local</code> - the first step, performed on local nodes • <code>step2Master</code> - the second step, performed on a master node
<code>algorithmFPType</code>	<code>double</code>	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<code>method</code>	<code>defaultDense</code>	Available computation methods for K-Means clustering: <ul style="list-style-type: none"> • <code>defaultDense</code> - implementation of Lloyd's algorithm • <code>lloydCSR</code> - implementation of Lloyd's algorithm for CSR numeric tables
<code>nClusters</code>	Not applicable	The number of clusters. Required to initialize the algorithm.
<code>gamma</code>	1.0	The weight to be used in distance calculation for binary categorical features.

Parameter	Default Value	Description
<i>distanceType</i>	euclidean	The measure of closeness between points (observations) being clustered. The only distance type supported so far is the Euclidian distance.
<i>assignFlag</i>	false	A flag that enables computation of assignments, that is, assigning cluster indices to respective observations.

To compute K-Means clustering in the distributed processing mode, use the general schema described in [Algorithms](#) as follows:

Step 1 - on Local Nodes



In this step, the K-Means clustering algorithm accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
<code>data</code>	Pointer to the $n_i \times p$ numeric table that represents the i -th data block on the local node. The input can be an object of any class derived from <code>NumericTable</code> .
<code>inputCentroids</code>	Pointer to the $nClusters \times p$ numeric table with the initial cluster centroids. This input can be an object of any class derived from <code>NumericTable</code> .

In this step, the K-Means clustering algorithm calculates the partial results and results described below. Pass the Partial Result ID or Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

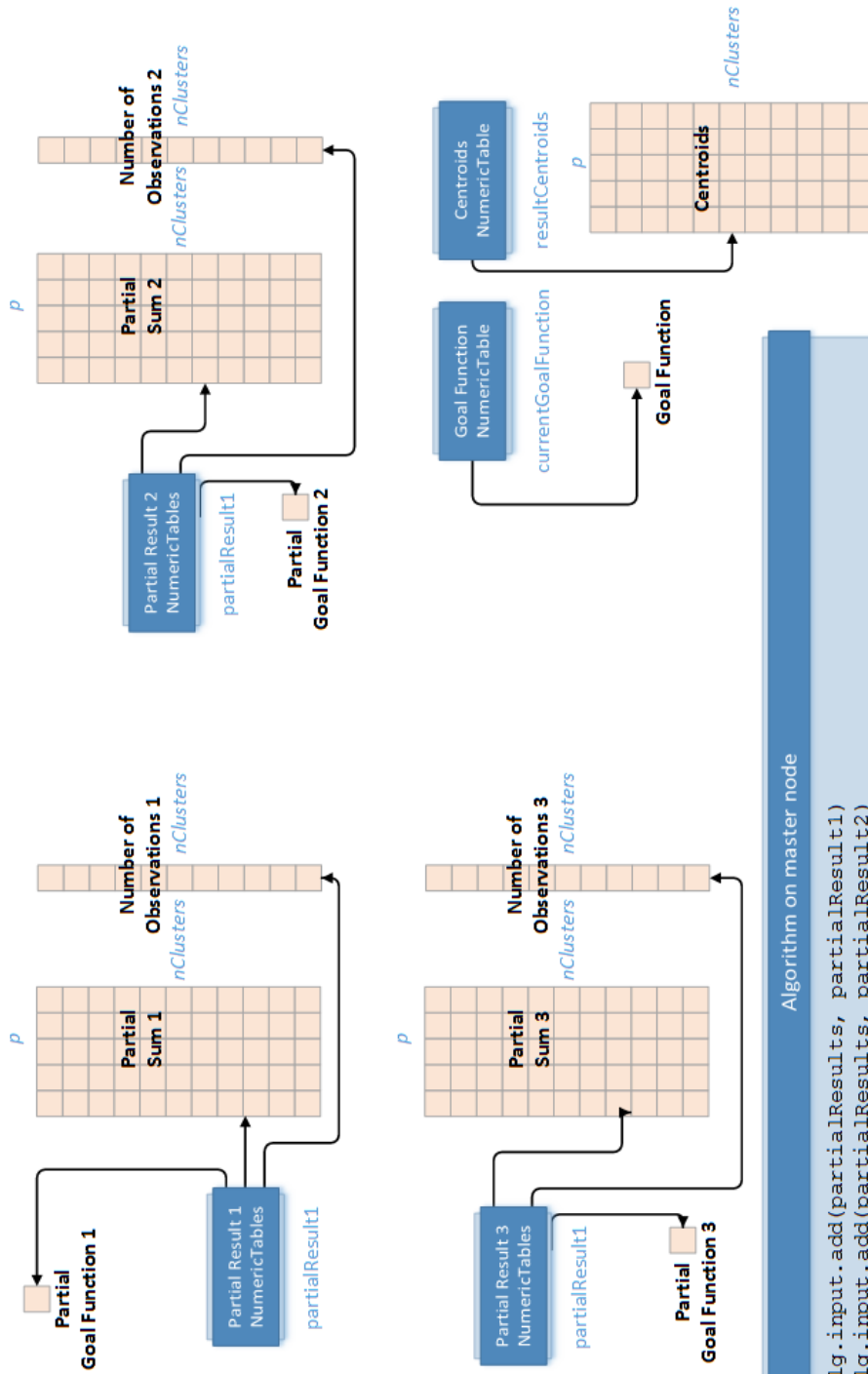
Partial Result ID	Result
<code>nObservations</code>	Pointer to the $nClusters \times 1$ numeric table that contains the number of observations assigned to the clusters on local node. By default, this result is an object of the <code>HomogenNumericTable</code> class, but you can define this result as an object of any class derived from <code>NumericTable</code> except <code>CSRNumericTable</code> .
<code>partialSums</code>	Pointer to the $nClusters \times p$ numeric table with partial sums of observations assigned to the clusters on the local node. By default, this result is an object of the <code>HomogenNumericTable</code> class, but you can define the result as an object of any class derived from <code>NumericTable</code> except <code>PackedTriangularMatrix</code> , <code>PackedSymmetricMatrix</code> , and <code>CSRNumericTable</code> .
<code>partialGoalFunction</code>	Pointer to the 1×1 numeric table that contains the value of the partial goal function for observations processed on the local node. By default, this result is an object of the <code>HomogenNumericTable</code> class, but you can define this result as an object of any class derived from <code>NumericTable</code> except <code>CSRNumericTable</code> .

Result ID	Result
<code>assignments</code>	Use when <code>assignFlag = true</code> . Pointer to the $n_i \times 1$ numeric table with 32-bit integer assignments of cluster indices to feature vectors in the input data on the local node. By default, this result is an object of the <code>HomogenNumericTable</code> class, but you can define this result as an object of any class derived from <code>NumericTable</code> except <code>PackedTriangularMatrix</code> , <code>PackedSymmetricMatrix</code> , and <code>CSRNumericTable</code> .

Step 2 - on Master Node

2

On master
node



Algorithm on master node

```

alg.input.add(partialResults, partialResult1)
alg.input.add(partialResults, partialResult2)
alg.input.add(partialResults, partialResult3)
alg.compute()
alg.finalizeCompute()
currentGoalFunction = alg.getResult(goalFunction)
If (iteration < nIterations && abs(currentGoalFunction-previousGoalFunction)
> accuracyThreshold )
then
    goto step1
else
    resultCentroids = alg.getResult(centroids)

```


In this step, the K-Means clustering algorithm accepts the input from each local node described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
<code>partialResults</code>	A collection that contains results computed in Step 1 on local nodes.

In this step, the K-Means clustering algorithm calculates the results described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
<code>centroids</code>	Pointer to the $nClusters \times p$ numeric table with the cluster centroids. By default, this result is an object of the <code>HomogenNumericTable</code> class, but you can define the result as an object of any class derived from <code>NumericTable</code> except <code>PackedTriangularMatrix</code> , <code>PackedSymmetricMatrix</code> , and <code>CSRNumericTable</code> .
<code>goalFunction</code>	Pointer to the 1×1 numeric table that contains the value of the goal function. By default, this result is an object of the <code>HomogenNumericTable</code> class, but you can define this result as an object of any class derived from <code>NumericTable</code> except <code>CSRNumericTable</code> .

Important

The algorithm computes assignments using input centroids. Therefore, to compute assignments using final computed centroids, after the last call to `Step2compute()` method on the master node, on each local node set `assignFlag` to true and do one additional call to `Step1compute()` and `finalizeCompute()` methods. Always set `assignFlag` to true and call `finalizeCompute()` to obtain assignments in each step.

NOTE

To compute assignments using original `inputCentroids` on the given node, you can use K-Means clustering algorithm in the batch processing mode with the subset of the data available on this node. See [Batch Processing](#) for more details.

Examples

Refer to the following examples in your Intel DAAL directory:

C++:

- `./examples/cpp/source/kmeans/kmeans_dense_distributed.cpp`
- `./examples/cpp/source/kmeans/kmeans_csr_distributed.cpp`

Java*:

- `./examples/java/source/com/intel/daal/examples/kmeans/KMeansDenseDistributed.java`
- `./examples/java/source/com/intel/daal/examples/kmeans/KMeansCSR Distributed.java`

Python*:

- `./examples/python/source/kmeans/kmeans_dense_distributed.py`
- `./examples/python/source/kmeans/kmeans_csr_distributed.py`

Performance Considerations

To get the best overall performance of the K-Means algorithm:

- If input data is homogeneous, provide the input data and store results in homogeneous numeric tables of the same type as specified in the `algorithmFType` class template parameter.
- If input data is non-homogeneous, use AOS layout rather than SOA layout.
- For the output `assignments` table, use a homogeneous numeric table of the `int` type.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Principal Component Analysis

Principal Component Analysis (PCA) is a method for exploratory data analysis. PCA transforms a set of observations of possibly correlated variables to a new set of uncorrelated variables, called principal components. Principal components are the directions of the largest variance, that is, the directions where the data is mostly spread out.

Because all principal components are orthogonal to each other, there is no redundant information. This is a way of replacing a group of variables with a smaller set of new variables. PCA is one of powerful techniques for dimension reduction.

Details

Given n feature vectors $x_1 = (x_{11}, \dots, x_{1p})$, ..., $x_n = (x_{n1}, \dots, x_{np})$ of dimension p or a $p \times p$ correlation matrix Cor , the problem is to compute the principal components for the data set. The library returns the transformation matrix T , which contains eigenvectors in the row-major order and a vector of corresponding eigenvalues. You can use the results to choose the new dimension $d < p$ and apply the transformation T_d : $x_i \rightarrow y_i$ to the original data set according to the rule $y_i = T_d x_i^T$, where the matrix T_d is the submatrix of T that contains d eigenvectors corresponding to the d largest eigenvalues.

You can provide these types of input data to the PCA algorithms of the library:

- Original, non-normalized data set
- Normalized data set, where each feature has the zero mean and unit variance
- Correlation matrix

Batch Processing

Algorithm Input

The PCA algorithm accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
<code>data</code>	Use when the input data is a normalized or non-normalized data set. Pointer to the $n \times p$ numeric table that contains the input data set. This input can be an object of any class derived from <code>NumericTable</code> .
<code>correlation</code>	Use when the input data is a correlation matrix. Pointer to the $p \times p$ numeric table that contains the correlation matrix. This input can be an object of any class derived from <code>NumericTable</code> except <code>PackedTriangularMatrix</code> .

Algorithm Parameters

The PCA algorithm has the following parameters, depending on the computation method parameter `method`:

Parameter	<code>method</code>	Default Value	Description
<code>algorithmFPType</code>	<code>defaultDense</code> or <code>svdDense</code>	<code>double</code>	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<code>method</code>	Not applicable	<code>defaultDense</code>	Available methods for PCA computation: <ul style="list-style-type: none"> <code>defaultDense</code> - the correlation method <code>svdDense</code> - the SVD method
<code>covariance</code>	<code>defaultDense</code>	<code>SharedPtr<covariance::Batch<algorithmFPType, covariance::defaultDense> ></code>	The correlation and variance-covariance matrices algorithm to be used for PCA computations with the correlation method. For details, see Correlation and Variance-covariance Matrices. Batch Processing .

Algorithm Output

The PCA algorithm calculates the results described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
<code>eigenvalues</code>	Pointer to the $1 \times p$ numeric table that contains eigenvalues in the descending order. By default, this result is an object of the <code>HomogenNumericTable</code> class, but you can define the result as an object of any class derived from <code>NumericTable</code> except <code>PackedSymmetricMatrix</code> , <code>PackedTriangularMatrix</code> , and <code>CSRNumericTable</code> .
<code>eigenvectors</code>	Pointer to the $p \times p$ numeric table that contains eigenvectors in the row-major order. By default, this result is an object of the <code>HomogenNumericTable</code> class, but you can define the result as an

Result ID	Result
	object of any class derived from <code>NumericTable</code> except <code>PackedSymmetricMatrix</code> , <code>PackedTriangularMatrix</code> , and <code>CSRNumericTable</code> .

Examples

Refer to the following examples in your Intel DAAL directory:

C++:

- `./examples/cpp/source/pca/pca_correlation_dense_batch.cpp`
- `./examples/cpp/source/pca/pca_correlation_csr_batch.cpp`
- `./examples/cpp/source/pca/pca_svd_dense_batch.cpp`

Java*:

- `./examples/java/source/com/intel/daal/examples/pca/PCACorrelationDenseBatch.java`
- `./examples/java/source/com/intel/daal/examples/pca/PCACorrelationCSRBatch.java`
- `./examples/java/source/com/intel/daal/examples/pca/PCASVDDenseBatch.java`

Python*:

- `./examples/python/source/pca/pca_correlation_dense_batch.py`
- `./examples/python/source/pca/pca_correlation_csr_batch.py`
- `./examples/python/source/pca/pca_svd_dense_batch.py`

Online Processing

Online processing computation mode assumes that data arrives in blocks $i = 1, 2, 3, \dots nblocks$.

PCA computation in the online processing mode follows the general computation schema for online processing described in [Algorithms](#).

Algorithm Input

The PCA algorithm in the online processing mode accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
<code>data</code>	Pointer to the $n_i \times p$ numeric table that represents the current data block. The input can be an object of any class derived from <code>NumericTable</code> .

Algorithm Parameters

The PCA algorithm in the online processing mode has the following parameters, depending on the computation method parameter *method*:

Parameter	<i>method</i>	Default Value	Description
<code>algorithmFPTType</code>	<code>defaultDense</code> or <code>svdDense</code>	<code>double</code>	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .

Parameter	method	Default Value	Description
<i>method</i>	Not applicable	defaultDense	Available methods for PCA computation: <ul style="list-style-type: none"> defaultDense - the correlation method svdDense - the SVD method
<i>initialization Procedure</i>	defaultDense	Not applicable	The procedure for setting initial parameters of the algorithm in the online processing mode. By default, the algorithm initializes <code>nObservationsCorrelation</code> , <code>sumCorrelation</code> , and <code>crossProductCorrelation</code> with zeros.
	svdDense	Not applicable	The procedure for setting initial parameters of the algorithm in the online processing mode. By default, the algorithm initializes <code>nObservationsSVD</code> , <code>sumSVD</code> , and <code>sumSquaresSVD</code> with zeros.
<i>covariance</i>	defaultDense	<code>SharedPtr<covariance::Online<algorithmFPType, covariance::defaultDense> ></code>	The correlation and variance-covariance matrices algorithm to be used for PCA computations with the correlation method. For details, see Correlation and Variance-covariance Matrices. Online Processing .

Partial Results

The PCA algorithm in the online processing mode calculates partial results described below. They depend on the computation method. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
Correlation method (defaultDense):	
<code>nObservationsCorrelation</code>	Pointer to the 1 x 1 numeric table with the number of observations processed so far. By default, this result is an object of the <code>HomogenNumericTable</code> class, but you can define it as an object of any class derived from <code>NumericTable</code> except <code>CSRNumericTable</code> .

Result ID	Result
<code>crossProductCorrelation</code>	Pointer to the $p \times p$ numeric table with the partial cross-product matrix computed so far. By default, this table is an object of the <code>HomogenNumericTable</code> class, but you can define it as an object of any class derived from <code>NumericTable</code> except <code>PackedSymmetricMatrix</code> , <code>PackedTriangularMatrix</code> , and <code>CSRNumericTable</code> .
<code>sumCorrelation</code>	Pointer to the $1 \times p$ numeric table with partial sums computed so far. By default, this table is an object of the <code>HomogenNumericTable</code> class, but you can define it as an object of any class derived from <code>NumericTable</code> except <code>PackedSymmetricMatrix</code> , <code>PackedTriangularMatrix</code> , and <code>CSRNumericTable</code> .
SVD method (<code>svdDense</code>):	
<code>nObservationsSVD</code>	Pointer to the 1×1 numeric table with the number of observations processed so far. By default, this result is an object of the <code>HomogenNumericTable</code> class, but you can define it as an object of any class derived from <code>NumericTable</code> except <code>CSRNumericTable</code> .
<code>sumSVD</code>	Pointer to the $1 \times p$ numeric table with partial sums computed so far. By default, this table is an object of the <code>HomogenNumericTable</code> class, but you can define it as an object of any class derived from <code>NumericTable</code> except <code>PackedSymmetricMatrix</code> , <code>PackedTriangularMatrix</code> , and <code>CSRNumericTable</code> .
<code>sumSquaresSVD</code>	Pointer to the $1 \times p$ numeric table with partial sums of squares computed so far. By default, this table is an object of the <code>HomogenNumericTable</code> class, but you can define it as an object of any class derived from <code>NumericTable</code> except <code>PackedSymmetricMatrix</code> , <code>PackedTriangularMatrix</code> , and <code>CSRNumericTable</code> .

Algorithm Output

The PCA algorithm in the online processing mode calculates the results described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
<code>eigenvalues</code>	Pointer to the $1 \times p$ numeric table that contains eigenvalues in the descending order. By default, this result is an object of the <code>HomogenNumericTable</code> class, but you can define the result as an object of any class derived from <code>NumericTable</code> except <code>PackedSymmetricMatrix</code> , <code>PackedTriangularMatrix</code> , and <code>CSRNumericTable</code> .
<code>eigenvectors</code>	Pointer to the $p \times p$ numeric table that contains eigenvectors in the row-major order. By default, this result is an object of the <code>HomogenNumericTable</code> class, but you can define the result as an

Result ID	Result
	object of any class derived from <code>NumericTable</code> except <code>PackedSymmetricMatrix</code> , <code>PackedTriangularMatrix</code> , and <code>CSRNumericTable</code> .

Examples

Refer to the following examples in your Intel DAAL directory:

C++:

- `./examples/cpp/source/pca/pca_correlation_dense_online.cpp`
- `./examples/cpp/source/pca/pca_correlation_csr_online.cpp`
- `./examples/cpp/source/pca/pca_svd_dense_online.cpp`

Java*:

- `./examples/java/source/com/intel/daal/examples/pca/PCACorrelationDenseOnline.java`
- `./examples/java/source/com/intel/daal/examples/pca/PCACorrelationCSROnline.java`
- `./examples/java/source/com/intel/daal/examples/pca/PCASVDDenseOnline.java`

Python*:

- `./examples/python/source/pca/pca_correlation_dense_online.py`
- `./examples/python/source/pca/pca_correlation_csr_online.py`
- `./examples/python/source/pca/pca_svd_dense_online.py`

Distributed Processing

This mode assumes that data set is split in *nblocks* blocks across computation nodes.

PCA computation in the distributed processing mode follows the general schema described in [Algorithms](#).

Algorithm Parameters

The PCA algorithm in the distributed processing mode has the following parameters, depending on the computation method parameter *method*:

Parameter	method	Default Value	Description
<i>computeStep</i>	defaultDense or svdDense	Not applicable	The parameter required to initialize the algorithm. Can be: <ul style="list-style-type: none"> • <code>step1Local</code> - the first step, performed on local nodes • <code>step2Master</code> - the second step, performed on a master node
<i>algorithmFPTType</i>	defaultDense or svdDense	double	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<i>method</i>	Not applicable	defaultDense	Available methods for PCA computation:

Parameter	method	Default Value	Description
			<ul style="list-style-type: none"> defaultDense - the correlation method svdDense - the SVD method
<code>covariance</code>	<code>defaultDense</code>	<code>SharedPtr<covariance::Distributed <compute Step, algorithmFPType, covariance::defaultDense> ></code>	The correlation and variance-covariance matrices algorithm to be used for PCA computations with the correlation method. For details, see Correlation and Variance-covariance Matrices. Distributed Processing .

Correlation Method (`defaultDense`)

Use the following two-step schema:

Step 1 - on Local Nodes

In this step, the PCA algorithm accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
<code>data</code>	Pointer to the $n_i \times p$ numeric table that represents the i -th data block on the local node. The input can be an object of any class derived from <code>NumericTable</code> .

In this step, PCA calculates the results described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
<code>nObservationsCorrelation</code>	Pointer to the 1×1 numeric table with the number of observations processed so far on the local node. By default, this result is an object of the <code>HomogenNumericTable</code> class, but you can define it as an object of any class derived from <code>NumericTable</code> except <code>CSRNumericTable</code> .
<code>crossProductCorrelation</code>	Pointer to the $p \times p$ numeric table with the cross-product matrix computed so far on the local node. By default, this table is an object of the <code>HomogenNumericTable</code> class, but you can define it as an object of any class derived from <code>NumericTable</code> except <code>PackedSymmetricMatrix</code> , <code>PackedTriangularMatrix</code> , and <code>CSRNumericTable</code> .
<code>sumCorrelation</code>	Pointer to the $1 \times p$ numeric table with partial sums computed so far on the local node. By default, this table is an object of the <code>HomogenNumericTable</code> class, but you can define it as an object of any class derived from <code>NumericTable</code> except <code>PackedSymmetricMatrix</code> , <code>PackedTriangularMatrix</code> , and <code>CSRNumericTable</code> .

Step 2 - on Master Node

In this step, the PCA algorithm accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
<code>partialResults</code>	A collection that contains results computed in Step 1 on local nodes (<code>nObservationsCorrelation</code> , <code>crossProductCorrelation</code> , and <code>sumCorrelation</code>). The collection can contain objects of any class derived from <code>NumericTable</code> except the <code>PackedSymmetricMatrix</code> and <code>PackedTriangularMatrix</code> .

In this step, PCA calculates the results described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
<code>eigenvalues</code>	Pointer to the $1 \times p$ numeric table that contains eigenvalues in the descending order. By default, this result is an object of the <code>HomogenNumericTable</code> class, but you can define the result as an object of any class derived from <code>NumericTable</code> except <code>PackedSymmetricMatrix</code> , <code>PackedTriangularMatrix</code> , and <code>CSRNumericTable</code> .
<code>eigenvectors</code>	Pointer to the $p \times p$ numeric table that contains eigenvectors in the row-major order. By default, this result is an object of the <code>HomogenNumericTable</code> class, but you can define the result as an object of any class derived from <code>NumericTable</code> except <code>PackedSymmetricMatrix</code> , <code>PackedTriangularMatrix</code> , and <code>CSRNumericTable</code> .

Examples

Refer to the following examples in your Intel DAAL directory:

C++:

- `./examples/cpp/source/pca/pca_correlation_dense_distributed.cpp`
- `./examples/cpp/source/pca/pca_correlation_csr_distributed.cpp`

Java*:

- `./examples/java/source/com/intel/daal/examples/pca/PCACorrelationDenseDistributed.java`
- `./examples/java/source/com/intel/daal/examples/pca/PCACorrelationCSR Distributed.java`

Python*:

- `./examples/python/source/pca/pca_correlation_dense_distributed.py`
- `./examples/python/source/pca/pca_correlation_csr_distributed.py`

SVD Method (`svdDense`)

Use the following two-step schema:

Step 1 - on Local Nodes

In this step, the PCA algorithm accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
<code>data</code>	Pointer to the $n_i \times p$ numeric table that represents the i -th data block on the local node. The input can be an object of any class derived from <code>NumericTable</code> .

In this step, PCA calculates the results described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
<code>nObservationsSVD</code>	Pointer to the 1×1 numeric table with the number of observations processed so far on the local node. By default, this result is an object of the <code>HomogenNumericTable</code> class, but you can define it as an object of any class derived from <code>NumericTable</code> except <code>CSRNumericTable</code> .
<code>sumSVD</code>	Pointer to the $1 \times p$ numeric table with partial sums computed so far on the local node. By default, this table is an object of the <code>HomogenNumericTable</code> class, but you can define it as an object of any class derived from <code>NumericTable</code> except <code>PackedSymmetricMatrix</code> , <code>PackedTriangularMatrix</code> , and <code>CSRNumericTable</code> .
<code>sumSquaresSVD</code>	Pointer to the $1 \times p$ numeric table with partial sums of squares computed so far on the local node. By default, this table is an object of the <code>HomogenNumericTable</code> class, but you can define it as an object of any class derived from <code>NumericTable</code> except <code>PackedSymmetricMatrix</code> , <code>PackedTriangularMatrix</code> , and <code>CSRNumericTable</code> .
<code>auxiliaryDataSVD</code>	A collection of numeric tables each with the partial result to transmit to the master node for Step 2 . The collection can contain objects of any class derived from <code>NumericTable</code> except the <code>PackedSymmetricMatrix</code> and <code>PackedTriangularMatrix</code> .

Step 2 - on Master Node

In this step, the PCA algorithm accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
<code>partialResults</code>	A collection that contains results computed in Step 1 on local nodes (<code>nObservationsSVD</code> , <code>sumSVD</code> , <code>sumSquaresSVD</code> , and <code>auxiliaryDataSVD</code>). The collection can contain objects of any class derived from <code>NumericTable</code> except <code>PackedSymmetricMatrix</code> and <code>PackedTriangularMatrix</code> .

In this step, PCA calculates the results described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
<code>eigenvalues</code>	Pointer to the $1 \times p$ numeric table that contains eigenvalues in the descending order. By default, this result is an object of the <code>HomogenNumericTable</code> class, but you can define the result as an

Result ID	Result
eigenvectors	<p>object of any class derived from <code>NumericTable</code> except <code>PackedSymmetricMatrix</code>, <code>PackedTriangularMatrix</code>, and <code>CSRNumericTable</code>.</p> <p>Pointer to the $p \times p$ numeric table that contains eigenvectors in the row-major order. By default, this result is an object of the <code>HomogenNumericTable</code> class, but you can define the result as an object of any class derived from <code>NumericTable</code> except <code>PackedSymmetricMatrix</code>, <code>PackedTriangularMatrix</code>, and <code>CSRNumericTable</code>.</p>

Examples

Refer to the following examples in your Intel DAAL directory:

C++: `./examples/cpp/source/pca/pca_svd_dense_distributed.cpp`

Java*: `./examples/java/source/com/intel/daal/examples/pca/PCASVDDenseDistributed.java`

Python*: `./examples/python/source/pca/pca_svd_dense_distributed.py`

Performance Considerations

To get the best overall performance of the PCA algorithm:

- If input data is homogeneous, provide the input data and store results in homogeneous numeric tables of the same type as specified in the `algorithmFPType` class template parameter.
- If input data is non-homogeneous, use AOS layout rather than SOA layout.

PCA computation using the correlation method involves the correlation and variance-covariance matrices algorithm. Depending on the method of this algorithm, the performance of PCA computations may vary. For sparse data sets, use the methods of this algorithm for sparse data. For available methods of the correlation and variance-covariance matrices algorithm, see:

- [Batch Processing](#)
- [Online Processing](#)
- [Distributed Processing](#)

Batch Processing

Because the PCA in the batch processing mode performs normalization for `data` passed as Input ID, to achieve the best performance, normalize the input data set. To inform the algorithm that the data is normalized, set the normalization flag for the input numeric table that represents your data set by calling the `setNormalizationFlag()` method of the `NumericTableIface` class.

Because the PCA with the correlation method (`defaultDense`) in the batch processing mode is based on the computation of the correlation matrix, to achieve the best performance, precompute the correlation matrix. To pass the precomputed correlation matrix to the algorithm, use `correlation` as Input ID.

Online Processing

PCA with the SVD method (`svdDense`) in the online processing mode is at least as computationally complex as in the batch processing mode and has high memory requirements for storing auxiliary data between calls to `compute()`. On the other hand, the online version of the PCA with the SVD method may enable you to hide the latency of reading data from a slow data source. To do this, implement load prefetching of the next data block in parallel with the `compute()` method for the current block.

Distributed Processing

PCA with the SVD method (`svdDense`) in the distributed processing mode requires gathering local-node $p \times p$ numeric tables on the master node. When the amount of local-node work is small, that is, when the local-node data set is small, the network data transfer may become a bottleneck. To avoid this situation, ensure that local nodes have a sufficient amount of work. For example, distribute the input data set across a smaller number of nodes.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Cholesky Decomposition

Cholesky decomposition is a matrix factorization technique that decomposes a symmetric positive-definite matrix into a product of a lower triangular matrix and its conjugate transpose.

Because of numerical stability and superior efficiency in comparison with other methods, Cholesky decomposition is widely used in numerical methods for solving symmetric linear systems. It is also used in non-linear optimization problems, Monte Carlo simulation, and Kalman filtration.

Details

Given a symmetric positive-definite matrix X of size $p \times p$, the problem is to compute the Cholesky decomposition $X = LL^T$, where L is a lower triangular matrix.

Batch Processing

Algorithm Input

Cholesky decomposition accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
<code>data</code>	Pointer to the $p \times p$ numeric table that represents the symmetric positive-definite matrix X for which the Cholesky decomposition is computed. The input can be an object of any class derived from <code>NumericTable</code> that can represent symmetric matrices. For example, the <code>PackedTriangularMatrix</code> class cannot represent a symmetric matrix.

Algorithm Parameters

Cholesky decomposition has the following parameters:

Parameter	Default Value	Description
<code>algorithmFPTYPE</code>	double	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<code>method</code>	defaultDense	Performance-oriented computation method, the only method supported by the algorithm.

Algorithm Output

Cholesky decomposition calculates the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
<code>choleskyFactor</code>	Pointer to the $p \times p$ numeric table that represents the lower triangular matrix L (Cholesky factor). By default, the result is an object of the <code>HomogenNumericTable</code> class, but you can define the result as an object of any class derived from <code>NumericTable</code> except the <code>PackedSymmetricMatrix</code> class, <code>CSRNumericTable</code> class, and <code>PackedTriangularMatrix</code> class with the <code>upperPackedTriangularMatrix</code> layout.

Examples

Refer to the following examples in your Intel DAAL directory:

C++: `./examples/cpp/source/cholesky/cholesky_batch.cpp`

Java*: `./examples/java/source/com/intel/daal/examples/cholesky/CholeskyBatch.java`

Python*: `./examples/python/source/cholesky/cholesky_batch.py`

Performance Considerations

To get the best overall performance when Cholesky decomposition:

- If input data is homogeneous, for input matrix X and output matrix L use homogeneous numeric tables of the same type as specified in the `algorithmFPTYPE` class template parameter.
- If input data is non-homogeneous, use AOS layout rather than SOA layout.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Singular Value Decomposition

Singular Value Decomposition (SVD) is one of matrix factorization techniques. It has a broad range of applications including dimensionality reduction, solving linear inverse problems, and data fitting.

Details

Given the matrix X of size $n \times p$, the problem is to compute the Singular Value Decomposition (SVD)

$$X = U\Sigma V^t,$$

where

- U is an orthogonal matrix of size $n \times n$
- Σ is a rectangular diagonal matrix of size $n \times p$ with non-negative values on the diagonal, called singular values
- V^t is an orthogonal matrix of size $p \times p$

Columns of the matrices U and V are called left and right singular vectors, respectively.

Batch and Online Processing

Online processing computation mode assumes that the data arrives in blocks $i = 1, 2, 3, \dots, nblocks$.

Algorithm Input

The SVD algorithm accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
data	<p>Pointer to the numeric table that represents:</p> <ul style="list-style-type: none"> • For batch processing, the entire $n \times p$ matrix X to be factorized. • For online processing, the $n_i \times p$ submatrix of X that represents the current data block in the online processing mode. Note that each current data block must have sufficient size: $n_i > p$. <p>The input can be an object of any class derived from <code>NumericTable</code>.</p>

Algorithm Parameters

The SVD algorithm has the following parameters:

Parameter	Default Value	Description
<code>algorithmFPType</code>	<code>double</code>	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<code>method</code>	<code>defaultDense</code>	Performance-oriented computation method, the only method supported by the algorithm.
<code>leftSingularMatrix</code>	<code>requiredInPackedForm</code>	<p>Specifies whether the matrix of left singular vectors is required. Can be:</p> <ul style="list-style-type: none"> • <code>notRequired</code> - the matrix is not required • <code>requiredInPackedForm</code> - the matrix in the packed format is required.
<code>rightSingularMatrix</code>	<code>requiredInPackedForm</code>	<p>Specifies whether the matrix of right singular vectors is required. Can be:</p> <ul style="list-style-type: none"> • <code>notRequired</code> - the matrix is not required

Parameter	Default Value	Description
		<ul style="list-style-type: none"> <code>requiredInPackedForm</code> - the matrix in the packed format is required.

Algorithm Output

The SVD algorithm calculates the results described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
<code>singularValues</code>	Pointer to the $1 \times p$ numeric table with singular values (the diagonal of the matrix Σ). By default, this result is an object of the <code>HomogenNumericTable</code> class, but you can define the result as an object of any class derived from <code>NumericTable</code> except <code>PackedSymmetricMatrix</code> , <code>PackedTriangularMatrix</code> , and <code>CSRNumericTable</code> .
<code>leftSingularMatrix</code>	Pointer to the $n \times p$ numeric table with left singular vectors (matrix U). Pass <code>NULL</code> if left singular vectors are not required. By default, this result is an object of the <code>HomogenNumericTable</code> class, but you can define the result as an object of any class derived from <code>NumericTable</code> except <code>PackedSymmetricMatrix</code> , <code>PackedTriangularMatrix</code> , and <code>CSRNumericTable</code> .
<code>rightSingularMatrix</code>	Pointer to the $p \times p$ numeric table with right singular vectors (matrix V). Pass <code>NULL</code> if right singular vectors are not required. By default, this result is an object of the <code>HomogenNumericTable</code> class, but you can define the result as an object of any class derived from <code>NumericTable</code> except <code>PackedSymmetricMatrix</code> , <code>PackedTriangularMatrix</code> , and <code>CSRNumericTable</code> .

Examples

Refer to the following examples in your Intel DAAL directory:

C++:

- `./examples/cpp/source/svd/svd_batch.cpp`
- `./examples/cpp/source/svd/svd_online.cpp`

Java*:

- `./examples/java/source/com/intel/daal/examples/svd/SVDBatch.java`
- `./examples/java/source/com/intel/daal/examples/svd/SVDOnline.java`

Python*:

- `./examples/python/source/svd/svd_batch.py`
- `./examples/python/source/svd/svd_online.py`

Distributed Processing

This mode assumes that data set is split in *nblocks* blocks across computation nodes.

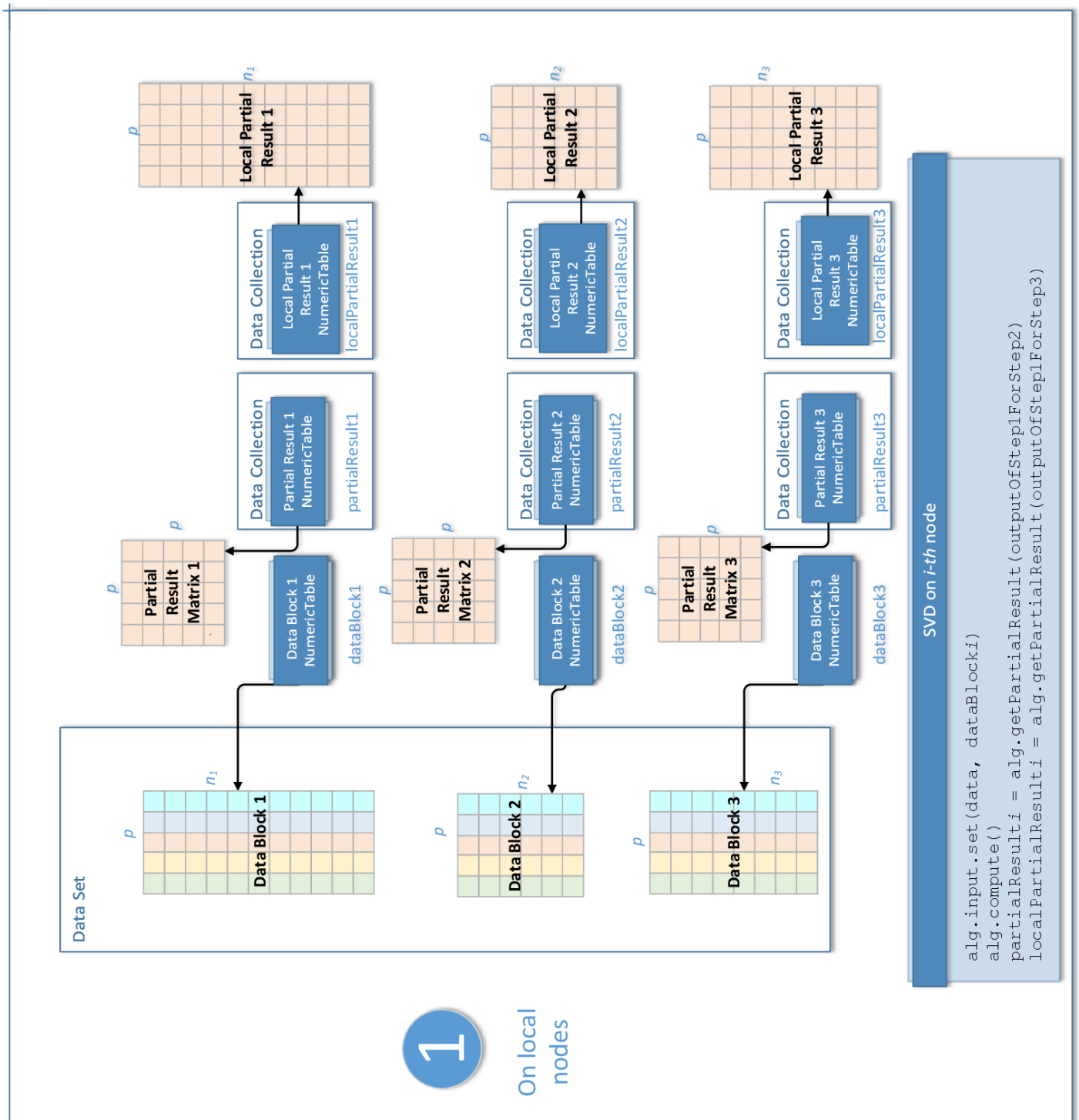
Algorithm Parameters

The SVD algorithm in the distributed processing mode has the following parameters:

Parameter	Default Value	Description
<code>computeStep</code>	Not applicable	<p>The parameter required to initialize the algorithm. Can be:</p> <ul style="list-style-type: none"> <code>step1Local</code> - the first step, performed on local nodes <code>step2Master</code> - the second step, performed on a master node <code>step3Local</code> - the final step, performed on local nodes
<code>algorithmFPTType</code>	<code>double</code>	<p>The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code>.</p>
<code>method</code>	<code>defaultDense</code>	<p>Performance-oriented computation method, the only method supported by the algorithm.</p>
<code>leftSingularMatrix</code>	<code>requiredInPackedForm</code>	<p>Specifies whether the matrix of left singular vectors is required. Can be:</p> <ul style="list-style-type: none"> <code>notRequired</code> - the matrix is not required <code>requiredInPackedForm</code> - the matrix in the packed format is required.
<code>rightSingularMatrix</code>	<code>requiredInPackedForm</code>	<p>Specifies whether the matrix of right singular vectors is required. Can be:</p> <ul style="list-style-type: none"> <code>notRequired</code> - the matrix is not required <code>requiredInPackedForm</code> - the matrix in the packed format is required.

Use the three-step computation schema to compute SVD:

Step 1 - on Local Nodes



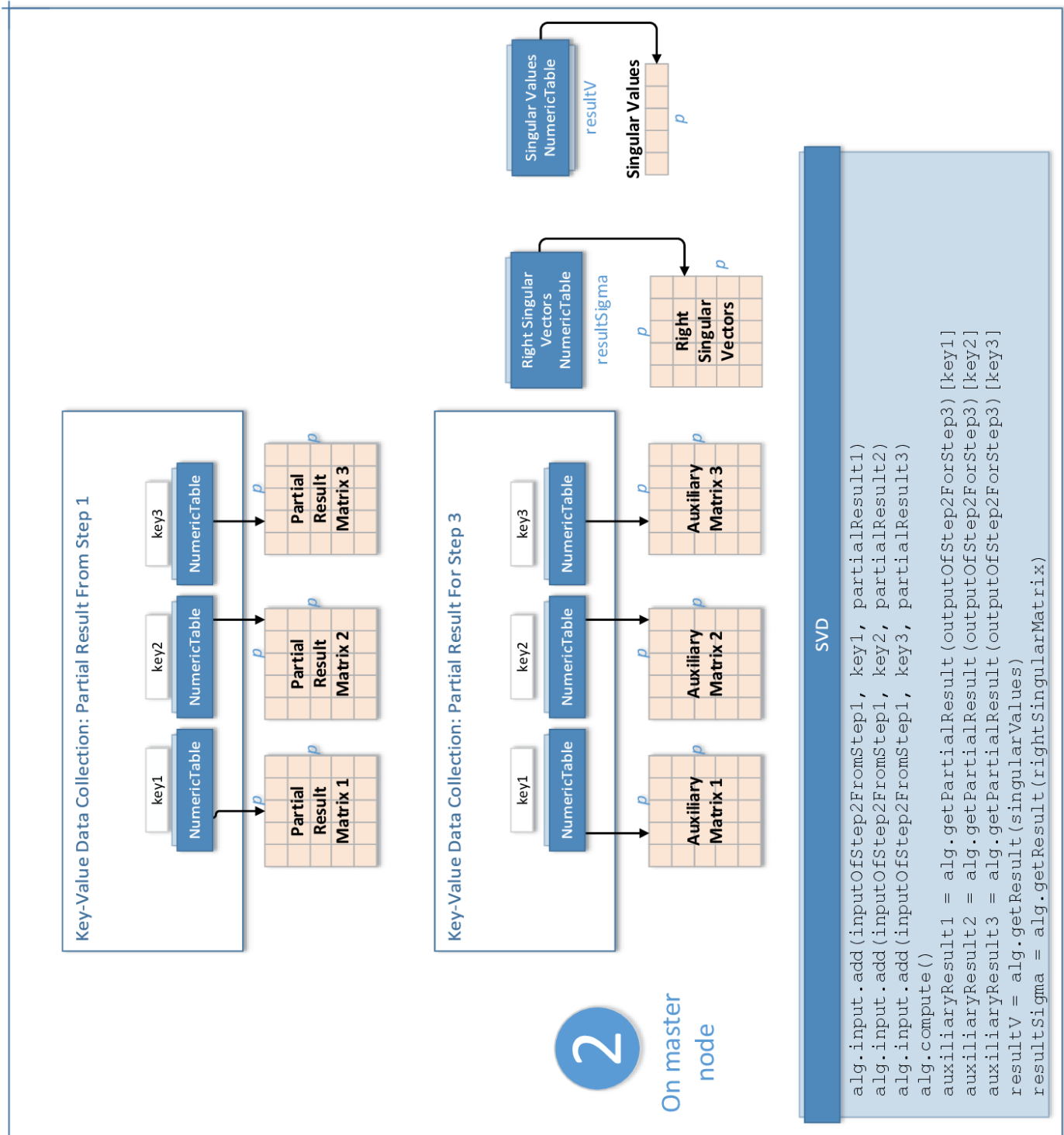
In this step, SVD accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
data	Pointer to the $n_i \times p$ numeric table that represents the i -th data block on the local node. Note that each data block must have sufficient size: $n_i > p$. The input can be an object of any class derived from <code>NumericTable</code> .

In this step, SVD calculates the results described below. Pass the Partial Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Partial Result ID	Result
outputOfStep1ForStep2	A collection that contains numeric tables each with the partial result to transmit to the master node for Step 2 . By default, these tables are objects of the <code>HomogenNumericTable</code> class, but you can define them as objects of any class derived from <code>NumericTable</code> except <code>PackedSymmetricMatrix</code> , <code>PackedTriangularMatrix</code> , and <code>CSRNumericTable</code> .
outputOfStep1ForStep3	A collection that contains numeric tables each with the partial result to keep on the local node for Step 3 . By default, these tables are objects of the <code>HomogenNumericTable</code> class, but you can define them as objects of any class derived from <code>NumericTable</code> except <code>PackedSymmetricMatrix</code> , <code>PackedTriangularMatrix</code> , and <code>CSRNumericTable</code> .

Step 2 - on Master Node



In this step, SVD accepts the input from each local node described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

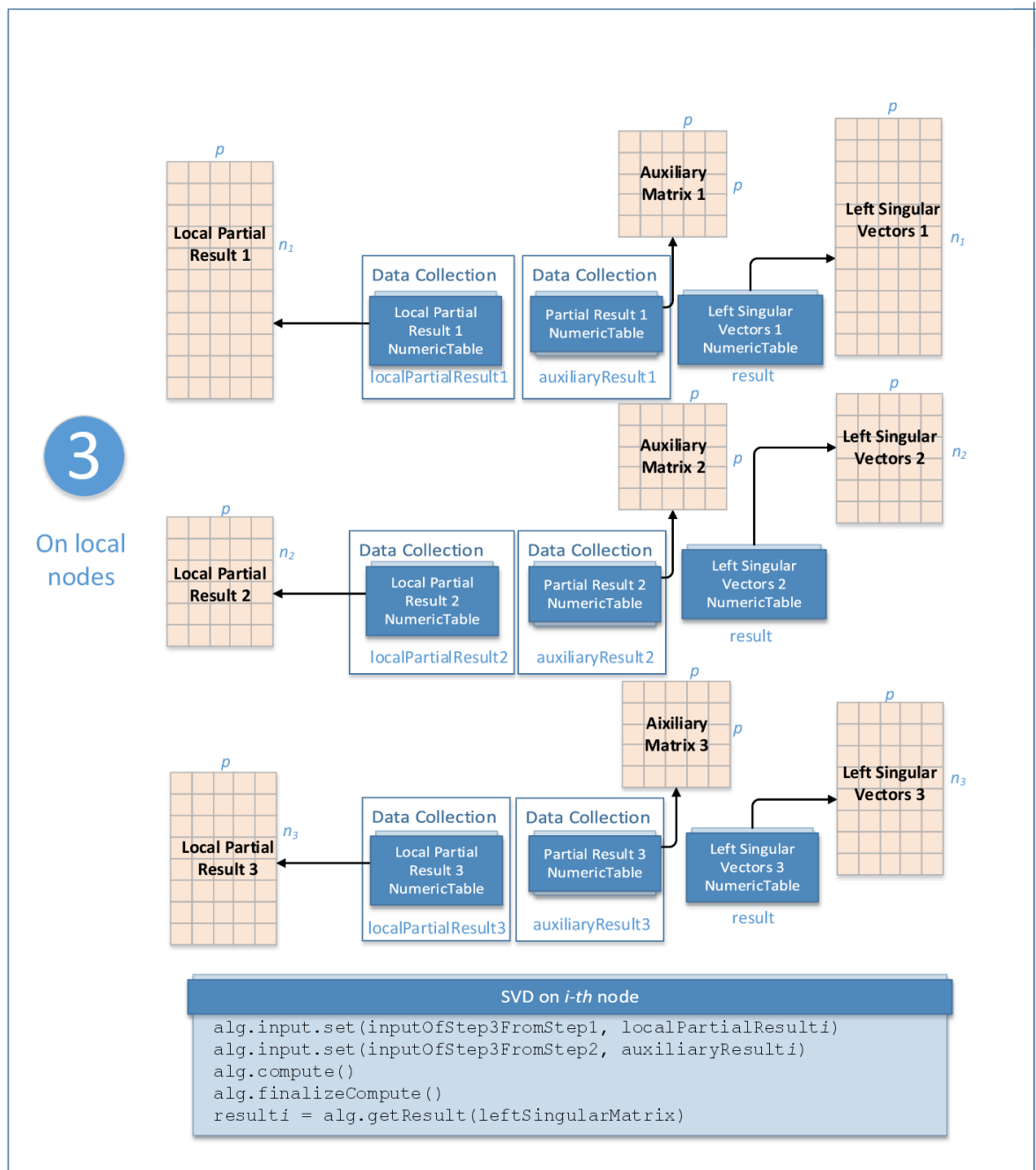
Input ID	Input
<code>inputOfStep2FromStep1</code>	A collection that contains results computed in Step 1 on local nodes (<code>outputOfStep1ForStep2</code>). The collection can contain objects of any class derived from <code>NumericTable</code> except the <code>PackedSymmetricMatrix</code> class and <code>PackedTriangularMatrix</code> class with the <code>lowerPackedTriangularMatrix</code> layout.
<code>key</code>	A key, a number of type <code>int</code> . Keys enable tracking the order in which partial results from Step 1 (<code>inputOfStep2FromStep1</code>) come to the master node, so that the partial results computed in Step 2 (<code>outputOfStep2ForStep3</code>) can be delivered back to local nodes in exactly the same order.

In this step, SVD calculates the results described below. Pass the Partial Result ID or Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Partial Result ID	Result
<code>outputOfStep2ForStep3</code>	A collection that contains numeric tables to be split across local nodes to compute left singular vectors. Set to <code>NULL</code> if you do not need left singular vectors. By default, these tables are objects of the <code>HomogenNumericTable</code> class, but you can define them as objects of any class derived from <code>NumericTable</code> except <code>PackedSymmetricMatrix</code> , <code>PackedTriangularMatrix</code> , and <code>CSRNumericTable</code> .

Result ID	Result
<code>singularValues</code>	Pointer to the $1 \times p$ numeric table with singular values (the diagonal of the matrix Σ). By default, this result is an object of the <code>HomogenNumericTable</code> class, but you can define the result as an object of any class derived from <code>NumericTable</code> except <code>PackedSymmetricMatrix</code> , <code>PackedTriangularMatrix</code> , and <code>CSRNumericTable</code> .
<code>rightSingularMatrix</code>	Pointer to the $p \times p$ numeric table with right singular vectors (matrix V). Pass <code>NULL</code> if right singular vectors are not required. By default, this result is an object of the <code>HomogenNumericTable</code> class, but you can define the result as an object of any class derived from <code>NumericTable</code> except <code>PackedSymmetricMatrix</code> , <code>PackedTriangularMatrix</code> , and <code>CSRNumericTable</code> .

Step 3 - on Local Nodes



In this step, SVD accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
inputOfStep3FromStep1	A collection that contains results computed in Step 1 on local nodes (outputOfStep1ForStep3). The collection can contain objects of any class derived from NumericTable except the PackedSymmetricMatrix and PackedTriangularMatrix.

Input ID	Input
<code>inputOfStep3FromStep2</code>	A collection that contains results computed in Step 2 on local nodes (<code>outputOfStep2ForStep3</code>). The collection can contain objects of any class derived from <code>NumericTable</code> except <code>PackedSymmetricMatrix</code> and <code>PackedTriangularMatrix</code> .

In this step, SVD calculates the results described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
<code>leftSingularMatrix</code>	Pointer to the $n \times p$ numeric table with left singular vectors (matrix U). Pass <code>NULL</code> if left singular vectors are not required. By default, this result is an object of the <code>HomogenNumericTable</code> class, but you can define the result as an object of any class derived from <code>NumericTable</code> except <code>PackedSymmetricMatrix</code> , <code>PackedTriangularMatrix</code> , and <code>CSRNumericTable</code> .

Examples

Refer to the following examples in your Intel DAAL directory:

C++: `./examples/cpp/source/svd/svd_distributed.cpp`

Java*: `./examples/java/source/com/intel/daal/examples/svd/SVDDistributed.java`

Python*: `./examples/python/source/svd/svd_distributed.py`

Performance Considerations

To get the best overall performance of singular value decomposition (SVD), for input, output, and auxiliary data, use homogeneous numeric tables of the same type as specified in the `algorithmFPType` class template parameter.

Online Processing

SVD in the online processing mode is at least as computationally complex as in the batch processing mode and has high memory requirements for storing auxiliary data between calls to the `compute()` method. On the other hand, the online version of SVD may enable you to hide the latency of reading data from a slow data source. To do this, implement load prefetching of the next data block in parallel with the `compute()` method for the current block.

Online processing mostly benefits SVD when the matrix of left singular vectors is not required. In this case, memory requirements for storing auxiliary data goes down from $O(p*n)$ to $O(p*p*nblocks)$.

Distributed Processing

Using SVD in the distributed processing mode requires gathering local-node $p \times p$ numeric tables on the master node. When the amount of local-node work is small, that is, when the local-node data set is small, the network data transfer may become a bottleneck. To avoid this situation, ensure that local nodes have a sufficient amount of work. For example, distribute input data set across a smaller number of nodes.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-

Optimization Notice

dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Association Rules

Association rules mining is the method for uncovering the most important relationships between variables. Its main application is a store basket analysis, which aims at discovery of a relationship between groups of products with some level of confidence.

Details

The library provides Apriori algorithm for association rule mining [Agrawal94].

Let $I = \{i_1, i_2, \dots, i_m\}$ be a set of items (products) and subset $T \subset I$ is a transaction associated with item set I . The association rule has the form: $X \Rightarrow Y$, where $X \subset I$, $Y \subset I$, and intersection of X and Y is empty: $X \cap Y = \emptyset$. The left-hand-side set of items (*itemset*) X is called *antecedent*, while the right-hand-side itemset Y is called *consequent* of the rule.

Let $D = \{T_1, T_2, \dots, T_n\}$ be a set of transactions, each associated with item set I . Item subset $X \subset I$ has support s in the transaction set D if s percent of transactions in D contains X .

The association rule $X \Rightarrow Y$ in the transaction set D holds with confidence c if c percent of transactions in D that contain X also contains Y . Confidence of the rule can be represented as conditional probability:

$\text{confidence}(X \Rightarrow Y) = \text{support}(X \cup Y) / \text{support}(X)$.

For a given set of transactions $D = \{T_1, T_2, \dots, T_n\}$, the minimum support s and minimum confidence c discover all item sets X with support greater than s and generate all association rules $X \Rightarrow Y$ with confidence greater than c .

Therefore, the association rule discovery is decomposed into two stages: mining (training) and discovery (prediction). The mining stage involves generation of large item sets, that is, the sets that have support greater than the given parameters. At the discovery stage, the algorithm generates association rules using the large item sets identified at the mining stage.

Batch Processing**Algorithm Input**

The association rules algorithm accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
data	<p>Pointer to the $n \times 2$ numeric table t with the mining data. Each row consists of two integers:</p> <ul style="list-style-type: none"> Transaction ID, the number between 0 and <code>nTransactions-1</code>. Item ID, the number between 0 and <code>nUniqueItems-1</code>. <p>The input can be an object of any class derived from <code>NumericTable</code> except <code>PackedTriangularMatrix</code> and <code>PackedSymmetricMatrix</code>.</p>

Algorithm Parameters

The association rules algorithm has the following parameters:

Parameter	Default Value	Description
<i>algorithmFPType</i>	double	The floating-point type that the algorithm uses for intermediate computations. Can be float or double.
<i>method</i>	defaultDense	The computation method used by the algorithm. The only method supported so far is Apriori.
<i>minSupport</i>	0.01	Minimal support, a number in the [0,1) interval.
<i>minConfidence</i>	0.6	Minimal confidence, a number in the [0,1) interval.
<i>nUniqueItems</i>	0	The total number of unique items. If set to zero, the library automatically determines the number of unique items from the input data.
<i>nTransactions</i>	0	The total number of transactions. If set to zero, the library automatically determines the number transactions from the input data.
<i>discoverRules</i>	true	A flag that enables generation of the rules from large item sets.
<i>itemsetsOrder</i>	itemsetsUnsorted	The sort order of returned item sets: <ul style="list-style-type: none"> • <i>itemsetsUnsorted</i> - not sorted • <i>itemsetsSortedBySupport</i> - sorted by support in a descending order
<i>rulesOrder</i>	rulesUnsorted	The sort order of returned rules: <ul style="list-style-type: none"> • <i>rulesUnsorted</i> - not sorted • <i>rulesSortedBySupport</i> - sorted by support in a descending order
<i>minItemsetSize</i>	0	A parameter that defines the minimal size of item sets to be included into the array of results. The value of zero imposes no limitations on the minimal size of item sets.
<i>maxItemsetSize</i>	0	A parameter that defines the maximal size of item sets to be included into the array of results. The value of zero imposes no limitations on the maximal size of item sets.

Algorithm Output

The association rules algorithm calculates the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
<code>largeItemsets</code>	<p>Pointer to the numeric table with large item sets. The number of rows in the table equals the number of items in the large item sets. Each row contains two integers:</p> <ul style="list-style-type: none"> • ID of the large item set, the number between 0 and <code>nLargeItemsets-1</code>. • ID of the item, the number between 0 and <code>nUniqueItems-1</code>. <p>By default, this result is an object of the <code>HomogenNumericTable</code> class, but you can define the result as an object of any class derived from <code>NumericTable</code> except <code>PackedSymmetricMatrix</code>, <code>PackedTriangularMatrix</code>, and <code>CSRNumericTable</code>.</p>
<code>largeItemsetsSupport</code>	<p>Pointer to the <code>nLargeItemsets</code> x 2 numeric table of support values. Each row contains two integers:</p> <ul style="list-style-type: none"> • ID of the large item set, the number between 0 and <code>nLargeItemsets-1</code>. • The support value, the number of times the item set is met in the array of transactions. <p>By default, this result is an object of the <code>HomogenNumericTable</code> class, but you can define the result as an object of any class derived from <code>NumericTable</code> except <code>PackedSymmetricMatrix</code>, <code>PackedTriangularMatrix</code>, and <code>CSRNumericTable</code>.</p>
<code>antecedentItemsets</code>	<p>Pointer to the <code>nAntecedentItems</code> x 2 numeric table that contains the left-hand-side (X) part of the association rules. Each row contains two integers:</p> <ul style="list-style-type: none"> • Rule ID, the number between 0 and <code>nAntecedentItems-1</code>. • Item ID, the number between 0 and <code>nUniqueItems-1</code>. <p>By default, this result is an object of the <code>HomogenNumericTable</code> class, but you can define the result as an object of any class derived from <code>NumericTable</code> except <code>PackedSymmetricMatrix</code>, <code>PackedTriangularMatrix</code>, and <code>CSRNumericTable</code>.</p>
<code>consequentItemsets</code>	<p>Pointer to the <code>nConsequentItems</code> x 2 numeric table that contains the right-hand-side (Y) part of the association rules. Each row contains two integers:</p> <ul style="list-style-type: none"> • Rule ID, the number between 0 and <code>nConsequentItems-1</code>. • Item ID, the number between 0 and <code>nUniqueItems-1</code>. <p>By default, this result is an object of the <code>HomogenNumericTable</code> class, but you can define the result as an object of any class derived from <code>NumericTable</code> except <code>PackedSymmetricMatrix</code>, <code>PackedTriangularMatrix</code>, and <code>CSRNumericTable</code>.</p>
<code>confidence</code>	<p>Pointer to the <code>nRules</code> x 1 numeric table that contains confidence values of rules, floating-point numbers between 0 and 1. Confidence value in the <i>i</i>-th position corresponds to the rule with the index <i>i</i>.</p>

Result ID	Result
	By default, this result is an object of the <code>HomogenNumericTable</code> class, but you can define the result as an object of any class derived from <code>NumericTable</code> except <code>PackedSymmetricMatrix</code> , <code>PackedTriangularMatrix</code> , and <code>CSRNumericTable</code> .
	<ul style="list-style-type: none"> • The library requires transactions and items for each transaction to be passed in the ascending order. • Numbering of rules starts at 0. • The library calculates the sizes of numeric tables intended for results in a call to the algorithm. Avoid allocating the memory in numeric tables intended for results because, in general, it is impossible to accurately estimate the required memory size. If the memory interfaced by the numeric tables is allocated and its amount is insufficient to store the results, the algorithm returns an error.

Examples

Refer to the following examples in your Intel DAAL directory:

C++: `./examples/cpp/source/association_rules/apriori_batch.cpp`

Java*: `./examples/java/source/com/intel/daal/examples/association_rules/AprioriBatch.java`

Python*: `./examples/python/source/association_rules/apriori_batch.py`

Performance Considerations

To get the best overall performance of the association rules algorithm, whenever possible use the following numeric tables and data types:

- A SOA numeric table of type `int` to store features.
- A homogenous numeric table of type `int` to store large item sets, support values, and left-hand-side and right-hand-side parts of association rules.
- A numeric table with the confidence values of the same data type as specified in the `algorithmFPType` template parameter of the class.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

QR Decomposition

QR decomposition is a matrix factorization technique that decomposes a matrix into a product of an orthogonal matrix Q and an upper triangular matrix R .

QR decomposition is used in solving linear inverse and least squares problems. It also serves as a basis for algorithms that find eigenvalues and eigenvectors.

QR Decomposition without Pivoting

Details

Given the matrix X of size $n \times p$, the problem is to compute the QR decomposition $X = QR$, where

- Q is an orthogonal matrix of size $n \times n$
- R is a rectangular upper triangular matrix of size $n \times p$

The library requires $n > p$. In this case:

$$X = QR = \begin{bmatrix} Q_1 & Q_2 \end{bmatrix} \cdot \begin{bmatrix} R_1 \\ 0 \end{bmatrix} = Q_1 R_1,$$

where the matrix Q_1 has the size $n \times p$ and R_1 has the size $p \times p$.

Batch and Online Processing

Online processing computation mode assumes that the data arrives in blocks $i = 1, 2, 3, \dots, nblocks$.

Algorithm Input

QR decomposition accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
<code>data</code>	<p>Pointer to the numeric table that represents:</p> <ul style="list-style-type: none"> • For batch processing, the entire $n \times p$ matrix X to be factorized. • For online processing, the $n_i \times p$ submatrix of X that represents the current data block in the online processing mode. Note that each current data block must have sufficient size: $n_i > p$. <p>The input can be an object of any class derived from <code>NumericTable</code>.</p>

Algorithm Parameters

QR decomposition has the following parameters:

Parameter	Default Value	Description
<code>algorithmFPTYPE</code>	<code>double</code>	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<code>method</code>	<code>defaultDense</code>	Performance-oriented computation method, the only method supported by the algorithm.

Algorithm Output

QR decomposition calculates the results described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
<code>matrixQ</code>	Pointer to the numeric table with the $n \times p$ matrix $Q1$. By default, this result is an object of the <code>HomogenNumericTable</code> class, but you can define the result as an object of any class derived from <code>NumericTable</code> except <code>PackedSymmetricMatrix</code> , <code>PackedTriangularMatrix</code> , and <code>CSRNumericTable</code> .
<code>matrixR</code>	Pointer to the numeric table with the $p \times p$ upper triangular matrix $R1$. By default, this result is an object of the <code>HomogenNumericTable</code> class, but you can define the result as an object of any class derived from <code>NumericTable</code> except the <code>PackedSymmetricMatrix</code> class, <code>CSRNumericTable</code> class, and <code>PackedTriangularMatrix</code> class with the <code>lowerPackedTriangularMatrix</code> layout.

Examples

Refer to the following examples in your Intel DAAL directory:

C++:

- `./examples/cpp/source/qr/qr_batch.cpp`
- `./examples/cpp/source/qr/qr_online.cpp`

Java*:

- `./examples/java/source/com/intel/daal/examples/qr/QRBatch.java`
- `./examples/java/source/com/intel/daal/examples/qr/QROnline.java`

Python*:

- `./examples/python/source/qr/qr_batch.py`
- `./examples/python/source/qr/qr_online.py`

Distributed Processing

This mode assumes that the data set is split into *nblocks* blocks across computation nodes.

Algorithm Parameters

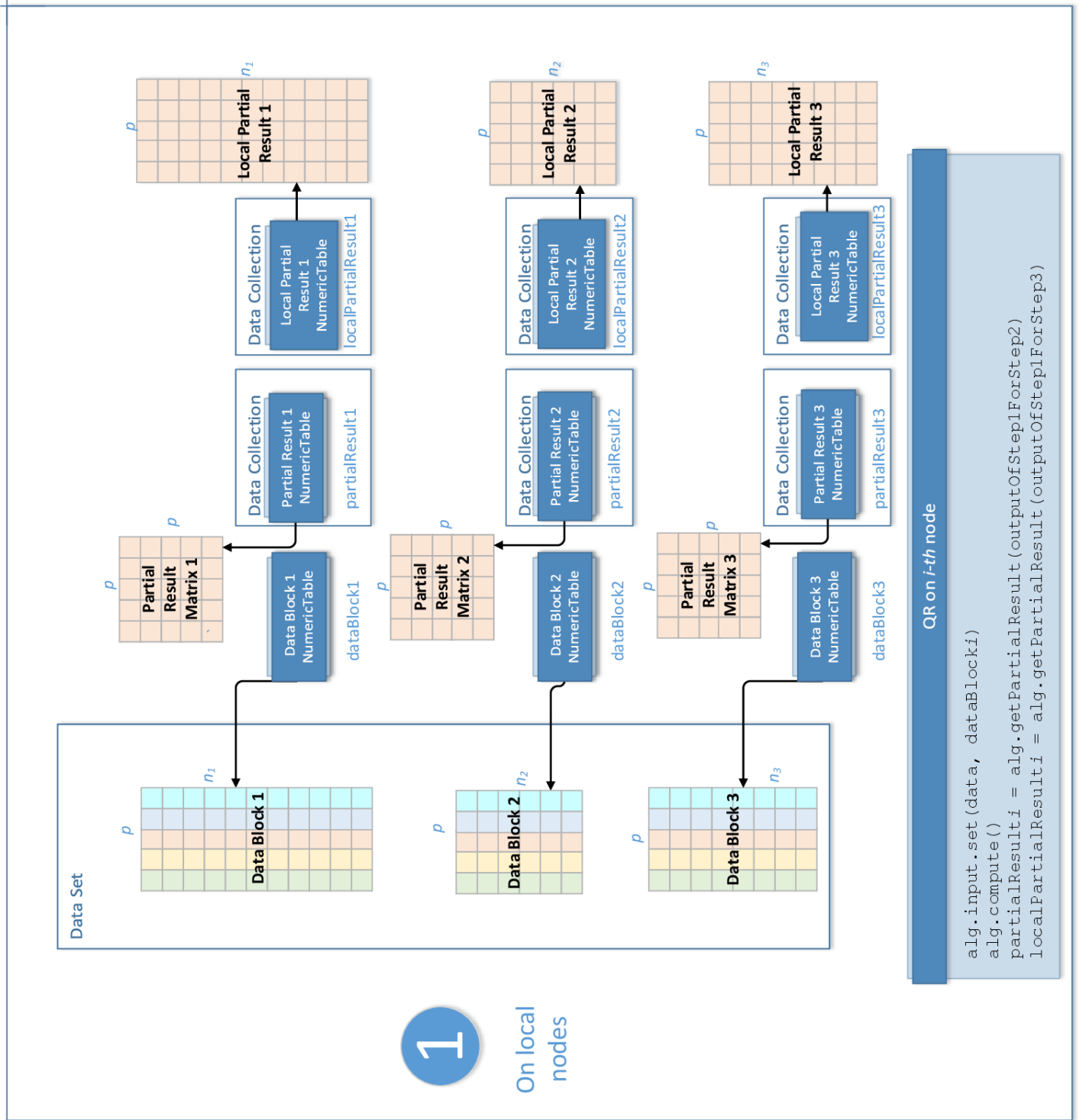
QR decomposition in the distributed processing mode has the following parameters:

Parameter	Default Value	Description
<code>computeStep</code>	Not applicable	The parameter required to initialize the algorithm. Can be: <ul style="list-style-type: none"> • <code>step1Local</code> - the first step, performed on local nodes • <code>step2Master</code> - the second step, performed on a master node • <code>step3Local</code> - the final step, performed on local nodes
<code>algorithmFPTYPE</code>	double	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .

Parameter	Default Value	Description
<i>method</i>	defaultDense	Performance-oriented computation method, the only method supported by the algorithm.

Use the three-step computation schema to compute QR decomposition:

Step 1 - on Local Nodes



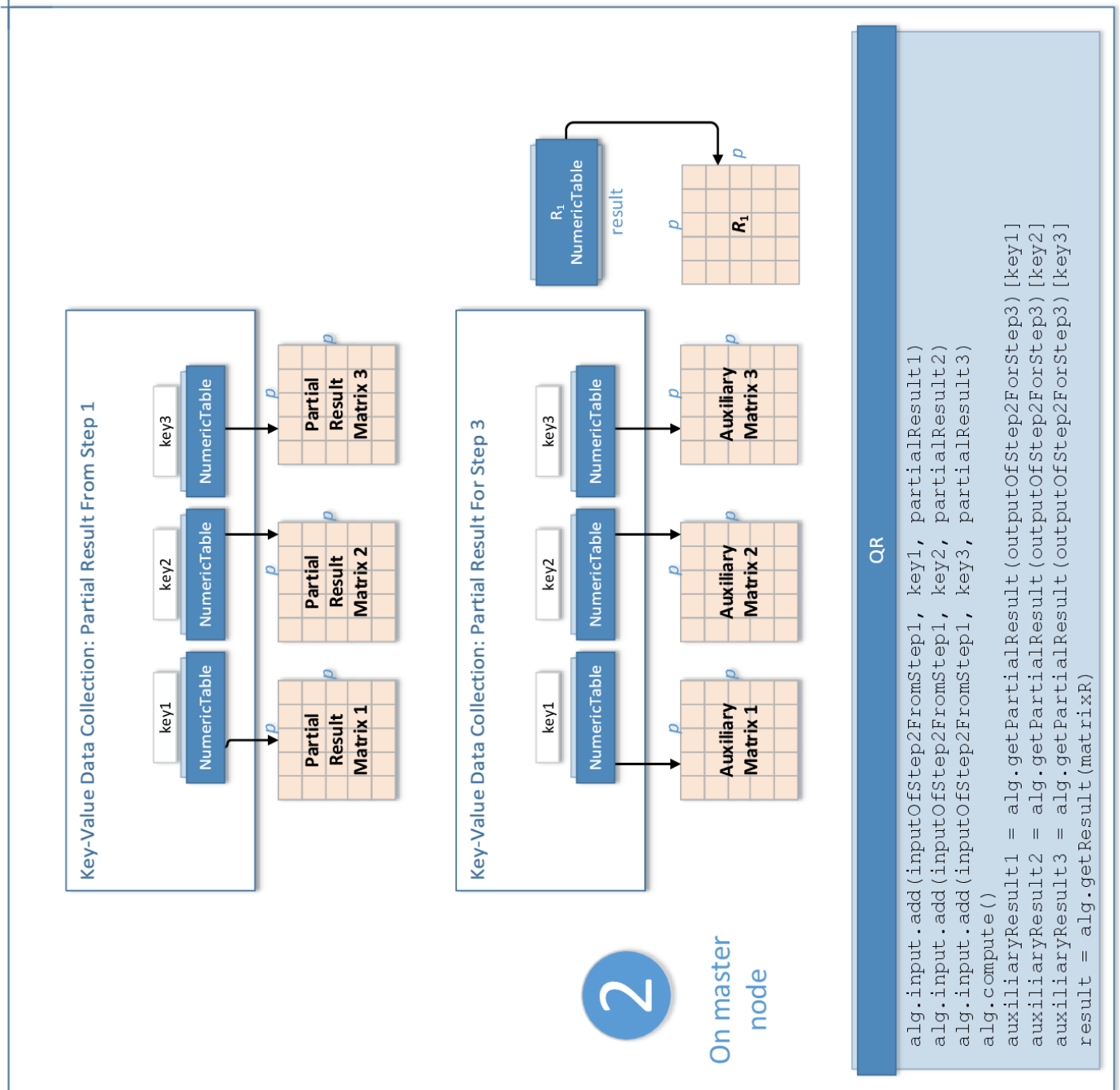
In this step, QR decomposition accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
data	Pointer to the $n_i \times p$ numeric table that represents the i -th data block on the local node. Note that each data block must have sufficient size: $n_i > p$. The input can be an object of any class derived from <code>NumericTable</code> .

In this step, QR decomposition calculates the results described below. Pass the Partial Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Partial Result ID	Result
outputOfStep1ForStep2	A collection that contains numeric tables each with the partial result to transmit to the master node for Step 2 . By default, these tables are objects of the <code>HomogenNumericTable</code> class, but you can define them as objects of any class derived from <code>NumericTable</code> except the <code>PackedSymmetricMatrix</code> class, <code>CSRNumericTable</code> class, and <code>PackedTriangularMatrix</code> class with the <code>lowerPackedTriangularMatrix</code> layout.
outputOfStep1ForStep3	A collection that contains numeric tables each with the partial result to keep on the local node for Step 3 . By default, these tables are objects of the <code>HomogenNumericTable</code> class, but you can define them as objects of any class derived from <code>NumericTable</code> except the <code>PackedSymmetricMatrix</code> , <code>PackedTriangularMatrix</code> , and <code>CSRNumericTable</code> .

Step 2 - on Master Node



In this step, QR decomposition accepts the input from each local node described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

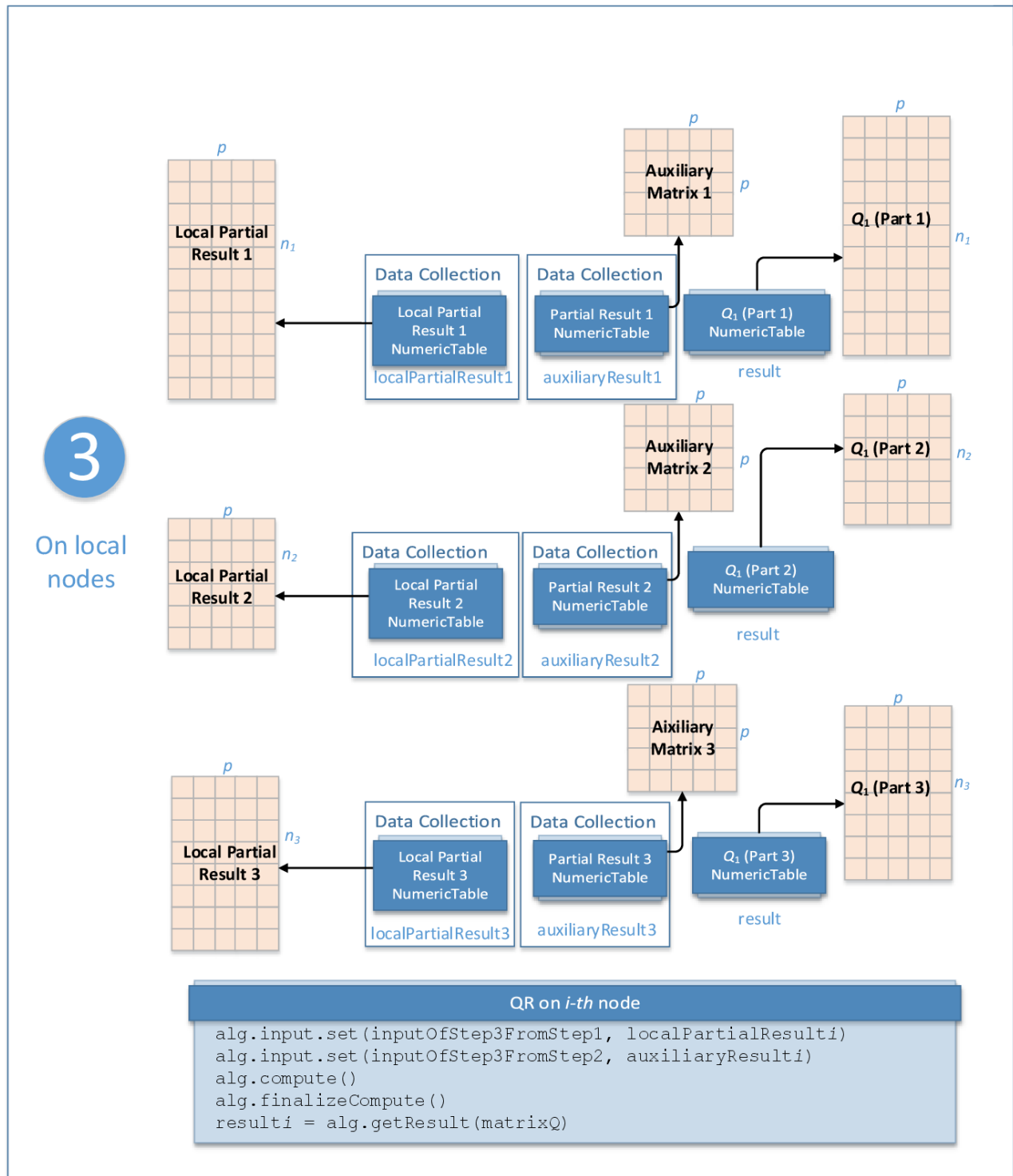
Input ID	Input
inputOfStep2FromStep1	A collection that contains results computed in Step 1 on local nodes (outputOfStep1ForStep2). This collection can contain objects of any class derived from <code>NumericTable</code> except the <code>PackedSymmetricMatrix</code> class and <code>PackedTriangularMatrix</code> class with the <code>lowerPackedTriangularMatrix</code> layout.
key	A key, a number of type <code>int</code> . Keys enable tracking the order in which partial results from Step 1 (inputOfStep2FromStep1) come to the master node, so that the partial results computed in Step 2 (outputOfStep2ForStep3) can be delivered back to local nodes in exactly the same order.

In this step, QR decomposition calculates the results described below. Pass the Result ID or Partial Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Partial Result ID	Result
outputOfStep2ForStep3	A collection that contains numeric tables to be split across local nodes to compute $Q1$. By default, these tables are objects of the <code>HomogenNumericTable</code> class, but you can define them as objects of any class derived from <code>NumericTable</code> except the <code>PackedSymmetricMatrix</code> class, <code>CSRNumericTable</code> class, and <code>PackedTriangularMatrix</code> class with the <code>lowerPackedTriangularMatrix</code> layout.

Result ID	Result
matrixR	Pointer to the numeric table with the $p \times p$ upper triangular matrix $R1$. By default, this result is an object of the <code>HomogenNumericTable</code> class, but you can define the result as an object of any class derived from <code>NumericTable</code> except the <code>PackedSymmetricMatrix</code> class, <code>CSRNumericTable</code> class, and <code>PackedTriangularMatrix</code> class with the <code>lowerPackedTriangularMatrix</code> layout.

Step 3 - on Local Nodes



In this step, QR decomposition accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
inputOfStep3FromStep1	A collection that contains results computed in Step 1 on local nodes (outputOfStep1ForStep3). The collection can contain objects of any class derived from <code>NumericTable</code> except the <code>PackedSymmetricMatrix</code> and <code>PackedTriangularMatrix</code> .
inputOfStep3FromStep2	A collection that contains results computed in Step 2 on local nodes (outputOfStep2ForStep3). The collection can contain objects of any class derived from <code>NumericTable</code> except the <code>PackedSymmetricMatrix</code> class and <code>PackedTriangularMatrix</code> class with the <code>lowerPackedTriangularMatrix</code> layout.

In this step, QR decomposition calculates the results described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
matrixQ	Pointer to the numeric table with the $n \times p$ matrix Q_1 . By default, the result is an object of the <code>HomogenNumericTable</code> class, but you can define the result as an object of any class derived from <code>NumericTable</code> except <code>PackedSymmetricMatrix</code> , <code>PackedTriangularMatrix</code> , and <code>CSRNumericTable</code> .

Examples

Refer to the following examples in your Intel DAAL directory:

C++: `./examples/cpp/source/qr/qr_distributed.cpp`

Java*: `./examples/java/source/com/intel/daal/examples/qr/QRDistributed.java`

Python*: `./examples/python/source/qr/qr_distributed.py`

Pivoted QR Decomposition

Details

Given the matrix X of size $n \times p$, the problem is to compute the QR decomposition with column pivoting $XP = QR$, where

- Q is an orthogonal matrix of size $n \times n$
- R is a rectangular upper triangular matrix of size $n \times p$
- P is a permutation matrix of size $n \times n$

The library requires $n > p$. In this case:

$$XP = QR = [Q_1, Q_2] \cdot \begin{bmatrix} R_1 \\ 0 \end{bmatrix} = Q_1 R_1,$$

where the matrix Q_1 has the size $n \times p$ and R_1 has the size $p \times p$.

Batch Processing

Algorithm Input

Pivoted QR decomposition accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
<code>data</code>	Pointer to the numeric table that represents the $n \times p$ matrix X to be factorized. The input can be an object of any class derived from <code>NumericTable</code> .

Algorithm Parameters

Pivoted QR decomposition has the following parameters:

Parameter	Default Value	Description
<code>algorithmFPTYPE</code>	<code>double</code>	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<code>method</code>	<code>defaultDense</code>	Performance-oriented computation method, the only method supported by the algorithm.
<code>permutedColumns</code>	<code>NULL</code>	<p>Pointer to the numeric table with the $1 \times p$ matrix with the information for the permutation:</p> <ul style="list-style-type: none"> If the i-th element is zero, the i-th column of the input matrix is a free column and may be permuted with any other free column during the computation. If the i-th element is non-zero, the i-th column of the input matrix is moved to the beginning of XP before the computation and remains in its place during the computation. <p>By default, this parameter is an object of the <code>HomogenNumericTable</code> class, filled by zeros. However, you can define this parameter as an object of any class derived from <code>NumericTable</code> except the <code>PackedSymmetricMatrix</code> class, <code>CSRNumericTable</code> class, and <code>PackedTriangularMatrix</code> class with the <code>lowerPackedTriangularMatrix</code> layout.</p>

Algorithm Output

Pivoted QR decomposition calculates the results described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
<code>matrixQ</code>	Pointer to the numeric table with the $n \times p$ matrix $Q1$. By default, this result is an object of the <code>HomogenNumericTable</code> class, but you can define the result as an object of any class derived from <code>NumericTable</code> except <code>PackedSymmetricMatrix</code> , <code>PackedTriangularMatrix</code> , and <code>CSRNumericTable</code> .
<code>matrixR</code>	Pointer to the numeric table with the $p \times p$ upper triangular matrix $R1$. By default, this result is an object of the <code>HomogenNumericTable</code> class, but you can define the result as an

Result ID	Result
	object of any class derived from <code>NumericTable</code> except the <code>PackedSymmetricMatrix</code> class, <code>CSRNumericTable</code> class, and <code>PackedTriangularMatrix</code> class with the <code>lowerPackedTriangularMatrix</code> layout.
<code>permutationMatrix</code>	<p>Pointer to the numeric table with the $1 \times p$ matrix such that <code>permutationMatrix(i) = k</code> if the column k of the full matrix X is permuted into the position i in XP.</p> <p>By default, this result is an object of the <code>HomogenNumericTable</code> class, but you can define the result as an object of any class derived from <code>NumericTable</code> except the <code>PackedSymmetricMatrix</code> class, <code>CSRNumericTable</code> class, and <code>PackedTriangularMatrix</code> class with the <code>lowerPackedTriangularMatrix</code> layout.</p>

Examples

Refer to the following examples in your Intel DAAL directory:

C++: `./examples/cpp/source/pivoted_qr/pivoted_qr_batch.cpp`

Java*: `./examples/java/source/com/intel/daal/examples/pivoted_qr/PivotedQRBatch.java`

Python*: `./examples/python/source/pivoted_qr/pivoted_qr_batch.py`

Performance Considerations

To get the best overall performance of the QR decomposition, for input, output, and auxiliary data, use homogeneous numeric tables of the same type as specified in the `algorithmFPType` class template parameter.

Online Processing

QR decomposition in the online processing mode is at least as computationally complex as in the batch processing mode and has high memory requirements for storing auxiliary data between calls to the `compute` method. On the other hand, the online version of QR decomposition may enable you to hide the latency of reading data from a slow data source. To do this, implement load prefetching of the next data block in parallel with the `compute()` method for the current block.

Online processing mostly benefits QR decomposition when the matrix Q is not required. In this case, memory requirements for storing auxiliary data goes down from $O(p*n)$ to $O(p*p*nblocks)$.

Distributed Processing

Using QR decomposition in the distributed processing mode requires gathering local-node $p \times p$ numeric tables on the master node. When the amount of local-node work is small, that is, when the local-node data set is small, the network data transfer may become a bottleneck. To avoid this situation, ensure that local nodes have a sufficient amount of work. For example, distribute the input data set across a smaller number of nodes.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-

Optimization Notice

dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Expectation-Maximization

Expectation-Maximization (EM) algorithm is an iterative method for finding the maximum likelihood and maximum a posteriori estimates of parameters in models that typically depend on hidden variables.

While serving as a clustering technique, EM is also used in non-linear dimensionality reduction, missing value problems, and other areas.

Details

Given a set X of n feature vectors $x_1 = (x_{11}, \dots, x_{1p})$, ..., $x_n = (x_{n1}, \dots, x_{np})$ of dimension p , the problem is to find a maximum-likelihood estimate of the parameters of the underlying distribution when the data is incomplete or has missing values.

Expectation-Maximization (EM) Algorithm in the General Form

Let X be the observed data which has log-likelihood $l(\theta; X)$ depending on the parameters θ . Let X^m be the latent or missing data, so that $T = (X, X^m)$ is the complete data with log-likelihood $l_0(\theta; X)$. The algorithm for solving the problem in its general form is the following EM algorithm ([Dempster77], [Hastie2009]):

1. Choose initial values of the parameters $\theta^{(0)}$
2. *Expectation step*: in the j -th step, compute $Q(\theta', \theta^{(j)}) = E(l_0(\theta'; T) | X, \theta^{(j)})$ as a function of the dummy argument θ'
3. *Maximization step*: in the j -th step, calculate the new estimate $\theta^{(j+1)}$ by maximizing $Q(\theta', \theta^{(j)})$ over θ'
4. Repeat steps 2 and 3 until convergence

EM algorithm for the Gaussian Mixture Model

Gaussian Mixture Model (GMM) is a mixture of k p -dimensional multivariate Gaussian distributions represented as

$$f(x | \alpha_1, \dots, \alpha_k; \theta_1, \dots, \theta_k) = \sum_{i=1}^k \alpha_i \int_{-\infty}^x pd(y | \theta_i),$$

where $\sum_{i=1}^k \alpha_i = 1$ and $\alpha_i \geq 0$.

The $pd(x | \theta_i)$ is the probability density function with parameters $\theta_i = (m_i, \Sigma_i)$, where m_i is the vector of means, and Σ_i is the variance-covariance matrix. The probability density function for a p -dimensional multivariate Gaussian distribution is defined as follows:

$$pd(x | \theta_i) = \frac{\exp\left(-\frac{1}{2}(x - m_i)^T \Sigma_i^{-1}(x - m_i)\right)}{\sqrt{(2\pi)^p |\Sigma_i|}}.$$

Let $z_{ij} = I\{x_i \text{ belongs to } j \text{ mixture component}\}$ be the indicator function and $\theta = (\alpha_1, \dots, \alpha_k; \theta_1, \dots, \theta_k)$.

Computation

The EM algorithm for GMM includes the following steps:

Define the weights as follows:

$$w_{ij} = \frac{pd(x_i | z_{ij}, \theta_j) \alpha_j}{\sum_{r=1}^k pd(x_i | z_{ir}, \theta_r) \alpha_r}.$$

for $i=1, \dots, n$ and $j=1, \dots, k$.

1. Choose initial values of the parameters

$$\theta^{(0)} = (\alpha_1^{(0)}, \dots, \alpha_k^{(0)}; \theta_1^{(0)}, \dots, \theta_k^{(0)})$$

2. *Expectation step*: in the j -th step, compute the matrix $W = (w_{ij})_{n \times k}$ with the weights w_{ij}
3. *Maximization step*: in the j -th step, for all $r=1, \dots, k$ compute:
 - a. The mixture weights

$$\alpha_r^{(j+1)} = \frac{n_r}{n},$$

where

$$n_r = \sum_{i=1}^n w_{ir}$$

is the "amount" of the feature vectors that are assigned to the r -th mixture component

- b. Mean estimates

$$m_r^{(j+1)} = \frac{1}{n_r} \sum_{i=1}^n w_{ir} x_i$$

- c. Covariance estimate

$$\Sigma_r^{(j+1)} = (\sigma_{r,hg}^{(j+1)})$$

of size $p \times p$ with

$$\sigma_{r,hg}^{(j+1)} = \frac{1}{n_r} \sum_{l=1}^n w_{lr} (x_{lh} - m_{r,h}^{(j+1)})(x_{lg} - m_{r,g}^{(j+1)})$$

4. Repeat steps 2 and 3 until any of these conditions is met:
 - $|\log(\theta^{(j+1)}) - \log(\theta^{(j)})| < \epsilon$,

where the likelihood function is

$$\log(\theta) = \sum_{i=1}^n \log \left(\sum_{j=1}^k p d(x_i | z_j, \theta_j) \alpha_j \right).$$

- The number of iterations exceeds the predefined level.

Initialization

The EM algorithm for GMM requires initialized vector of weights, vectors of means, and variance-covariance [Biernacki2003, Maitra2009].

The EM initialization algorithm for GMM includes the following steps:

1. Perform $nTrials$ starts of the EM algorithm with $nIterations$ iterations and start values:
 - Initial means - k different random observations from the input data set
 - Initial weights - the values of $1/k$
 - Initial covariance matrices - the covariance of the input data
2. Regard the result of the best EM algorithm in terms of the likelihood function values as the result of initialization

Initialization

The EM algorithm for GMM requires initialized vector of weights, vectors of means, and variance-covariance. Skip the initialization step if you already calculated initial weights, means, and covariance matrices.

Batch Processing

Algorithm Input

The EM for GMM initialization algorithm accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
<code>data</code>	Pointer to the $n \times p$ numeric table with the data to which the EM initialization algorithm is applied. The input can be an object of any class derived from <code>NumericTable</code> .

Algorithm Parameters

The EM for GMM initialization algorithm has the following parameters:

Parameter	Default Value	Description
<code>algorithmFPType</code>	<code>double</code>	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<code>method</code>	<code>defaultDense</code>	Performance-oriented computation method, the only method supported by the algorithm.
<code>nComponents</code>	Not applicable	The number of components in the Gaussian Mixture Model, a required parameter.
<code>nTrials</code>	20	The number of starts of the EM algorithm.
<code>nIterations</code>	10	The maximal number of iterations in each start of the EM algorithm.
<code>seed</code>	777	The seed value for the random number generator to get the initial means in each EM start.
<code>accuracyThreshold</code>	1.0e-04	The threshold for termination of the algorithm.
<code>covarianceStorage</code>	<code>full</code>	Covariance matrix storage scheme in the Gaussian Mixture Model: <ul style="list-style-type: none"> <code>full</code> - covariance matrices are stored as numeric tables of size $p \times p$. All elements of the matrix are updated during the processing. <code>diagonal</code> - covariance matrices are stored as numeric tables of size $1 \times p$. Only diagonal elements of the matrix are updated during the processing, and the rest are assumed to be zero.

Algorithm Output

The EM for GMM initialization algorithm calculates the results described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
weights	Pointer to the $1 \times k$ numeric table with mixture weights. By default, this result is an object of the <code>HomogenNumericTable</code> class, but you can define the result as an object of any class derived from <code>NumericTable</code> except <code>PackedTriangularMatrix</code> , <code>PackedSymmetricMatrix</code> , and <code>CSRNumericTable</code> .
means	Pointer to the $k \times p$ numeric table with each row containing the estimate of the means for the i -th mixture component, where $i=0, 1, \dots, k-1$. By default, this result is an object of the <code>HomogenNumericTable</code> class, but you can define the result as an object of any class derived from <code>NumericTable</code> except <code>PackedTriangularMatrix</code> , <code>PackedSymmetricMatrix</code> , and <code>CSRNumericTable</code> .
covariances	<p>Pointer to the <code>DataCollection</code> object that contains k numeric tables, each with the $p \times p$ variance-covariance matrix for the i-th mixture component of size:</p> <ul style="list-style-type: none"> $p \times p$ - for the full covariance matrix storage scheme $1 \times p$ - for the diagonal covariance matrix storage scheme <p>By default, the collection contains objects of the <code>HomogenNumericTable</code> class, but you can define them as objects of any class derived from <code>NumericTable</code> except <code>PackedTriangularMatrix</code> and <code>CSRNumericTable</code>.</p>

Computation

Batch Processing

Algorithm Input

The EM for GMM algorithm accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
data	Pointer to the $n \times p$ numeric table with the data to which the EM algorithm is applied. The input can be an object of any class derived from <code>NumericTable</code> .
inputWeights	Pointer to the $1 \times k$ numeric table with initial mixture weights. This input can be an object of any class derived from <code>NumericTable</code> .
inputMeans	Pointer to a $k \times p$ numeric table. Each row in this table contains the initial value of the means for the i -th mixture component, where $i=0, 1, \dots, k-1$. This input can be an object of any class derived from <code>NumericTable</code> .
inputCovariances	Pointer to the <code>DataCollection</code> object that contains k numeric tables, each with the $p \times p$ variance-covariance matrix for the i -th mixture component of size:

Input ID	Input
	<ul style="list-style-type: none"> • $p \times p$ - for the full covariance matrix storage scheme • $1 \times p$ - for the diagonal covariance matrix storage scheme <p>The collection can contain objects of any class derived from <code>NumericTable</code>.</p>
<code>inputValues</code>	<p>Pointer to the result of the EM for GMM initialization algorithm. The result of initialization contains weights, means, and a collection of covariances. You can use this input to set the initial values for the EM for GMM algorithm instead of explicitly specifying the weights, means, and covariance collection.</p>

Algorithm Parameters

The EM for GMM algorithm has the following parameters:

Parameter	Default Value	Description
<code>algorithmFPType</code>	<code>double</code>	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<code>method</code>	<code>defaultDense</code>	Performance-oriented computation method, the only method supported by the algorithm.
<code>nComponents</code>	Not applicable	The number of components in the Gaussian Mixture Model, a required parameter.
<code>maxIterations</code>	10	The maximal number of iterations in the algorithm.
<code>accuracyThreshold</code>	1.0e-04	The threshold for termination of the algorithm.
<code>covariance</code>	Pointer to an object of the <code>BatchIface</code> class	Pointer to the algorithm that computes the covariance matrix. By default, the respective Intel DAAL algorithm is used, implemented in the class derived from <code>BatchIface</code> .
<code>regularizationFactor</code>	0.01	Factor for covariance regularization in the case of ill-conditional data.
<code>covarianceStorage</code>	<code>full</code>	<p>Covariance matrix storage scheme in the Gaussian Mixture Model:</p> <ul style="list-style-type: none"> • <code>full</code> - covariance matrices are stored as numeric tables of size $p \times p$. All elements of the matrix are updated during the processing. • <code>diagonal</code> - covariance matrices are stored as numeric tables of size $1 \times p$. Only diagonal elements of the matrix are updated during the processing, and the rest are assumed to be zero.

Algorithm Output

The EM for GMM algorithm calculates the results described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
weights	Pointer to the $1 \times k$ numeric table with mixture weights. By default, this result is an object of the <code>HomogenNumericTable</code> class, but you can define the result as an object of any class derived from <code>NumericTable</code> except <code>PackedTriangularMatrix</code> , <code>PackedSymmetricMatrix</code> , and <code>CSRNumericTable</code> .
means	Pointer to the $k \times p$ numeric table with each row containing the estimate of the means for the i -th mixture component, where $i=0, 1, \dots, k-1$. By default, this result is an object of the <code>HomogenNumericTable</code> class, but you can define the result as an object of any class derived from <code>NumericTable</code> except <code>PackedTriangularMatrix</code> , <code>PackedSymmetricMatrix</code> , and <code>CSRNumericTable</code> .
covariances	<p>Pointer to the <code>DataCollection</code> object that contains k numeric tables, each with the $p \times p$ variance-covariance matrix for the i-th mixture component of size:</p> <ul style="list-style-type: none"> $p \times p$ - for the full covariance matrix storage scheme $1 \times p$ - for the diagonal covariance matrix storage scheme <p>By default, the collection contains objects of the <code>HomogenNumericTable</code> class, but you can define them as objects of any class derived from <code>NumericTable</code> except <code>PackedTriangularMatrix</code> and <code>CSRNumericTable</code>.</p>
goalFunction	Pointer to the 1×1 numeric table with the value of the logarithm of the likelihood function after the last iteration. By default, this result is an object of the <code>HomogenNumericTable</code> class.
nIterations	Pointer to the 1×1 numeric table with the number of iterations computed after completion of the algorithm. By default, this result is an object of the <code>HomogenNumericTable</code> class.

Examples

Refer to the following examples in your Intel DAAL directory:

C++: `./examples/cpp/source/em/em_gmm_batch.cpp`

Java*: `./examples/java/source/com/intel/daal/examples/em/EmGmmBatch.java`

Python*: `./examples/python/source/em/em_gmm_batch.py`

Performance Considerations

To get the best overall performance of the expectation-maximization algorithm at the initialization and computation stages:

- If input data is homogeneous, provide the input data and store results in homogeneous numeric tables of the same type as specified in the `algorithmFPTType` class template parameter.
- If input data is non-homogeneous, use AOS layout rather than SOA layout.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Multivariate Outlier Detection

Outlier detection methods aim to identify observation points that are abnormally distant from other observation points. In multivariate outlier detection methods, the observation point is the entire feature vector.

Details

Given a set X of n feature vectors $x_1 = (x_{11}, \dots, x_{1p})$, ..., $x_n = (x_{n1}, \dots, x_{np})$ of dimension p , the problem is to identify the vectors that do not belong to the underlying distribution (see [Ben05] for exact definitions of an outlier).

The multivariate outlier detection method takes into account dependencies between features. This method can be parametric, assumes a known underlying distribution for the data set, and defines an outlier region such that if an observation belongs to the region, it is marked as an outlier. Definition of the outlier region is connected to the assumed underlying data distribution. The following is an example of an outlier region for multivariate outlier detection:

$$Outlier(\alpha_n, M_n, \Sigma_n) = \left\{ x : \sqrt{(x - M_n) \Sigma_n^{-1} (x - M_n)} > g(n, \alpha_n) \right\},$$

where M_n and Σ_n are (robust) estimates of the vector of means and variance-covariance matrix computed for a given data set, α_n is the confidence coefficient, and $g(n, \alpha_n)$ defines the limit of the region.

Batch Processing**Algorithm Input**

The multivariate outlier detection algorithm accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
data	Pointer to the $n \times p$ numeric table with the data for outlier detection. The input can be an object of any class derived from the <code>NumericTable</code> class.

Algorithm Parameters

The multivariate outlier detection algorithm has the following parameters, which depend on the computation method parameter *method*:

Parameter	method	Default Value	Description
<i>algorithmFPTYPE</i>	defaultDense or baconDense	double	The floating-point type that the algorithm uses for intermediate computations. Can be float or double.
<i>method</i>	Not applicable	defaultDense	<p>Available methods for multivariate outlier detection:</p> <ul style="list-style-type: none"> • defaultDense - Performance-oriented computation method • baconDense - Blocked Adaptive Computationally-efficient Outlier Nominators (BACON) method
<i>initialization Procedure</i>	defaultDense	Not applicable	<p>The procedure for setting initial parameters of the algorithm. It is your responsibility to define the procedure.</p> <p>Input objects for the initialization procedure are:</p> <ul style="list-style-type: none"> • data - numeric table of size $n \times p$ that contains input data of the multivariate outlier detection algorithm <p>Results of the initialization procedure are:</p> <ul style="list-style-type: none"> • location - numeric table of size $1 \times p$ that contains the vector of means • scatter - numeric table of size $p \times p$ that contains the variance-covariance matrix • threshold - numeric table of size 1×1 with the non-negative number that defines the outlier region <p>If you do not set this parameter, the library uses the default initialization, which sets:</p> <ul style="list-style-type: none"> • location to 0.0 • scatter to the numeric table with diagonal elements equal to 1.0 and non-diagonal elements equal to 0.0 • threshold to 3.0

Parameter	method	Default Value	Description
	baconDense	baconMedian	The initialization method. Can be: <ul style="list-style-type: none"> baconMedian - Median-based method. defaultDense - Mahalanobis distance-based method.
<i>alpha</i>	baconDense	0.05	One-tailed probability that defines the $(1 - \alpha)$ quantile of the χ^2 distribution with p degrees of freedom. Recommended value: α/n , where n is the number of observations.
<i>accuracyThreshold</i>	baconDense	0.005	The stopping criterion. The algorithm is terminated if the size of the basic subset is changed by less than the threshold.

Algorithm Output

The multivariate outlier detection algorithm calculates the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
weights	Pointer to the $n \times 1$ numeric table of zeros and ones. Zero in the i -th position indicates that the i -th feature vector is an outlier. By default, the result is an object of the <code>HomogenNumericTable</code> class, but you can define the result as an object of any class derived from <code>NumericTable</code> except the <code>PackedSymmetricMatrix</code> , <code>PackedTriangularMatrix</code> , and <code>CSRNumericTable</code> .

Examples

Refer to the following examples in your Intel DAAL directory:

C++:

- `./examples/cpp/source/outlier_detection/outlier_detection_multivariate_default_batch.cpp`
- `./examples/cpp/source/outlier_detection/outlier_detection_multivariate_bacon_batch.cpp`

Java*:

- `./examples/java/source/com/intel/daal/examples/outlier_detection/OutlierDetectionMultivariateDefaultBatch.java`
- `./examples/java/source/com/intel/daal/examples/outlier_detection/OutlierDetectionMultivariateBaconBatch.java`

Python*:

- `./examples/python/source/outlier_detection/outlier_detection_multivariate_default_batch.py`
- `./examples/python/source/outlier_detection/outlier_detection_multivariate_bacon_batch.py`

Performance Considerations

To get the best overall performance of multivariate outlier detection:

- If input data is homogeneous, provide input data and store results in homogeneous numeric tables of the same type as specified in the `algorithmFPType` class template parameter.
- If input data is non-homogeneous, use AOS layout rather than SOA layout.
- For the default outlier detection method (`defaultDense`), you can benefit from splitting the input data set into blocks for parallel processing.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Univariate Outlier Detection

Outlier detection methods aim to identify observation points that are abnormally distant from other observation points. A univariate outlier is an occurrence of an abnormal value within a single observation point.

Details

Given a set X of n feature vectors $x_1 = (x_{11}, \dots, x_{1p})$, ..., $x_n = (x_{n1}, \dots, x_{np})$ of dimension p , the problem is to identify the vectors that do not belong to the underlying distribution (see [Ben05] for exact definitions of an outlier).

The algorithm for univariate outlier detection considers each feature independently. The univariate outlier detection method can be parametric, assumes a known underlying distribution for the data set, and defines an outlier region such that if an observation belongs to the region, it is marked as an outlier. Definition of the outlier region is connected to the assumed underlying data distribution. The following is an example of an outlier region for the univariate outlier detection:

$$Outlier(\alpha_n, m_n, \sigma_n) = \left\{ x: \frac{|x - m_n|}{\sigma_n} > g(n, \alpha_n) \right\},$$

where m_n and σ_n are (robust) estimates of the mean and standard deviation computed for a given data set, α_n is the confidence coefficient, and $g(n, \alpha_n)$ defines the limits of the region and should be adjusted to the number of observations.

Batch Processing

Algorithm Input

The univariate outlier detection algorithm accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
<code>data</code>	Pointer to the $n \times p$ numeric table with the data for outlier detection. The input can be an object of any class derived from the <code>NumericTable</code> class.

Algorithm Parameters

The univariate outlier detection algorithm has the following parameters:

Parameter	Default Value	Description
<code>algorithmFPType</code>	<code>double</code>	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<code>method</code>	<code>defaultDense</code>	Performance-oriented computation method, the only method supported by the algorithm.
<code>initialization Procedure</code>	Not applicable	<p>The procedure for setting initial parameters of the algorithm. It is your responsibility to define the procedure.</p> <p>Input objects for the initialization procedure are:</p> <ul style="list-style-type: none"> <code>data</code> - numeric table of size $n \times p$ that contains input data of the univariate outlier detection algorithm <p>Results of the initialization procedure are:</p> <ul style="list-style-type: none"> <code>location</code> - numeric table of size $1 \times p$ that contains the vector of means <code>scatter</code> - numeric table of size $1 \times p$ that contains the vector of deviations <code>threshold</code> - numeric table of size $1 \times p$ with the non-negative numbers that define the outlier region <p>If you do not set this parameter, the library uses the default initialization, which sets:</p> <ul style="list-style-type: none"> <code>location</code> to 0.0 <code>scatter</code> to 1.0 <code>threshold</code> to 3.0

Algorithm Output

The univariate outlier detection algorithm calculates the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
weights	Pointer to the $n \times p$ numeric table of zeros and ones. Zero in the position (i, j) indicates an outlier in the i -th observation of the j -th feature. By default, the result is an object of the <code>HomogenNumericTable</code> class, but you can define the result as an object of any class derived from <code>NumericTable</code> except <code>PackedSymmetricMatrix</code> , <code>PackedTriangularMatrix</code> , and <code>CSRNumericTable</code> .

Examples

Refer to the following examples in your Intel DAAL directory:

C++: `./examples/cpp/source/outlier_detection/outlier_detection_univariate_batch.cpp`

Java*: `./examples/java/source/com/intel/daal/examples/outlier_detection/OutlierDetectionUnivariateBatch.java`

Python*: `./examples/python/source/outlier_detection/outlier_detection_univariate_batch.py`

Performance Considerations

To get the best overall performance of univariate outlier detection:

- If input data is homogeneous, provide input data and store results in homogeneous numeric tables of the same type as specified in the `algorithmFPType` class template parameter.
- If input data is non-homogeneous, use AOS layout rather than SOA layout.
- You can benefit from splitting the input data set into blocks for parallel processing.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Kernel Functions

Kernel functions form a class of algorithms for pattern analysis. The main characteristic of kernel functions is a distinct approach to this problem. Instead of reducing the dimension of the original data, kernel functions map the data into higher-dimensional spaces in order to make the data more easily separable there.

Linear Kernel

A linear kernel is the simplest kernel function.

Problem Statement

Given a set X of n feature vectors $x_1 = (x_{11}, \dots, x_{1p})$, ..., $x_n = (x_{n1}, \dots, x_{np})$ of dimension p and a set Y of m feature vectors $y_1 = (y_{11}, \dots, y_{1p})$, ..., $y_m = (y_{m1}, \dots, y_{mp})$, the problem is to compute the linear kernel function $K(x_i, y_j)$ for any pair of input vectors: $K(x_i, y_j) = k x_i^T y_j + b$.

Batch Processing

Algorithm Input

The linear kernel function accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
X	Pointer to the $n \times p$ numeric table that represents the matrix X . This table can be an object of any class derived from <code>NumericTable</code> .
Y	Pointer to the $m \times p$ numeric table that represents the matrix Y . This table can be an object of any class derived from <code>NumericTable</code> .

Algorithm Parameters

The linear kernel function has the following parameters:

Parameter	Default Value	Description
<code>algorithmFPType</code>	<code>double</code>	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<code>method</code>	<code>defaultDense</code>	Available computation methods: <ul style="list-style-type: none"> <code>defaultDense</code> - performance-oriented method <code>fastCSR</code> - performance-oriented method for CSR numeric tables
<code>ComputationMode</code>	<code>matrixMatrix</code>	Computation mode for the kernel function. Can be: <ul style="list-style-type: none"> <code>vectorVector</code> - compute the kernel function for given feature vectors x_i and y_j <code>matrixVector</code> - compute the kernel function for all vectors in the set X and a given feature vector y_j <code>matrixMatrix</code> - compute the kernel function for all vectors in the sets X and Y. In Intel DAAL, this mode requires equal numbers of observations in both input tables: $n=m$.
<code>rowIndexX</code>	0	Index i of the vector in the set X for the <code>vectorVector</code> computation mode.
<code>rowIndexY</code>	0	Index j of the vector in the set Y for the <code>vectorVector</code> or <code>matrixVector</code> computation mode.
<code>rowIndexResult</code>	0	Row index in the <code>values</code> numeric table to locate the result of the computation for the <code>vectorVector</code> computation mode.

Parameter	Default Value	Description
k	1	The coefficient k of the linear kernel.
b	0	The coefficient b of the linear kernel.

Algorithm Output

The linear kernel function calculates the results described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
values	Pointer to the $n \times m$ numeric table with the values of the kernel function. By default, this result is an object of the <code>HomogenNumericTable</code> class, but you can define the result as an object of any class derived from <code>NumericTable</code> except <code>PackedSymmetricMatrix</code> , <code>PackedTriangularMatrix</code> , and <code>CSRNumericTable</code> .

Examples

Refer to the following examples in your Intel DAAL directory:

C++:

- `./examples/cpp/source/kernel_function/kernel_function_linear_dense_batch.cpp`
- `./examples/cpp/source/kernel_function/kernel_function_linear_csr_batch.cpp`

Java*:

- `./examples/java/source/com/intel/daal/examples/kernel_function/KernelFunctionLinearDenseBatch.java`
- `./examples/java/source/com/intel/daal/examples/kernel_function/KernelFunctionLinearCSRBatch.java`

Python*:

- `./examples/python/source/kernel_function/kernel_function_linear_dense_batch.py`
- `./examples/python/source/kernel_function/kernel_function_linear_csr_batch.py`

Radial Basis Function Kernel

The Radial Basis Function (RBF) kernel is a popular kernel function used in kernelized learning algorithms.

Problem Statement

Given a set X of n feature vectors $x_1 = (x_{11}, \dots, x_{1p})$, ..., $x_n = (x_{n1}, \dots, x_{np})$ of dimension p and a set Y of m feature vectors $y_1 = (y_{11}, \dots, y_{1p})$, ..., $y_m = (y_{m1}, \dots, y_{mp})$, the problem is to compute the RBF kernel function $K(x_i, y_j)$ for any pair of input vectors:

$$K(x_i, y_j) = \exp\left(-\frac{\|x_i - y_j\|^2}{2\sigma^2}\right).$$

Batch Processing

Algorithm Input

The RBF kernel accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
X	Pointer to the $n \times p$ numeric table that represents the matrix X . This table can be an object of any class derived from <code>NumericTable</code> .
Y	Pointer to the $m \times p$ numeric table that represents the matrix Y . This table can be an object of any class derived from <code>NumericTable</code> .

Algorithm Parameters

The RBF kernel has the following parameters:

Parameter	Default Value	Description
<code>algorithmFPType</code>	<code>double</code>	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<code>method</code>	<code>defaultDense</code>	Available computation methods: <ul style="list-style-type: none"> <code>defaultDense</code> - performance-oriented method <code>fastCSR</code> - performance-oriented method for CSR numeric tables
<code>ComputationMode</code>	<code>matrixMatrix</code>	Computation mode for the kernel function. Can be: <ul style="list-style-type: none"> <code>vectorVector</code> - compute the kernel function for given feature vectors x_i and y_j <code>matrixVector</code> - compute the kernel function for all vectors in the set X and a given feature vector y_j <code>matrixMatrix</code> - compute the kernel function for all vectors in the sets X and Y. In Intel DAAL, this mode requires equal numbers of observations in both input tables: $n=m$.
<code>rowIndexX</code>	0	Index i of the vector in the set X for the <code>vectorVector</code> computation mode.
<code>rowIndexY</code>	0	Index j of the vector in the set Y for the <code>vectorVector</code> or <code>matrixVector</code> computation mode.
<code>rowIndexResult</code>	0	Row index in the <code>values</code> numeric table to locate the result of the computation for the <code>vectorVector</code> computation mode.

Parameter	Default Value	Description
<i>sigma</i>	0	The coefficient σ of the RBF kernel.

Algorithm Output

The RBF kernel calculates the results described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
values	Pointer to the $n \times m$ numeric table with the values of the kernel function. By default, this result is an object of the <code>HomogenNumericTable</code> class, but you can define the result as an object of any class derived from <code>NumericTable</code> except <code>PackedSymmetricMatrix</code> , <code>PackedTriangularMatrix</code> , and <code>CSRNumericTable</code> .

Examples

Refer to the following examples in your Intel DAAL directory:

C++:

- `./examples/cpp/source/kernel_function/kernel_function_rbf_dense_batch.cpp`
- `./examples/cpp/source/kernel_function/kernel_function_rbf_csr_batch.cpp`

Java*:

- `./examples/java/source/com/intel/daal/examples/kernel_function/KernelFunctionRbfDenseBatch.java`
- `./examples/java/source/com/intel/daal/examples/kernel_function/KernelFunctionRbfCSRBatch.java`

Python*:

- `./examples/python/source/kernel_function/kernel_function_rbf_dense_batch.py`
- `./examples/python/source/kernel_function/kernel_function_rbf_csr_batch.py`

Quality Metrics

In Intel® Data Analytics Acceleration Library (Intel® DAAL), a quality metric is a numerical characteristic or a set of connected numerical characteristics that represents the qualitative aspect of the result returned by an algorithm: a computed statistical estimate, model, or result of decision making.

Although quality metrics are algorithm-specific, a common set of quality metrics can be defined for some training and prediction algorithms.

Quality metrics are optional. They are computed when the computation is explicitly requested.

See Also

[Classification](#)

[Linear Regression](#)

Quality Metrics for Binary Classification Algorithms

For two classes C_1 and C_2 , given a vector $X = (x_1, \dots, x_n)$ of class labels computed at the prediction stage of the classification algorithm and a vector $Y = (y_1, \dots, y_n)$ of expected class labels, the problem is to evaluate the classifier by computing the confusion matrix and connected quality metrics: precision, recall, and so on.

Further definitions use the following notations:

tp (true positive)	the number of correctly recognized observations for class C_1
tn (true negative)	the number of correctly recognized observations that do not belong to the class C_1
fp (false positive)	the number of observations that were incorrectly assigned to the class C_1
fn (false negative)	the number of observations that were not recognized as belonging to the class C_1

The library uses the following quality metrics for binary classifiers:

Quality Metric	Definition
Accuracy	$\frac{tp + tn}{tp + fn + fp + tn}$
Precision	$\frac{tp}{tp + fp}$
Recall	$\frac{tp}{tp + fn}$
F-score	$\frac{(\beta^2 + 1)tp}{(\beta^2 + 1)tp + \beta^2 fn + fp}$
Specificity	$\frac{tn}{fp + tn}$
Area under curve (AUC)	$\frac{1}{2} \left(\frac{tp}{tp + fn} + \frac{tp}{fp + tn} \right)$

For more details of these metrics, including the evaluation focus, refer to [Sokolova09].

The confusion matrix is defined as follows:

	Classified as Class C_1	Classified as Class C_2
Actual Class C_1	tp	fn
Actual Class C_2	fp	tn

Batch Processing

Algorithm Input

The quality metric algorithm for binary classifiers accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
<code>predictedLabels</code>	Pointer to the $n \times 1$ numeric table that contains labels computed at the prediction stage of the classification algorithm. This input can be an object of any class derived from <code>NumericTable</code> except <code>PackedSymmetricMatrix</code> , <code>PackedTriangularMatrix</code> , and <code>CSRNumericTable</code> .
<code>groundTruthLabels</code>	Pointer to the $n \times 1$ numeric table that contains expected labels. This input can be an object of any class derived from <code>NumericTable</code> except <code>PackedSymmetricMatrix</code> , <code>PackedTriangularMatrix</code> , and <code>CSRNumericTable</code> .

Algorithm Parameters

The quality metric algorithm has the following parameters:

Parameter	Default Value	Description
<code>algorithmFPType</code>	<code>double</code>	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<code>method</code>	<code>defaultDense</code>	Performance-oriented computation method, the only method supported by the algorithm.
<code>useDefaultMetrics</code>	<code>true</code>	A flag that defines a need to compute the default metrics provided by the library.
<code>beta</code>	<code>1</code>	The β parameter of the F-score quality metric provided by the library.

Algorithm Output

The quality metric algorithm calculates the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
<code>confusionMatrix</code>	Pointer to the 2 x 2 numeric table with the confusion matrix. By default, this result is an object of the <code>HomogenNumericTable</code> class, but you can define the result as an object of any class derived from <code>NumericTable</code> except <code>PackedTriangularMatrix</code> , <code>PackedSymmetricMatrix</code> , and <code>CSRNumericTable</code> .
<code>binaryMetrics</code>	<p>Pointer to the 1 x 6 numeric table that contains quality metrics, which you can access by an appropriate Binary Metrics ID:</p> <ul style="list-style-type: none"> <code>accuracy</code> - accuracy <code>precision</code> - precision <code>recall</code> - recall <code>fscore</code> - F-score <code>specificity</code> - specificity <code>AUC</code> - area under the curve <p>By default, this result is an object of the <code>HomogenNumericTable</code> class, but you can define the result as an object of any class derived from <code>NumericTable</code> except <code>PackedTriangularMatrix</code>, <code>PackedSymmetricMatrix</code>, and <code>CSRNumericTable</code>.</p>

Examples

Refer to the following examples in your Intel DAAL directory:

C++: `./examples/cpp/source/quality_metrics/svm_two_class_quality_metric_set_batch.cpp`

Java*: `./examples/java/source/com/intel/daal/examples/quality_metrics/SVMTwoClassQualityMetricSetBatchExample.java`

Python*: `./examples/python/source/quality_metrics/svm_two_class_quality_metric_set_batch.py`

Quality Metrics for Multi-class Classification Algorithms

For l classes C_1, \dots, C_l , given a vector $X = (x_1, \dots, x_n)$ of class labels computed at the prediction stage of the classification algorithm and a vector $Y = (y_1, \dots, y_n)$ of expected class labels, the problem is to evaluate the classifier by computing the confusion matrix and connected quality metrics: precision, error rate, and so on.

Further definitions use the following notations:

tp_i (true positive)	the number of correctly recognized observations for class C_i
tn_i (true negative)	the number of correctly recognized observations that do not belong to the class C_i
fp_i (false positive)	the number of observations that were incorrectly assigned to the class C_i
fn_i (false negative)	the number of observations that were not recognized as belonging to the class C_i

The library uses the following quality metrics for multi-class classifiers:

Quality Metric	Definition
Average accuracy	$\frac{\sum_{i=1}^l \frac{tp_i + tn_i}{tp_i + fn_i + fp_i + tn_i}}{l}$
Error rate	$\frac{\sum_{i=1}^l \frac{fp_i + fn_i}{tp_i + fn_i + fp_i + tn_i}}{l}$
Micro precision ($Precision_\mu$)	$\frac{\sum_{i=1}^l tp_i}{\sum_{i=1}^l (tp_i + fp_i)}$
Micro recall ($Recall_\mu$)	$\frac{\sum_{i=1}^l tp_i}{\sum_{i=1}^l (tp_i + fn_i)}$
Micro F-score ($F-score_\mu$)	$\frac{(\beta^2 + 1) \left(Precision_\mu \times Recall_\mu \right)}{\beta^2 \times Precision_\mu \times Recall_\mu}$
Macro precision ($Precision_M$)	$\frac{\sum_{i=1}^l \frac{tp_i}{tp_i + fp_i}}{l}$
Macro recall ($Recall_M$)	$\frac{\sum_{i=1}^l \frac{tp_i}{tp_i + fn_i}}{l}$
Macro F-score ($F-score_M$)	$\frac{(\beta^2 + 1) \left(Precision_M \times Recall_M \right)}{\beta^2 \times Precision_M \times Recall_M}$

For more details of these metrics, including the evaluation focus, refer to [Sokolova09].

The following is the confusion matrix:

	Classified as Class C_1	...	Classified as Class C_i	...	Classified as Class C_l
Actual Class C_1	c_{11}	...	c_{1i}	...	c_{1l}

...
Actual Class C_j	c_{j1}	...	c_{ij}	...	c_{jl}
...
Actual Class C_l	c_{l1}	...	c_{li}	...	c_{ll}

The positives and negatives are defined through elements of the confusion matrix as follows:

$$tp_i = c_{ii};$$

$$fp_i = \sum_{n=1}^l c_{ni} - tp_i;$$

$$fn_i = \sum_{n=1}^l c_{in} - tp_i;$$

$$tn_i = \sum_{n=1}^l \sum_{k=1}^l c_{nk} - tp_i - fp_i - fn_i.$$

Batch Processing

Algorithm Input

The quality metric algorithm for multi-class classifiers accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
predictedLabels	Pointer to the $n \times 1$ numeric table that contains labels computed at the prediction stage of the classification algorithm. This input can be an object of any class derived from <code>NumericTable</code> except <code>PackedSymmetricMatrix</code> , <code>PackedTriangularMatrix</code> , and <code>CSRNumericTable</code> .
groundTruthLabels	Pointer to the $n \times 1$ numeric table that contains expected labels. This input can be an object of any class derived from <code>NumericTable</code> except <code>PackedSymmetricMatrix</code> , <code>PackedTriangularMatrix</code> , and <code>CSRNumericTable</code> .

Algorithm Parameters

The quality metric algorithm has the following parameters:

Parameter	Default Value	Description
<code>algorithmFPType</code>	double	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<code>method</code>	defaultDense	Performance-oriented computation method, the only method supported by the algorithm.
<code>nClasses</code>	0	The number of classes (l).
<code>useDefaultMetrics</code>	true	A flag that defines a need to compute the default metrics provided by the library.

Parameter	Default Value	Description
<i>beta</i>	1	The β parameter of the F-score quality metric provided by the library.

Algorithm Output

The quality metric algorithm calculates the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
<code>confusionMatrix</code>	Pointer to the $nClasses \times nClasses$ numeric table with the confusion matrix. By default, this result is an object of the <code>HomogenNumericTable</code> class, but you can define the result as an object of any class derived from <code>NumericTable</code> except <code>PackedTriangularMatrix</code> , <code>PackedSymmetricMatrix</code> , and <code>CSRNumericTable</code> .
<code>multiClassMetrics</code>	<p>Pointer to the 1 x 8 numeric table that contains quality metrics, which you can access by an appropriate Multi-class Metrics ID:</p> <ul style="list-style-type: none"> • <code>averageAccuracy</code> - average accuracy • <code>errorRate</code> - error rate • <code>microPrecision</code> - micro precision • <code>microRecall</code> - micro recall • <code>microFscore</code> - micro F-score • <code>macroPrecision</code> - macro precision • <code>macroRecall</code> - macro recall • <code>macroFscore</code> - macro F-score <p>By default, this result is an object of the <code>HomogenNumericTable</code> class, but you can define the result as an object of any class derived from <code>NumericTable</code> except <code>PackedTriangularMatrix</code>, <code>PackedSymmetricMatrix</code>, and <code>CSRNumericTable</code>.</p>

Examples

Refer to the following examples in your Intel DAAL directory:

C++: `./examples/cpp/source/quality_metrics/svm_multi_class_quality_metric_set_batch.cpp`

Java*: `./examples/java/source/com/intel/daal/examples/quality_metrics/SVMMultiClassQualityMetricSetBatchExample.java`

Python*: `./examples/python/source/quality_metrics/svm_multi_class_quality_metric_set_batch.py`

Quality Metrics for Linear Regression

Given a data set $X = (x_i)$ that contains vectors of input variables $x_i = (x_{i1}, \dots, x_{ip})$, respective responses $z_i = (z_{i1}, \dots, z_{ik})$ computed at the prediction stage of the linear regression model defined by its coefficients β_{ht} , $h = 1, \dots, k$, $t = 1, \dots, p$, and expected responses $y_i = (y_{i1}, \dots, y_{ik})$, $i = 1, \dots, n$, the problem is to evaluate the linear regression model by computing the root mean square error, variance-covariance matrix of beta coefficients, various statistics functions, and so on. See [Linear Regression](#) for additional details and notations.

For linear regressions, the library computes statistics listed in tables below for testing insignificance of beta coefficients:

- [For a single coefficient](#)
- [For a group of coefficients](#)

The statistics are computed given the following assumptions about the data distribution:

- Responses y_{ij} , $i = 1, \dots, n$, are independent and have a constant variance σ_j^2 , $j = 1, \dots, k$
- Conditional expectation of responses y_{ij} , $j = 1, \dots, k$, is linear in input variables $x_i = (x_{i,1}, \dots, x_{i,p})$
- Deviations of y_{ij} , $i = 1, \dots, n$, around the mean of expected responses ERM_j , $j = 1, \dots, k$, are additive and Gaussian.

For more details, see [\[Hastie2009\]](#).

Testing Insignificance of a Single Beta

The library uses the following quality metrics:

Quality Metric	Definition
Root Mean Square (RMS) Error	$\sqrt{\frac{1}{n} \sum_{i=1}^n (y_{ij} - z_{ij})^2}, \quad j = 1, \dots, k$
Vector of variances $\sigma^2 = (\sigma_1^2, \dots, \sigma_k^2)$	$\sigma_j^2 = \frac{1}{n-p-1} \sum_{i=1}^n (y_{ij} - z_{ij})^2, \quad j = 1, \dots, k$
A set of variance-covariance matrices $C = C_1, \dots, C_k$ for vectors of betas β_{jt} , $j = 1, \dots, k$	$C_j = (X^T X)^{-1} \sigma_j^2, \quad j = 1, \dots, k$
Z-score statistics used in testing of insignificance of a single coefficient β_{jt}	$zscore_{jt} = \frac{\beta_{jt}}{\sigma_j \sqrt{v_t}}, \quad j = 1, \dots, k,$ <p>σ_j is the j-th element of the vector of variance σ^2 and v_t is the t-th diagonal element of the matrix $(X^T X)^{-1}$</p>
Confidence interval for β_{jt}	$(\beta_{jt} - pc_{1-\alpha} \sqrt{v_t} \sigma_j, \beta_{jt} + pc_{1-\alpha} \sqrt{v_t} \sigma_j), \quad j = 1, \dots, k,$ <p>$pc_{1-\alpha}$ is the $(1 - \alpha)$ percentile of the Gaussian distribution, σ_j is the j-th element of the vector of variance σ^2, and v_t is the t-th diagonal element of the matrix $(X^T X)^{-1}$</p>

Testing Insignificance of a Group of Betas

The library uses the following quality metrics:

Quality Metric	Definition
Mean of expected responses, $ERM = (ERM_1, \dots, ERM_k)$	$ERM_j = \frac{1}{n} \sum_{i=1}^n y_{ij}, \quad j = 1, \dots, k$
Variance of expected responses, $ERV = (ERV_1, \dots, ERV_k)$	$ERV_j = \frac{1}{n-1} \sum_{i=1}^n (y_{ij} - ERM_j)^2, \quad j = 1, \dots, k$

Quality Metric	Definition
Regression Sum of Squares $RegSS = (RegSS_1, \dots, RegSS_k)$	$RegSS_j = \frac{1}{n} \sum_{i=1}^n (z_{ij} - ERM_j)^2, \quad j = 1, \dots, k$
Sum of Squares of Residuals $ResSS = (ResSS_1, \dots, ResSS_k)$	$ResSS_j = \sum_{i=1}^n (y_{ij} - z_{ij})^2, \quad j = 1, \dots, k$
Total Sum of Squares $TSS = (TSS_1, \dots, TSS_k)$	$TSS_j = \sum_{i=1}^n (y_{ij} - ERM_j)^2, \quad j = 1, \dots, k$
Determination Coefficient	$R_j^2 = \frac{RegSS_j}{TSS_j}, \quad j = 1, \dots, k$
$R^2 = (R_1^2, \dots, R_k^2)$	
F-statistics used in testing insignificance of a group of betas $F = (F_1, \dots, F_k)$	$F_j = \frac{(ResSS_{0j} - ResSS_j)/(p - p_0)}{ResSS_j/(n - p - 1)}, \quad j = 1, \dots, k,$ where $ResSS_j$ are computed for a model with $p + 1$ betas and $ResSS_{0j}$ are computed for a reduced model with $p_0 + 1$ betas ($p - p_0$ betas are set to zero)

Batch Processing

Testing Insignificance of a Single Beta

Algorithm Input

The quality metric algorithm for linear regression accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
<code>expectedResponses</code>	Pointer to the $n \times k$ numeric table with responses (k dependent variables) used for training the linear regression model. This table can be an object of any class derived from <code>NumericTable</code> .
<code>model</code>	Pointer to the model computed at the training stage of the linear regression algorithm. The model can only be an object of the <code>linear_regression::Model</code> class.
<code>predictedResponses</code>	Pointer to the $n \times k$ numeric table with responses (k dependent variables) computed at the prediction stage of the linear regression algorithm. This table can be an object of any class derived from <code>NumericTable</code> .

Algorithm Parameters

The quality metric algorithm for linear regression has the following parameters:

Parameter	Default Value	Description
<code>algorithmFPTYPE</code>	<code>float</code>	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<code>method</code>	<code>defaultDense</code>	Performance-oriented computation method, the only method supported by the algorithm.
<code>useDefaultMetrics</code>	<code>true</code>	Flag that defines if the default metrics provided by the library need to be computed.
<code>alpha</code>	<code>0.05</code>	Significance level used in the computation of confidence intervals for coefficients of the linear regression model.
<code>accuracyThreshold</code>	<code>0.001</code>	Values below this threshold are considered equal to it.

Algorithm Output

The quality metric algorithm for linear regression calculates the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
<code>rms</code>	Pointer to the $1 \times k$ numeric table that contains root mean square errors computed for each response (dependent variable). By default, this result is an object of the <code>HomogenNumericTable</code> class, but you can define the result as an object of any class derived from <code>NumericTable</code> , except for <code>PackedTriangularMatrix</code> , <code>PackedSymmetricMatrix</code> , and <code>CSRNumericTable</code> .
<code>variance</code>	Pointer to the $1 \times k$ numeric table that contains variances σ_j^2 , $j = 1, \dots, k$ computed for each response (dependent variable). By default, this result is an object of the <code>HomogenNumericTable</code> class, but you can define the result as an object of any class derived from <code>NumericTable</code> , except for <code>PackedTriangularMatrix</code> , <code>PackedSymmetricMatrix</code> , and <code>CSRNumericTable</code> .
<code>betaCovariances</code>	Pointer to the <code>DataCollection</code> object that contains k numeric tables, each with the $m \times m$ variance-covariance matrix for betas of the j -th response (dependent variable), where m is the number of betas in the model (m is equal to p when <code>interceptFlag</code> is set to <code>false</code> at the training stage of the linear regression algorithm; otherwise, m is equal to $p + 1$). The collection can contain objects of any class derived from <code>NumericTable</code> .
<code>zScore</code>	Pointer to the $k \times m$ numeric table that contains the Z-score statistics used in the testing of insignificance of individual linear regression coefficients, where m is the number of betas in the model (m is equal to p when <code>interceptFlag</code> is set to <code>false</code> at the training stage of the linear regression algorithm; otherwise, m is equal to $p + 1$). By default, this result is an object of the <code>HomogenNumericTable</code> class, but you can define the result as an object of any class derived from <code>NumericTable</code> , except for <code>PackedTriangularMatrix</code> , <code>PackedSymmetricMatrix</code> , and <code>CSRNumericTable</code> .

Result ID	Result
<code>confidenceIntervals</code>	<p>Pointer to the $k \times 2 \times m$ numeric table that contains limits of the confidence intervals for linear regression coefficients:</p> <ul style="list-style-type: none"> <code>confidenceIntervals[t][2*j]</code> is the left limit of the confidence interval computed for the j-th beta of the t-th response (dependent variable) <code>confidenceIntervals[t][2*j+1]</code> is the right limit of the confidence interval computed for the j-th beta of the t-th response (dependent variable), <p>where m is the number of betas in the model (m is equal to p when <code>interceptFlag</code> is set to <code>false</code> at the training stage of the linear regression algorithm; otherwise, m is equal to $p + 1$). By default, this result is an object of the <code>HomogenNumericTable</code> class, but you can define the result as an object of any class derived from <code>NumericTable</code>, except for <code>PackedTriangularMatrix</code>, <code>PackedSymmetricMatrix</code>, and <code>CSRNumericTable</code>.</p>
<code>inverseOfXtX</code>	<p>Pointer to the $m \times m$ numeric table that contains the $(X^T X)^{-1}$ matrix, where m is the number of betas in the model (m is equal to p when <code>interceptFlag</code> is set to <code>false</code> at the training stage of the linear regression algorithm; otherwise, m is equal to $p + 1$).</p>

Testing Insignificance of a Group of Betas

Algorithm Input

The quality metric algorithm for linear regression accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
<code>expectedResponses</code>	Pointer to the $n \times k$ numeric table with responses (k dependent variables) used for training the linear regression model. This table can be an object of any class derived from <code>NumericTable</code> .
<code>predictedResponses</code>	Pointer to the $n \times k$ numeric table with responses (k dependent variables) computed at the prediction stage of the linear regression algorithm. This table can be an object of any class derived from <code>NumericTable</code> .
<code>predictedReducedModelResponses</code>	Pointer to the $n \times k$ numeric table with responses (k dependent variables) computed at the prediction stage of the linear regression algorithm using the reduced linear regression model, where $p - p_0$ out of p beta coefficients are set to zero. This table can be an object of any class derived from <code>NumericTable</code> .

Algorithm Parameters

The quality metric algorithm for linear regression has the following parameters:

Parameter	Default Value	Description
<code>algorithmFPType</code>	<code>float</code>	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .

Parameter	Default Value	Description
<i>method</i>	defaultDense	Performance-oriented computation method, the only method supported by the algorithm.
<i>numBeta</i>	0	Number of beta coefficients used for prediction.
<i>numBetaReducedModel</i>	0	Number of beta coefficients (p_0) used for prediction with the reduced linear regression model, where $p - p_0$ out of p beta coefficients are set to zero.

Algorithm Output

The quality metric algorithm for linear regression calculates the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
<i>expectedMeans</i>	Pointer to the 1 x k numeric table that contains the mean of expected responses computed for each dependent variable. By default, this result is an object of the <code>HomogenNumericTable</code> class, but you can define the result as an object of any class derived from <code>NumericTable</code> , except for <code>PackedTriangularMatrix</code> , <code>PackedSymmetricMatrix</code> , and <code>CSRNumericTable</code> .
<i>expectedVariance</i>	Pointer to the 1 x k numeric table that contains the variance of expected responses computed for each dependent variable. By default, this result is an object of the <code>HomogenNumericTable</code> class, but you can define the result as an object of any class derived from <code>NumericTable</code> , except for <code>PackedTriangularMatrix</code> , <code>PackedSymmetricMatrix</code> , and <code>CSRNumericTable</code> .
<i>regSS</i>	Pointer to the 1 x k numeric table that contains the regression sum of squares computed for each dependent variable. By default, this result is an object of the <code>HomogenNumericTable</code> class, but you can define the result as an object of any class derived from <code>NumericTable</code> , except for <code>PackedTriangularMatrix</code> , <code>PackedSymmetricMatrix</code> , and <code>CSRNumericTable</code> .
<i>resSS</i>	Pointer to the 1 x k numeric table that contains the sum of squares of residuals computed for each dependent variable. By default, this result is an object of the <code>HomogenNumericTable</code> class, but you can define the result as an object of any class derived from <code>NumericTable</code> , except for <code>PackedTriangularMatrix</code> , <code>PackedSymmetricMatrix</code> , and <code>CSRNumericTable</code> .
<i>tSS</i>	Pointer to the 1 x k numeric table that contains the total sum of squares computed for each dependent variable. By default, this result is an object of the <code>HomogenNumericTable</code> class, but you can define the result as an object of any class derived from <code>NumericTable</code> , except for <code>PackedTriangularMatrix</code> , <code>PackedSymmetricMatrix</code> , and <code>CSRNumericTable</code> .
<i>determinationCoeff</i>	Pointer to the 1 x k numeric table that contains the determination coefficient computed for each dependent variable. By default, this result is an object of the <code>HomogenNumericTable</code> class, but you can

Result ID	Result
	define the result as an object of any class derived from <code>NumericTable</code> , except for <code>PackedTriangularMatrix</code> , <code>PackedSymmetricMatrix</code> , and <code>CSRNumericTable</code> .
<code>fStatistics</code>	Pointer to the $1 \times k$ numeric table that contains the F-statistics computed for each dependent variable. By default, this result is an object of the <code>HomogenNumericTable</code> class, but you can define the result as an object of any class derived from <code>NumericTable</code> , except for <code>PackedTriangularMatrix</code> , <code>PackedSymmetricMatrix</code> , and <code>CSRNumericTable</code> .

Examples

Refer to the following examples in your Intel DAAL directory:

C++: `./examples/cpp/source/quality_metrics/lin_reg_quality_metric_set_batch.cpp`

Java*: `./examples/java/source/com/intel/daal/examples/quality_metrics/LinRegQualityMetricSetBatchExample.java`

Python*: `./examples/python/source/quality_metrics/lin_reg_metrics_dense_batch.py`

Working with User-defined Quality Metrics

In addition to or instead of the metrics available in the library, you can use your own quality metrics. To do this:

1. Add your own implementation of the quality metrics algorithm and define `Input` and `Result` classes for that algorithm.
2. Register this new algorithm in the `inputAlgorithms` collection of the quality metric set. Also register the input objects for the new algorithm in the `inputData` collection of the quality metric set.

Use the unique key when registering the new algorithm and its input, and use the same key to obtain the computed results.

You can use the `useDefaultMetrics` flag to inform classification algorithms of a need to compute the default quality metrics.

Sorting

In Intel DAAL, sorting is an algorithm to sort the observations by each feature (column) in the ascending order.

Details

In Intel DAAL, the result of the sorting algorithm applied to the matrix $X = (x_{ij})_{n \times p}$ is the matrix $Y = (y_{ij})_{n \times p}$ where the j -th column $(Y)_j = (y_{ij})_{i=1, \dots, n}$ is the column $(X)_j = (x_{ij})_{i=1, \dots, n}$ sorted in the ascending order.

Batch Processing

Algorithm Input

The sorting algorithm accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
<code>data</code>	Pointer to the $n \times p$ numeric table that contains the input data set. This table can be an object of any class derived from <code>NumericTable</code> except <code>PackedSymmetricMatrix</code> , <code>PackedTriangularMatrix</code> , and <code>CSRNumericTable</code> .

Algorithm Parameters

The sorting algorithm has the following parameters:

Parameter	Default Value	Description
<code>algorithmFPType</code>	<code>double</code>	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<code>method</code>	<code>defaultDense</code>	The radix method for sorting a data set, the only method supported by the algorithm.

Algorithm Output

The sorting algorithm function calculates the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
<code>sortedData</code>	Pointer to the $n \times p$ numeric table that stores the results of sorting.

NOTE

If the number of feature vectors is greater than or equal to 2^{31} , the library uses the quick sort method instead of radix sort.

Examples

Refer to the following examples in your Intel DAAL directory:

C++: `./examples/cpp/source/sorting/sorting_batch.cpp`

Java*: `./examples/java/source/com/intel/daal/examples/sorting/SortingBatch.java`

Python*: `./examples/python/source/sorting/sorting_batch.py`

Normalization

Z-score

Z-score normalization is an algorithm to normalize the observations by each feature (column).

Problem Statement

Given a set X of n feature vectors $x_1 = (x_{11}, \dots, x_{1p})$, \dots , $x_n = (x_{n1}, \dots, x_{np})$ of dimension p , the problem is to compute the matrix $Y = (y_{ij})_{n \times p}$ where the j -th column $(Y)_j = (y_{ij})_{i=1, \dots, n}$ is obtained as a result of normalizing the column $(X)_j = (x_{ij})_{i=1, \dots, n}$ of the original matrix as:

$y_{ij} = \frac{x_{ij} - m_j}{\sigma_j}$, where m_j is the mean and σ_j is the standard deviation of column $(X)_j$.

Batch Processing

Algorithm Input

The Z-score normalization algorithm accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
data	Pointer to the numeric table of size $n \times p$. This table can be an object of any class derived from <code>NumericTable</code> .

Algorithm Parameters

The Z-score normalization algorithm has the following parameters. Some of them are required only for specific values of the computation method parameter *method*:

Parameter	<i>method</i>	Default Value	Description
<i>algorithm</i> <i>FPTYPE</i>	defaultDense or sumDense	double	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<i>method</i>	Not applicable	defaultDense	Available computation methods: <ul style="list-style-type: none"> <code>defaultDense</code> - performance-oriented method. <code>sumDense</code> - implementation of the algorithm in the cases where the basic statistics associated with the numeric table are pre-computed sums; returns an error if pre-computed sums are not defined.
<i>moments</i>	defaultDense	<code>SharedPtr<low_order_moments::Batch<algorithmFPTYPE, low_order_moments::defaultDense> ></code>	Pointer to the low order moments algorithm that computes means and standard deviations to be used for Z-score normalization with the <code>defaultDense</code> method. For details, see Moments of Low Order. Batch Processing .

Algorithm Output

The Z-score normalization algorithm calculates the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
normalizedData	Pointer to the $n \times p$ numeric table that stores the result of normalization. By default, the result is an object of the <code>HomogenNumericTable</code> class, but you can define the result as an object of any class derived from <code>NumericTable</code> except <code>PackedTriangularMatrix</code> , <code>PackedSymmetricMatrix</code> , and <code>CSRNumericTable</code> .

Examples

Refer to the following examples in your Intel DAAL directory:

C++: `./examples/cpp/source/normalization/zscore_batch.cpp`

Java*: `./examples/java/source/com/intel/daal/examples/normalization/ZScoreBatch.java`

Python*: `./examples/python/source/normalization/zscore_batch.py`

Min-max

Min-max normalization is an algorithm to linearly scale the observations by each feature (column) into the range $[a, b]$.

Problem Statement

Given a set X of n feature vectors $x_1 = (x_{11}, \dots, x_{1p}), \dots, x_n = (x_{n1}, \dots, x_{np})$ of dimension p , the problem is to compute the matrix $Y = (y_{ij})_{n \times p}$ where the j -th column $(Y)_j = (y_{ij})_{i=1, \dots, n}$ is obtained as a result of normalizing the column $(X)_j = (x_{ij})_{i=1, \dots, n}$ of the original matrix as:

$$y_{ij} = a + \frac{x_{ij} - \min(j)}{\max(j) - \min(j)}(b - a),$$

where

$$\min(j) = \min_{i=1..n} x_{ij},$$

$$\max(j) = \max_{i=1..n} x_{ij},$$

a and b are the parameters of the algorithm.

Batch Processing

Algorithm Input

The min-max normalization algorithm accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
data	Pointer to the numeric table of size $n \times p$. This table can be an object of any class derived from <code>NumericTable</code> .

Algorithm Parameters

The min-max normalization algorithm has the following parameters:

Parameter	Default Value	Description
<code>algorithmFPTYPE</code>	<code>double</code>	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<code>method</code>	<code>defaultDense</code>	Performance-oriented computation method, the only method supported by the algorithm.
<code>lowerBound</code>	<code>0.0</code>	The lower bound of the range to which the normalization scales values of the features.
<code>upperBound</code>	<code>1.0</code>	The upper bound of the range to which the normalization scales values of the features.
<code>moments</code>	<code>SharedPtr<low_order_moments::Batch<algorithmFPTYPE, low_order_moments::defaultDense> ></code>	Pointer to the low order moments algorithm that computes minimums and maximums to be used for min-max normalization with the <code>defaultDense</code> method. For more details, see Moments of Low Order > Batch Processing .

Algorithm Output

The min-max normalization algorithm calculates the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
<code>normalizedData</code>	Pointer to the $n \times p$ numeric table that stores the result of normalization. By default, the result is an object of the <code>HomogenNumericTable</code> class, but you can define the result as an object of any class derived from <code>NumericTable</code> except <code>PackedTriangularMatrix</code> , <code>PackedSymmetricMatrix</code> , and <code>CSRNumericTable</code> .

Examples

Refer to the following examples in your Intel DAAL directory:

C++: `./examples/cpp/source/normalization/minmax_dense_batch.cpp`

Performance Considerations

To get the best performance of normalization algorithms for homogeneous input data, provide the input data and store results in homogeneous numeric tables of the same type as specified in the `algorithmFPTYPE` class template parameter.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Math Functions

Logistic

The *logistic* function is defined as:

$$f(x) = \frac{1}{1 + e^{-x}}.$$

Problem Statement

Given n feature vectors $x_1 = (x_{11}, \dots, x_{1p})$, \dots , $x_n = (x_{n1}, \dots, x_{np})$ of dimension p , the problem is to compute the $(n \times p)$ -dimensional matrix $Y = (y_{ij})_{n \times p}$, where:

$$y_{ij} = \frac{1}{1 + e^{-x_{ij}}}.$$

Batch Processing

Algorithm Input

The logistic function accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
data	Pointer to the numeric table of size $n \times p$. The input can be an object of any class derived from <code>NumericTable</code> .

Algorithm Parameters

The logistic function has the following parameters:

Parameter	Default Value	Description
<code>algorithmFPType</code>	double	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<code>method</code>	defaultDense	Performance-oriented computation method, the only method supported by the algorithm.

Algorithm Output

The logistic function calculates the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
value	Pointer to the $n \times p$ numeric table with the matrix Y . By default, the result is an object of the <code>HomogenNumericTable</code> class, but you can define the result as an object of any class derived from <code>NumericTable</code> except <code>PackedTriangularMatrix</code> , <code>PackedSymmetricMatrix</code> , and <code>CSRNumericTable</code> .

Examples

Refer to the following examples in your Intel DAAL directory:

C++: `./examples/cpp/source/math/logistic_batch.cpp`

Java*: `./examples/java/source/com/intel/daal/examples/math/LogisticBatch.java`

Python*: `./examples/python/source/math/logistic_batch.py`

Softmax

Softmax function is defined as:

$$y_i(x) = \frac{e^{x_j}}{\sum_{k=1}^p e^{x_k}}$$

for $j = 1, \dots, p$. Softmax function is also known as the normalized exponential (see [Bishop2006] for exact definitions).

Problem Statement

Given n feature vectors $x_1 = (x_{11}, \dots, x_{1p})$, \dots , $x_n = (x_{n1}, \dots, x_{np})$ of dimension p , the problem is to compute the $(n \times p)$ -dimensional matrix $Y = (y_{ij})_{n \times p}$ such that:

$$y_{ij} = \frac{e^{x_{ij}}}{\sum_{k=1}^p e^{x_{ik}}}$$

The library supports the numerically stable version of the softmax function:

$$y_{ij} = \frac{e^{x_{ij} - \max_{j=1, \dots, p} x_{ij}}}{\sum_{k=1}^p e^{x_{ik} - \max_{k=1, \dots, p} x_{ik}}}$$

Batch Processing

Algorithm Input

The softmax function accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
data	Pointer to the numeric table of size $n \times p$. The input can be an object of any class derived from <code>NumericTable</code> .

Algorithm Parameters

The softmax function has the following parameters:

Parameter	Default Value	Description
<code>algorithmFPType</code>	<code>double</code>	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<code>method</code>	<code>defaultDense</code>	Performance-oriented computation method, the only method supported by the algorithm.

Algorithm Output

The softmax function calculates the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
<code>value</code>	Pointer to the $n \times p$ numeric table with the matrix Y . By default, the result is an object of the <code>HomogenNumericTable</code> class, but you can define the result as an object of any class derived from <code>NumericTable</code> except <code>PackedTriangularMatrix</code> , <code>PackedSymmetricMatrix</code> , and <code>CSRNumericTable</code> .

Examples

Refer to the following examples in your Intel DAAL directory:

C++: `./examples/cpp/source/math/softmax_batch.cpp`

Java*: `./examples/java/source/com/intel/daal/examples/math/SoftmaxBatch.java`

Python*: `./examples/python/source/math/softmax_batch.py`

Hyperbolic Tangent

The *hyperbolic tangent* is a trigonometric function defined as the ratio of hyperbolic sine to hyperbolic cosine:

$$p = \frac{\sinh x}{\cosh x}.$$

Problem Statement

Given n feature vectors $x_1 = (x_{11}, \dots, x_{1p})$, \dots , $x_n = (x_{n1}, \dots, x_{np})$ of dimension p , the problem is to compute the $(n \times p)$ -dimensional matrix $Y = (y_{ij})_{n \times p}$ such that:

$$y_{ij} = \tanh x_{ij} = \frac{\sinh x_{ij}}{\cosh x_{ij}}.$$

Batch Processing

Algorithm Parameters

The hyperbolic tangent function has the following parameters:

Parameter	Default Value	Description
<i>algorithmFPTType</i>	double	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<i>method</i>	defaultDense	Available computation methods: <ul style="list-style-type: none"> <code>defaultDense</code> - performance-oriented method <code>fastCSR</code> - performance-oriented method for CSR numeric tables

Algorithm Input

The hyperbolic tangent function accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
<i>data</i>	Pointer to the numeric table of size $n \times p$. The value of <i>method</i> determines the type of this numeric table: <ul style="list-style-type: none"> <i>method</i> = <code>defaultDense</code>: an object of any class derived from <code>NumericTable</code>. <i>method</i> = <code>fastCSR</code>: an object of the <code>CSRNumericTable</code> class.

Algorithm Output

The hyperbolic tangent function calculates the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
<i>value</i>	Pointer to the $n \times p$ numeric table with the matrix Y . The value of <i>method</i> determines the type of this numeric table: <ul style="list-style-type: none"> <i>method</i> = <code>defaultDense</code>: by default, an object of the <code>HomogenNumericTable</code> class, but can be an object of any class derived from <code>NumericTable</code> except <code>PackedTriangularMatrix</code>, <code>PackedSymmetricMatrix</code>, and <code>CSRNumericTable</code>. <i>method</i> = <code>fastCSR</code>: an object of the <code>CSRNumericTable</code> class.

Examples

Refer to the following examples in your Intel DAAL directory:

C++:

- `./examples/cpp/source/math/tanh_dense_batch.cpp`
- `./examples/cpp/source/math/tanh_csr_batch.cpp`

Java*:

- `./examples/java/source/com/intel/daal/examples/math/TanhDenseBatch.java`
- `./examples/java/source/com/intel/daal/examples/math/TanhCSRBatch.java`

Python*:

- `./examples/python/source/math/tanh_dense_batch.py`
- `./examples/python/source/math/tanh_csr_batch.py`

Absolute Value (abs)

The absolute value function *abs* is defined as $y = |x|$.

Problem Statement

Given n feature vectors $x_1 = (x_{11}, \dots, x_{1p}), \dots, x_n = (x_{n1}, \dots, x_{np})$ of dimension p , the problem is to compute the $(n \times p)$ -dimensional matrix $Y = (y_{ij})_{n \times p}$ such that:

$$y_{ij} = \text{abs}(x_{ij}).$$

Batch Processing

Algorithm Parameters

The *abs* function has the following parameters:

Parameter	Default Value	Description
<i>algorithmFPType</i>	double	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<i>method</i>	defaultDense	Available computation methods: <ul style="list-style-type: none"> • <code>defaultDense</code> - performance-oriented method • <code>fastCSR</code> - performance-oriented method for CSR numeric tables

Algorithm Input

The *abs* function accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
<i>data</i>	<p>Pointer to the numeric table of size $n \times p$. The value of <i>method</i> determines the type of this numeric table:</p> <ul style="list-style-type: none"> • <i>method</i> = <code>defaultDense</code>: an object of any class derived from <code>NumericTable</code>. • <i>method</i> = <code>fastCSR</code>: an object of the <code>CSRNumericTable</code> class.

Algorithm Output

The *abs* function calculates the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
<i>value</i>	<p>Pointer to the $n \times p$ numeric table with the matrix Y. The value of <i>method</i> determines the type of this numeric table:</p>

Result ID	Result
	<ul style="list-style-type: none"> <code>method = defaultDense</code>: by default, an object of the <code>HomogenNumericTable</code> class, but can be an object of any class derived from <code>NumericTable</code> except <code>PackedTriangularMatrix</code>, <code>PackedSymmetricMatrix</code>, and <code>CSRNumericTable</code>. <code>method = fastCSR</code>: an object of the <code>CSRNumericTable</code> class.

Examples

Refer to the following examples in your Intel DAAL directory:

C++:

- `./examples/cpp/source/math/abs_dense_batch.cpp`
- `./examples/cpp/source/math/abs_csr_batch.cpp`

Java*:

- `./examples/java/source/com/intel/daal/examples/math/AbsDenseBatch.java`
- `./examples/java/source/com/intel/daal/examples/math/AbsCSRBatch.java`

Python*:

- `./examples/python/source/math/abs_dense_batch.py`
- `./examples/python/source/math/abs_csr_batch.py`

Rectifier Linear Unit (ReLU)

The *ReLU* function is defined as $f(x) = \max(0, x)$.

Problem Statement

Given n feature vectors $x_1 = (x_{11}, \dots, x_{1p})$, \dots , $x_n = (x_{n1}, \dots, x_{np})$ of dimension p , the problem is to compute the $(n \times p)$ -dimensional matrix $Y = (y_{ij})_{n \times p}$ such that:

$$y_{ij} = \max(0, x_{ij}).$$

Batch Processing

Algorithm Parameters

The ReLU function has the following parameters:

Parameter	Default Value	Description
<code>algorithmFPType</code>	<code>double</code>	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<code>method</code>	<code>defaultDense</code>	Available computation methods: <ul style="list-style-type: none"> <code>defaultDense</code> - performance-oriented method <code>fastCSR</code> - performance-oriented method for CSR numeric tables

Algorithm Input

The ReLU function accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
data	<p>Pointer to the numeric table of size $n \times p$. The value of <i>method</i> determines the type of this numeric table:</p> <ul style="list-style-type: none"> <i>method</i> = defaultDense: an object of any class derived from NumericTable. <i>method</i> = fastCSR: an object of the CSRNumericTable class.

Algorithm Output

The ReLU function calculates the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
value	<p>Pointer to the $n \times p$ numeric table with the matrix Y. The value of <i>method</i> determines the type of this numeric table:</p> <ul style="list-style-type: none"> <i>method</i> = defaultDense: by default, an object of the HomogenNumericTable class, but can be an object of any class derived from NumericTable except PackedTriangularMatrix, PackedSymmetricMatrix, and CSRNumericTable. <i>method</i> = fastCSR: an object of the CSRNumericTable class.

Examples

Refer to the following examples in your Intel DAAL directory:

C++:

- `./examples/cpp/source/math/relu_dense_batch.cpp`
- `./examples/cpp/source/math/relu_csr_batch.cpp`

Java*:

- `./examples/java/source/com/intel/daal/examples/math/ReLUBatch.java`
- `./examples/java/source/com/intel/daal/examples/math/ReLUCSRBatch.java`

Python*:

- `./examples/python/source/math/relu_dense_batch.py`
- `./examples/python/source/math/relu_csr_batch.py`

Smooth Rectifier Linear Unit (SmoothReLU)

The *SmoothReLU* function is defined as $f(x) = \log(1 + \exp(x))$.

Problem Statement

Given n feature vectors $x_1 = (x_{11}, \dots, x_{1p})$, \dots , $x_n = (x_{n1}, \dots, x_{np})$ of dimension p , the problem is to compute the $(n \times p)$ -dimensional matrix $Y = (y_{ij})_{n \times p}$ such that:

$$y_{ij} = \log(1 + \exp(x_{ij})).$$

Batch Processing

Algorithm Input

The SmoothReLU function accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
data	Pointer to the numeric table of size $n \times p$. This table can be an object of any class derived from <code>NumericTable</code> .

Algorithm Parameters

The SmoothReLU function has the following parameters:

Parameter	Default Value	Description
<code>algorithmFPType</code>	double	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<code>method</code>	defaultDense	Performance-oriented computation method, the only method supported by the algorithm.

Algorithm Output

The SmoothReLU function calculates the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
output	Pointer to the numeric table of size $n \times p$ with the matrix Y . By default, the result is an object of the <code>HomogenNumericTable</code> class, but you can define the result as an object of any class derived from <code>NumericTable</code> except <code>PackedTriangularMatrix</code> , <code>PackedSymmetricMatrix</code> , and <code>CSRNumericTable</code> .

Examples

Refer to the following examples in your Intel DAAL directory:

C++: `./examples/cpp/source/math/smoothrelu_batch.cpp`

Java*: `./examples/java/source/com/intel/daal/examples/math/SmoothReLUBatch.java`

Python*: `./examples/python/source/math/smoothrelu_batch.py`

Performance Considerations

To get the best performance when computing math functions:

- If input data is homogeneous, provide the input data and store results in homogeneous numeric tables of the same type as specified in the `algorithmFPType` class template parameter.
- If input data is non-homogeneous, use AOS layout rather than SOA layout.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Optimization Solvers

An optimization solver is an algorithm to solve an optimization problem, that is, to find the maximum or minimum of an objective function in the presence of constraints on its variables. In Intel DAAL the optimization solver represents the interface of algorithms that search for the argument θ_* that minimizes the function $F(\theta)$:

$$\theta_* = \underset{\theta \in \Theta}{\operatorname{argmin}} F(\theta).$$

Objective Function

In Intel DAAL, the objective function represents an interface of objective functions that receive input arguments $\theta \in \mathbb{R}^p$ and return the gradient \vec{g} , value y , and Hessian matrix H :

$$\vec{g}(\theta) = \nabla F(\theta) = \left\{ \frac{\partial F}{\partial \theta_1}, \dots, \frac{\partial F}{\partial \theta_p} \right\},$$

$$y = F(\theta),$$

$$H = \nabla^2 F(\theta) = \begin{pmatrix} \frac{\partial^2 F}{\partial \theta_1 \partial \theta_1} & \frac{\partial^2 F}{\partial \theta_1 \partial \theta_2} & \dots & \frac{\partial^2 F}{\partial \theta_1 \partial \theta_p} \\ \frac{\partial^2 F}{\partial \theta_2 \partial \theta_1} & \frac{\partial^2 F}{\partial \theta_2 \partial \theta_2} & \dots & \frac{\partial^2 F}{\partial \theta_2 \partial \theta_p} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 F}{\partial \theta_p \partial \theta_1} & \frac{\partial^2 F}{\partial \theta_p \partial \theta_2} & \dots & \frac{\partial^2 F}{\partial \theta_p \partial \theta_p} \end{pmatrix}.$$

Computation

Input

The objective function accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
argument	Numeric table of size $p \times 1$ with the input argument of the objective function.

Parameters

The objective function has the following parameters:

Parameter	Default Value	Description	
<i>resultsToCompute</i>	gradient	The 64-bit integer flag that specifies which characteristics of the objective function to compute.	
		Provide one of the following values to request a single characteristic or use bitwise OR to request a combination of the characteristics:	
		gradient	gradient
		value	value
		hessian	Hessian

Output

The objective function calculates the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
gradientIdx	Numeric table of size $p \times 1$ with the gradient of the objective function in the given argument.
valueIdx	Numeric table of size 1×1 with the value of the objective function in the given argument.
hessianIdx	Numeric table of size $p \times p$ with the Hessian of the objective function in the given argument.

- If the function result is not requested through the *resultsToCompute* parameter, the respective element of the result contains a NULL pointer.
- By default, each numeric table specified by the collection elements is an object of the `HomogenNumericTable` class, but you can define the result as an object of any class derived from `NumericTable`, **except for** `PackedSymmetricMatrix`, `PackedTriangularMatrix`, and `CSRNumericTable`.

Sum of Functions

The sum of functions $F(\theta)$ is a function that has the form of a sum:

$$F(\theta) = \sum_{i=1}^n F_i(\theta), \quad \theta \in \mathbb{R}^p.$$

For given set of the indices $I = \{i_1, i_2, \dots, i_m\}$, $1 \leq i_k < n$, $k \in \{1, \dots, m\}$, the value and the gradient of the sum of functions in the argument θ has the format:

$$F_I(\theta) = \sum_{i \in I} F_i(\theta),$$

$$\nabla_I F(\theta) = \sum_{i \in I} \nabla F_i(\theta).$$

The set of the indices I is called a batch of indices.

Computation

Algorithm Input

The sum of functions algorithm accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
argument	Numeric table of size $p \times 1$ with the input argument of the objective function.

Algorithm Parameters

The sum of functions algorithm has the following parameters:

Parameter	Default Value	Description						
<i>numberOfTerms</i>	Not applicable	The number of terms in the sum.						
<i>batchIndices</i>	NULL	<p>The numeric table of size $1 \times m$, where m is the batch size, with a batch of indices to be used to compute the function results. If no indices are provided, the implementation uses all the terms in the sum.</p> <p>This parameter can be an object of any class derived from <code>NumericTable</code> except <code>PackedTriangularMatrix</code> and <code>PackedSymmetricMatrix</code>.</p>						
<i>resultsToCompute</i>	gradient	<p>The 64-bit integer flag that specifies which characteristics of the objective function to compute.</p> <p>Provide one of the following values to request a single characteristic or use bitwise OR to request a combination of the characteristics:</p> <table><tr><td>gradient</td><td>gradient</td></tr><tr><td>value</td><td>value</td></tr><tr><td>hessian</td><td>Hessian</td></tr></table>	gradient	gradient	value	value	hessian	Hessian
gradient	gradient							
value	value							
hessian	Hessian							

Algorithm Output

For the output of the sum of functions algorithm, see [Output](#) for objective functions.

Mean Squared Error Algorithm

Given $x = (x_{i1}, \dots, x_{ip}) \in R^p$, a set of feature vectors $i \in \{1, \dots, n\}$, and a set of respective responses y_i , the mean squared error (MSE) objective function $F(\theta; x, y)$ is a function that has the format:

$$F(\theta; x, y) = \sum_{i=1}^n F_i(\theta; x, y) = \frac{1}{2n} \sum_{i=1}^n (y_i - h(\theta, x_i))^2$$

In Intel DAAL implementation of the MSE, the $h(\theta, y_i)$ function is represented as:

$$h(\theta, x_i) = \theta_0 + \sum_{j=1}^p \theta_j x_{ij}$$

For a given set of the indices $I = \{i_1, i_2, \dots, i_m\}$, $1 \leq i_r < n$, $i \in \{1, \dots, m\}$, $|I| = m$, the value and the gradient of the sum of functions in the argument x respectively have the format:

$$F_I(\theta; x, y) = \frac{1}{2m} \sum_{i_k \in I} \left(y_{i_k} - h(\theta, x_{i_k}) \right)^2,$$

$$\nabla F_I(\theta; x, y) = \left\{ \frac{\partial F_I}{\partial \theta_0}, \dots, \frac{\partial F_I}{\partial \theta_p} \right\},$$

where

$$\frac{\partial F_I}{\partial \theta_0} = \frac{1}{m} \sum_{i_k \in I} \left(y_{i_k} - h(\theta, x_{i_k}) \right),$$

$$\frac{\partial F_I}{\partial \theta_j} = \frac{1}{m} \sum_{i_k \in I} \left(y_{i_k} - h(\theta, x_{i_k}) \right) x_{i_k j}, \quad j = 1, \dots, p.$$

Computation

Algorithm Input

The mean squared error algorithm accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
argument	Numeric table of size $(p + 1) \times 1$ with the input argument θ of the objective function.
data	Numeric table of size $n \times p$ with the data x_{ij} .
dependentVariables	Numeric table of size $n \times 1$ with dependent variables y_i .

Algorithm Parameters

The mean squared error algorithm has the following parameters. Some of them are required only for specific values of the computation method parameter *method*:

Parameter	Default Value	Description
<i>algorithmFPTType</i>	double	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<i>method</i>	defaultDense	Performance-oriented computation method.
<i>numberOfTerms</i>	Not applicable	The number of terms in the objective function.
<i>batchIndices</i>	NULL	The numeric table of size $1 \times m$, where m is the batch size, with a batch of indices to be used to compute the function results. If no indices are provided, the implementation uses all the terms in the computation.

Parameter	Default Value	Description						
		This parameter can be an object of any class derived from NumericTable except PackedTriangularMatrix and PackedSymmetricMatrix.						
resultsToCompute	gradient	<p>The 64-bit integer flag that specifies which characteristics of the objective function to compute.</p> <p>Provide one of the following values to request a single characteristic or use bitwise OR to request a combination of the characteristics:</p> <table><tr><td>gradient</td><td>gradient</td></tr><tr><td>value</td><td>value</td></tr><tr><td>hessian</td><td>Hessian</td></tr></table>	gradient	gradient	value	value	hessian	Hessian
gradient	gradient							
value	value							
hessian	Hessian							

Algorithm Output

For the output of the mean squared error algorithm, see [Output](#) for objective functions.

Examples

Refer to the following examples in your Intel DAAL directory:

C++: `./examples/cpp/source/optimization_solvers/mse_dense_batch.cpp`

Java*: `./examples/java/source/com/intel/daal/examples/optimization_solvers/MSEDenseBatch.java`

Python*: `./examples/python/source/optimization_solvers/mse_dense_batch.py`

See Also

[Performance Considerations](#)

Objective Function with Precomputed Characteristics Algorithm

Objective function with precomputed characteristics gives an ability to provide the results of the objective function precomputed with the user-defined algorithm.

Set an earlier computed value and/or gradient and/or Hessian by allocating the result object and setting the characteristics of this result object. After that provide the modified result object to the algorithm for its further use with the iterative solver.

For more details on iterative solvers, refer to [Iterative Solver](#).

Computation

Algorithm Input

For the input of the objective function with precomputed characteristics algorithm, see [Input](#) for objective functions.

NOTE

The algorithm does not use the provided input objects.

Algorithm Output

For the output of the objective function with precomputed characteristics algorithm, see [Output](#) for objective functions.

Iterative Solver

The iterative solver provides an iterative method to minimize an objective function that can be represented as a sum of functions:

$$\theta_* = \underset{\theta \in \mathbb{R}^p}{\operatorname{argmin}} F(\theta),$$

$$F(\theta) = \sum_{i=1}^n F_i(\theta), \quad \theta \in \mathbb{R}^p.$$

The Algorithmic Framework of an Iterative Solver

Let S_t be a set of intrinsic parameters of the iterative solver for updating the argument of the objective function. This set is the algorithm-specific and can be empty. The solver determines the choice of S_0 .

To do the computations, iterate t from 1 until $nIterations$:

1. Choose a set of indices without replacement

$$I = \{i_1, \dots, i_b\}, 1 \leq i_j \leq n, j = 1, \dots, b,$$

where b is the batch size.

2. Compute the gradient

$$g(\theta_{t-1}) = \nabla F_I(\theta_{t-1}),$$

where

$$F_I(\theta_{t-1}) = \sum_{i \in I} F_i(\theta_{t-1}).$$

3. Stop if

$$\|g(\theta_{t-1})\| < \varepsilon \max(1, \|\theta_{t-1}\|).$$

4. Compute

$$\theta_t = T(\theta_{t-1}, g(\theta_{t-1}), S_{t-1}),$$

where T is an algorithm-specific transformation that updates the function argument.

5. Update S_t : $S_t = U(S_{t-1})$, where U is an algorithm-specific update of the set of intrinsic parameters.

The result of the solver is the argument θ_* and a set of parameters S_* after the exit from the loop.

NOTE

You can resume the computations to get a more precise estimate of the objective function minimum. To do this, pass to the algorithm the results θ_0 and S_0 of the previous run of the optimization solver. By default, the solver does not return the set of intrinsic parameters. If you need it, set the `optionalResultRequired` flag for the algorithm.

Computation**Algorithm Input**

The iterative solver algorithm accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
<code>inputArgument</code>	Numeric table of size $p \times 1$ with the value of start argument θ_0 .
<code>optionalArgument</code>	Object of the <code>OptionalArgument</code> class that contains a set of algorithm-specific intrinsic parameters. For a detailed definition of the set, see the problem statement above and the description of a specific algorithm.

Algorithm Parameters

The iterative solver algorithm has the following parameters:

Parameter	Default Value	Description
<code>function</code>	Not applicable	Objective function represented as a sum of functions.
<code>nIterations</code>	100	Maximum number of iterations of the algorithm.
<code>accuracyThreshold</code>	1.0-e5	Accuracy of the algorithm. The algorithm terminates when this accuracy is achieved.
<code>optionalResultRequired</code>	false	Indicates whether the set of the intrinsic parameters should be returned by the solver.

Algorithm Output

The iterative solver algorithm calculates the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
<code>minimum</code>	Numeric table of size $p \times 1$ with argument θ_* . By default, the result is an object of the <code>HomogenNumericTable</code> class, but you can define the result as an object of any class derived from <code>NumericTable</code> , except for <code>PackedTriangularMatrix</code> and <code>PackedSymmetricMatrix</code> .

Result ID	Result
nIterations	Numeric table of size 1 x 1 with a 32-bit integer number of iterations done by the algorithm. By default, the result is an object of the <code>HomogenNumericTable</code> class, but you can define the result as an object of any class derived from <code>NumericTable</code> , except for <code>PackedTriangularMatrix</code> , <code>PackedSymmetricMatrix</code> , and <code>CSRNumericTable</code> .
optionalResult	Object of the <code>OptionalArgument</code> class that contains a set of algorithm-specific intrinsic parameters. For a detailed definition of the set, see the problem statement above and the description of a specific algorithm.

See Also

Performance Considerations

Stochastic Gradient Descent Algorithm

The stochastic gradient descent (SGD) algorithm is a special case of an iterative solver. For more details, see [Iterative Solver](#).

The following computation methods are available in Intel DAAL for the stochastic gradient descent algorithm:

- Mini-batch.
- A special case of mini-batch used by default.
- Momentum

Mini-batch method

The mini-batch method (`miniBatch`) of the stochastic gradient descent algorithm [Mu2014] follows the [algorithmic framework of an iterative solver](#) with an empty set of intrinsic parameters of the algorithm S_t and the algorithm-specific transformation T defined for the learning rate sequence $\{\eta_t\}_{t=1,\dots,nIterations}$, conservative sequence $\{\gamma_t\}_{t=1,\dots,nIterations}$, and the number of iterations in the internal loop L as follows:

$$T(\theta_{t-1}, g(\theta_{t-1}), S_{t-1}):$$

For l from 1 until L :

1. Update the function argument

$$\theta_t := \theta_{t-1} - \eta_t(g(\theta_{t-1}) + \gamma_t(\theta_{t-1} - \theta_{t-2}))$$

2. Compute the gradient

$$g(\theta_t) = \nabla F_I(\theta_t)$$

Default method

The default method (`defaultDense`) is a particular case of the mini-batch method with the batch size $b=1$, $L=1$, and conservative sequence $\gamma_t \equiv 0$.

Momentum method

The momentum method (`momentum`) of the stochastic gradient descent algorithm [Rumelhart86] follows the algorithmic framework of an iterative solver with the set of intrinsic parameters S_t and algorithm-specific transformation T defined for the learning rate sequence $\{\eta_t\}_{t=1,\dots,nIterations}$ and momentum parameter $\mu \in [0,1]$ as follows:

$$T(\theta_{t-1}, g(\theta_{t-1}), S_{t-1}):$$

1. $v_t = \mu \cdot v_{t-1} + \eta_t \cdot g(\theta_{t-1})$
2. $\theta_t = \theta_{t-1} - v_t$

For the momentum method of the SGD algorithm, the set of intrinsic parameters S_t only contains the last update vector v_t .

Computation

The stochastic gradient descent algorithm is a special case of an iterative solver. For parameters, input, and output of iterative solvers, see [Iterative Solver > Computation](#).

Algorithm Parameters

In addition to parameters of the iterative solver, the stochastic gradient descent algorithm has the following parameters. Some of them are required only for specific values of the computation method parameter `method`:

Parameter	method	Default Value	Description
<code>algorithmFPType</code>	<code>defaultDense</code> , <code>miniBatch</code> , or <code>momentum</code>	<code>double</code>	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<code>method</code>	Not applicable	<code>defaultDense</code>	Available computation methods: <ul style="list-style-type: none"> <code>defaultDense</code> <code>miniBatch</code> <code>momentum</code>
<code>batchIndices</code>	<code>defaultDense</code> , <code>miniBatch</code> , or <code>momentum</code>	<code>NULL</code>	The numeric table with 32-bit integer indices of terms in the objective function. The <code>method</code> parameter determines the size of the numeric table: <ul style="list-style-type: none"> <code>defaultDense</code>: <code>nIterations</code> x 1 <code>miniBatch</code> or <code>momentum</code>: <code>nIterations</code> x <code>batchSize</code>

Parameter	method	Default Value	Description
			<p>If no indices are provided, the implementation generates random indices.</p> <p>This parameter can be an object of any class derived from <code>NumericTable</code> except <code>PackedTriangularMatrix</code>, <code>PackedSymmetricMatrix</code>, and <code>CSRNumericTable</code>.</p>
<i>batchSize</i>	miniBatch or momentum	128	<p>Number of batch indices to compute the stochastic gradient. If <i>batchSize</i> equals the number of terms in the objective function, no random sampling is performed, and all terms are used to calculate the gradient.</p> <p>The algorithm ignores this parameter if the <i>batchIndices</i> parameter is provided.</p> <p>For the <code>defaultDense</code> value of <i>method</i>, one term is used to compute the gradient on each iteration.</p>
<i>conservativeSequence</i>	miniBatch	Numeric table of size 1 x 1 that contains the default conservative coefficient equal to 1.	<p>The numeric table of size 1 x <i>nIterations</i> or 1 x 1. The contents of the table depend on its size:</p> <ul style="list-style-type: none"> size = 1 x <i>nIterations</i>: values of the conservative coefficient sequence γ^k for $k = 1, \dots, nIterations$. size = 1 x 1: the value of conservative coefficient at each iteration $\gamma^1 = \dots = \gamma^{nIterations}$. <p>This parameter can be an object of any class derived from <code>NumericTable</code> except <code>PackedTriangularMatrix</code>, <code>PackedSymmetricMatrix</code>, and <code>CSRNumericTable</code>.</p>
<i>innerNIterations</i>	miniBatch	5	<p>The number of inner iterations for the <code>miniBatch</code> method.</p>

Parameter	method	Default Value	Description
<i>learningRateSequence</i>	defaultDense, miniBatch, or momentum	Numeric table of size 1 x 1 that contains the default step length equal to 1.	<p>The numeric table of size 1 x <i>nIterations</i> or 1 x 1. The contents of the table depend on its size:</p> <ul style="list-style-type: none"> size = 1 x <i>nIterations</i>: values of the learning rate sequence η^k for $k = 1, \dots, nIterations$. size = 1 x 1: the value of learning rate at each iteration $\eta^1 = \dots = \eta^{nIterations}$. <p>This parameter can be an object of any class derived from <code>NumericTable</code> except <code>PackedTriangularMatrix</code>, <code>PackedSymmetricMatrix</code>, and <code>CSRNumericTable</code>.</p>
<i>momentum</i>	momentum	0.9	The momentum value.
<i>seed</i>	defaultDense, miniBatch, or momentum	777	The seed for random generation of 32-bit integer indices of terms in the objective function.

Examples

Refer to the following examples in your Intel DAAL directory:

C++:

- ./examples/cpp/source/optimization_solvers/sgd_dense_batch.cpp
- ./examples/cpp/source/optimization_solvers/sgd_mini_dense_batch.cpp
- ./examples/cpp/source/optimization_solvers/sgd_moment_dense_batch.cpp
- ./examples/cpp/source/optimization_solvers/sgd_moment_opt_res_dense_batch.cpp

Java*:

- ./examples/java/source/com/intel/daal/examples/optimization_solvers/SGDDenseBatch.java
- ./examples/java/source/com/intel/daal/examples/optimization_solvers/SGDMiniDenseBatch.java
- ./examples/java/source/com/intel/daal/examples/optimization_solvers/SGDMomentDenseBatch.java
- ./examples/java/source/com/intel/daal/examples/optimization_solvers/SGDMomentOptResDenseBatch.java

Python*:

- ./examples/python/source/optimization_solvers/sgd_batch.py
- ./examples/python/source/optimization_solvers/sgd_mini_batch.py

- `./examples/python/source/optimization_solvers/sgd_moment_dense_batch.py`
- `./examples/python/source/optimization_solvers/sgd_moment_opt_res_dense_batch.py`

See Also

Performance Considerations

Limited-Memory Broyden-Fletcher-Goldfarb-Shanno Algorithm

The limited-memory Broyden-Fletcher-Goldfarb-Shanno (LBFGS) algorithm [Byrd2015] follows the [algorithmic framework of an iterative solver](#) with the algorithm-specific transformation T and set of intrinsic parameters S_t defined for the memory parameter m , frequency of curvature estimates calculation L , and step-length sequence $\alpha^t > 0$ as follows:

Transformation

$$T(\theta_{t-1}, g(\theta_{t-1}), S_{t-1}) :$$

$$\theta_t = \begin{cases} \theta_{t-1} - \alpha^t g(\theta_{t-1}), & \text{if } t \leq 2L \\ \theta_{t-1} - \alpha^t H g(\theta_{t-1}), & \text{otherwise} \end{cases},$$

where H is an approximation of the inverse Hessian matrix computed from m correction pairs by the [Hessian Update Algorithm](#).

Intrinsic Parameters

For the LBFGS algorithm, the set of intrinsic parameters S_t includes the following:

- Correction pairs (s_j, y_j)
- Correction index k in the buffer that stores correction pairs
- Index of last iteration t of the main loop from the previous run
- Average value of arguments for the previous L iterations $\overline{\theta_{k-1}}$
- Average value of arguments for the last L iterations $\overline{\theta_k}$

Below is the definition and update flow of the intrinsic parameters (s_i, y_i) . The index is set and remains zero for the first $2L-1$ iterations of the main loop. Starting with iteration $2L$, the algorithm executes the following steps for each of L iterations of the main loop:

1. $k := k+1$
2. Choose a set of indices without replacement: $I_H = \{i_1, i_2, \dots, i_{b_H}\}, 1 \leq i_l < n, l \in \{1, \dots, b_H\}, |I_H| = b_H = \text{correctionPairBatchSize}$.
3. Compute the sub-sampled Hessian

$$\nabla^2 F(\overline{\theta_k}) = \frac{1}{b_H} \sum_{i \in I_H} \nabla^2 F_i(\overline{\theta_k})$$

at the point $\overline{\theta_k} = \frac{1}{L} \sum_{i=Lk}^{L(k+1)} \theta_i$ for the objective function using Hessians of its terms

$$\nabla^2 F_i = \begin{bmatrix} \frac{\partial F_i}{\partial \theta_0 \partial \theta_0} & \cdots & \frac{\partial F_i}{\partial \theta_0 \partial \theta_p} \\ \vdots & \ddots & \vdots \\ \frac{\partial F_i}{\partial \theta_p \partial \theta_0} & \cdots & \frac{\partial F_i}{\partial \theta_p \partial \theta_p} \end{bmatrix}$$

4. Compute the correction pairs (s_k, y_k):

$$s_k = \overline{\theta}_k - \overline{\theta}_{k-1},$$

$$y_k = \nabla^2 F(\overline{\theta}_k) s_k$$

- The set S_k of intrinsic parameters is updated once per L iterations of the major loop and remains unchanged between iterations with the numbers that are multiples of L
- A cyclic buffer stores correction pairs. The algorithm fills the buffer with pairs one-by-one. Once the buffer is full, it returns to the beginning and overwrites the previous correction pairs.

Hessian Update Algorithm

This algorithm computes the approximation of the inverse Hessian matrix from the set of correction pairs [Byrd2015]).

For a given set of correction pairs (s_j, y_j), $j = k - \min(k, m) + 1, \dots, k$:

1. Set $H = s_k^T y_k / y_k^T y_k$
2. Iterate j from $k - \min(k, m) + 1$ until k :
 - a. $\rho_j = 1 / y_j^T y_j$.
 - b. $H := (I - \rho_j s_j y_j^T) H (I - \rho_j y_j s_j^T) + \rho_j s_j s_j^T$.
3. Return H

Computation

The limited-memory BFGS algorithm is a special case of an iterative solver. For parameters, input, and output of iterative solvers, see [Iterative Solver > Computation](#).

Algorithm Input

In addition to the input of the iterative solver, the limited-memory BFGS algorithm accepts the following optional input:

OptionalDataID	Input
correctionPairs	Numeric table of size $2m \times p$ where the rows represent correction pairs s and y . The row <code>correctionPairs[j]</code> , $0 \leq j < m$, is a correction vector s_j , and the row <code>correctionPairs[j]</code> , $m \leq j < 2m$, is a correction vector y_j .
correctionIndices	Numeric table of size 1×2 with 32-bit integer indexes. The first value is the index of correction pair t , the second value is the index of last iteration k from the previous run.
averageArgumentLIterations	Numeric table of size $2 \times p$, where row 0 represents average arguments for previous L iterations, and row 1 represents average arguments for last L iterations. These values are required to compute s correction vectors in the next step. See step 6.b.iii of the limited-memory BFGS algorithm.

Algorithm Parameters

In addition to parameters of the iterative solver, the limited-memory BFGS algorithm has the following parameters:

Parameter	Default Value	Description
<i>algorithmFPTType</i>	double	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<i>method</i>	defaultDense	Performance-oriented computation method.
<i>batchIndices</i>	NULL	<p>The numeric table of size <i>nIterations</i> x <i>batchSize</i> with 32-bit integer indices of terms in the objective function to be used in step 2 of the limited-memory BFGS algorithm. If no indices are provided, the implementation generates random indices.</p> <p>This parameter can be an object of any class derived from <code>NumericTable</code>, except for <code>PackedTriangularMatrix</code>, <code>PackedSymmetricMatrix</code>, and <code>CSRNumericTable</code>.</p>
<i>batchSize</i>	10	<p>The number of observations to compute the stochastic gradient. The implementation of the algorithm ignores this parameter if the <i>batchIndices</i> numeric table is provided.</p> <p>If <i>BatchSize</i> equals the number of terms in the objective function, no random sampling is performed and all terms are used to calculate the gradient.</p>
<i>correctionPairBatchSize</i>	100	<p>The number of observations to compute the sub-sampled Hessian for correction pairs computation in step 6.b.ii of the limited-memory BFGS algorithm. The implementation of the algorithm ignores this parameter if the <i>correctionPairIndices</i> numeric table is provided.</p> <p>If <i>correctionPairBatchSize</i> equals the number of terms in the objective function, no random sampling is performed and all terms are used to calculate the Hessian matrix.</p>
<i>correctionPairIndices</i>	NULL	<p>The numeric table of size $(nIterations/L) \times correctionPairBatchSize$ with 32-bit integer indices to be used instead of random values in step 6.b.i of the limited-memory BFGS algorithm. If no indices are provided, the implementation generates random indices.</p>

Parameter	Default Value	Description
		<p>This parameter can be an object of any class derived from <code>NumericTable</code>, except for <code>PackedTriangularMatrix</code>, <code>PackedSymmetricMatrix</code>, and <code>CSRNumericTable</code>.</p> <hr/> <p>NOTE If the algorithm runs with no optional input data, $(nIterations / L - 1)$ rows of the table are used. Otherwise, it can use one more row, $(nIterations / L)$ in total.</p> <hr/>
<i>m</i>	10	The memory parameter. The maximum number of correction pairs that define the approximation of the Hessian matrix.
<i>L</i>	10	The number of iterations between calculations of the curvature estimates.
<i>stepLengthSequence</i>	Numeric table of size 1 x 1 that contains the default step length equal to 1.	<p>The numeric table of size 1 x <i>nIterations</i> or 1 x 1. The contents of the table depend on its size:</p> <ul style="list-style-type: none"> size = 1 x <i>nIterations</i>: values of the step-length sequence α^k for $k = 1, \dots, nIterations$. size = 1 x 1: the value of step length at each iteration $\alpha^1 = \dots = \alpha^{nIterations}$ <p>This parameter can be an object of any class derived from <code>NumericTable</code>, except for <code>PackedTriangularMatrix</code>, <code>PackedSymmetricMatrix</code>, and <code>CSRNumericTable</code>.</p> <p>The recommended data type for storing the step-length sequence is the floating-point type, either <code>float</code> or <code>double</code>, that the algorithm uses in intermediate computations.</p>
<i>seed</i>	777	The seed for randomly choosing terms from the objective function.

Algorithm Output

In addition to the output of the iterative solver, the limited-memory BFGS algorithm calculates the following optional results:

OptionalDataID	Output
correctionPairs	Numeric table of size $2m \times p$ where the rows represent correction pairs s and y . The row <code>correctionPairs[j]</code> , $0 \leq j < m$, is a correction vector s_j , and the row <code>correctionPairs[j]</code> , $m \leq j < 2m$, is a correction vector y_j .
correctionIndices	Numeric table of size 1×2 with 32-bit integer indexes. The first value is the index of correction pair t , the second value is the index of last iteration k from the previous run.
averageArgumentLIterations	Numeric table of size $2 \times p$, where row 0 represents average arguments for previous L iterations, and row 1 represents average arguments for last L iterations. These values are required to compute s correction vectors in the next step. See step 6.b.iii of the limited-memory BFGS algorithm.

Examples

Refer to the following examples in your Intel DAAL directory:

C++:

- `./examples/cpp/source/optimization_solvers/lbfgs_batch.cpp`
- `./examples/cpp/source/optimization_solvers/lbfgs_opt_res_dense_batch.cpp`

Java*:

- `./examples/java/source/com/intel/daal/examples/optimization_solvers/LBFGSBatch.java`
- `./examples/java/source/com/intel/daal/examples/optimization_solvers/LBFGSOptResDenseBatch.java`

Python*:

- `./examples/python/source/optimization_solvers/lbfgs_batch.py`
- `./examples/python/source/optimization_solvers/lbfgs_opt_res_dense_batch.py`

See Also

Performance Considerations

Adaptive Subgradient Method

The adaptive subgradient method (AdaGrad) [Duchi2011] follows the [algorithmic framework of an iterative solver](#) with the algorithm-specific transformation T and set of intrinsic parameters S_t defined for the learning rate η as follows:

- $S_t = \{G_t\}$, $G_t = (G_{t,i})_{i=1,\dots,p}$, $G_0 \equiv 0$
- $T(\theta_{t-1}, g(\theta_{t-1}), S_{t-1})$:

1. $G_{t,i} = G_{t-1,i} + g_i^2(\theta_{t-1})$,

where $g_i(\theta_{t-1})$ is the i -th coordinate of the gradient $g(\theta_{t-1})$

2. $\theta_t = \theta_{t-1} - \frac{\eta}{\sqrt{G_t + \varepsilon}} g(\theta_{t-1})$,

where $\frac{\eta}{\sqrt{G_t + \varepsilon}} g(\theta_{t-1}) = \left\{ \frac{\eta}{\sqrt{G_{t,1} + \varepsilon}} g_1(\theta_{t-1}), \dots, \frac{\eta}{\sqrt{G_{t,p} + \varepsilon}} g_p(\theta_{t-1}) \right\}$

Computation

The adaptive subgradient (AdaGrad) method is a special case of an iterative solver. For parameters, input, and output of iterative solvers, see [Iterative Solver > Computation](#).

Algorithm Input

In addition to the input of the iterative solver, the AdaGrad method accepts the following optional input:

OptionalDataID	Input
<code>gradientSquareSum</code>	Numeric table of size $p \times 1$ with the values of G_t . Each value is an accumulated sum of squares of coordinate values of a corresponding gradient.

Algorithm Parameters

In addition to parameters of the iterative solver, the AdaGrad method has the following parameters:

Parameter	Default Value	Description
<code>algorithmFPType</code>	<code>double</code>	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<code>method</code>	<code>defaultDense</code>	Default performance-oriented computation method.
<code>batchIndices</code>	<code>NULL</code>	Numeric table of size <code>nIterations</code> x <code>batchSize</code> for the <code>defaultDense</code> method that represents 32-bit integer indices of terms in the objective function. If no indices are provided, the algorithm generates random indices.
<code>batchSize</code>	<code>128</code>	Number of batch indices to compute the stochastic gradient. If <code>batchSize</code> equals the number of terms in the objective function, no random sampling is performed, and all terms are used to calculate the gradient. The algorithm ignores this parameter if the <code>batchIndices</code> parameter is provided.
<code>learningRate</code>	Numeric table of size 1×1 that contains the default step length equal to 0.01.	Numeric table of size 1×1 that contains the value of learning rate η . This parameter can be an object of any class derived from <code>NumericTable</code> , except for <code>PackedTriangularMatrix</code> , <code>PackedSymmetricMatrix</code> , and <code>CSRNumericTable</code> .

Parameter	Default Value	Description
<i>degenerateCasesThreshold</i>	1e-08	Value ε needed to avoid degenerate cases when computing square roots.
<i>seed</i>	777	The seed for random generation of 32-bit integer indices of terms in the objective function.

Algorithm Output

In addition to the output of the iterative solver, the AdaGrad method calculates the following optional result:

OptionalDataID	Output
<i>gradientSquareSum</i>	Numeric table of size $p \times 1$ with the values of G_t . Each value is an accumulated sum of squares of coordinate values of a corresponding gradient.

Examples

Refer to the following examples in your Intel DAAL directory:

C++:

- `./examples/cpp/source/optimization_solvers/adagrad_batch.cpp`
- `./examples/cpp/source/optimization_solvers/adagrad_opt_res_dense_batch.cpp`

Java*:

- `./examples/java/source/com/intel/daal/examples/optimization_solvers/AdagradBatch.java`
- `./examples/java/source/com/intel/daal/examples/optimization_solvers/AdagradOptResDenseBatch.java`

Python*:

- `./examples/python/source/optimization_solvers/adagrad_batch.py`
- `./examples/python/source/optimization_solvers/adagrad_opt_res_dense_batch.py`

See Also

Performance Considerations

Performance Considerations

To get the best performance of iterative solvers:

- If input data is homogeneous, provide the input data and store results in homogeneous numeric tables of the same type as specified in the *algorithmFPType* class template parameter.
- If input data is non-homogeneous, use AOS layout rather than SOA layout.
- For the numeric tables specified by the following parameters of specific iterative solvers, use the floating-point data type specified in the *algorithmFPType* class template parameter:
 - *learningRateSequence*
 - *stepLengthSequence*
 - *learningRate*

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Training and Prediction

Training and prediction algorithms in Intel® Data Analytics Acceleration Library (Intel® DAAL) include a range of popular machine learning algorithms. Unlike analysis algorithms, which are intended to characterize the structure of data sets, machine learning algorithms model the data. Modeling operates in two major stages:

1. *Training.*

At this stage, the algorithm estimates model parameters based on a training data set.

2. *Prediction or decision making.*

At this stage, the algorithm uses the trained model to predict the outcome based on new data.

The following major categories of training and prediction methods are available in Intel DAAL:

- Regression methods.
These methods predict the values of dependent variables (responses) by observing independent variables.
- Classification methods.
These methods identify to which sub-population (class) a given observation belongs.
- Recommendation methods.
These methods predict the preference that a user would give to a certain item.
- Neural networks.
These information processing systems provide methods to approximate functions of large numbers of arguments and to solve different kinds of machine learning tasks in particular.

Training is typically a lot more computationally complex problem than prediction. Therefore, certain end-to-end analytics usage scenarios require that training and prediction phases are done on distinct devices, the training is done on more powerful devices, while prediction is done on smaller devices. Because smaller devices may have stricter memory footprint requirements, Intel DAAL separates Training, Prediction, and respective Model in three different class hierarchies to minimize the footprint.

Regression

Regression analysis is widely used to predict and forecast, as well as to understand which of the independent variables are related to dependent variables, along with the form of the relationship. The regression function, which is estimated based on the training data, defines the form of the relationship.

Regression methods are divided into the following groups:

- Parametric methods,
where the regression function is defined in terms of a finite number of unknown parameters. For example: Linear Regression or Linear Least Squares.
- Non-parametric methods,
where the regression function is constructed from a set of kernel functions that may be infinite-dimensional. For example: Non-Parametric Multiplicative Regression or Regression Trees.

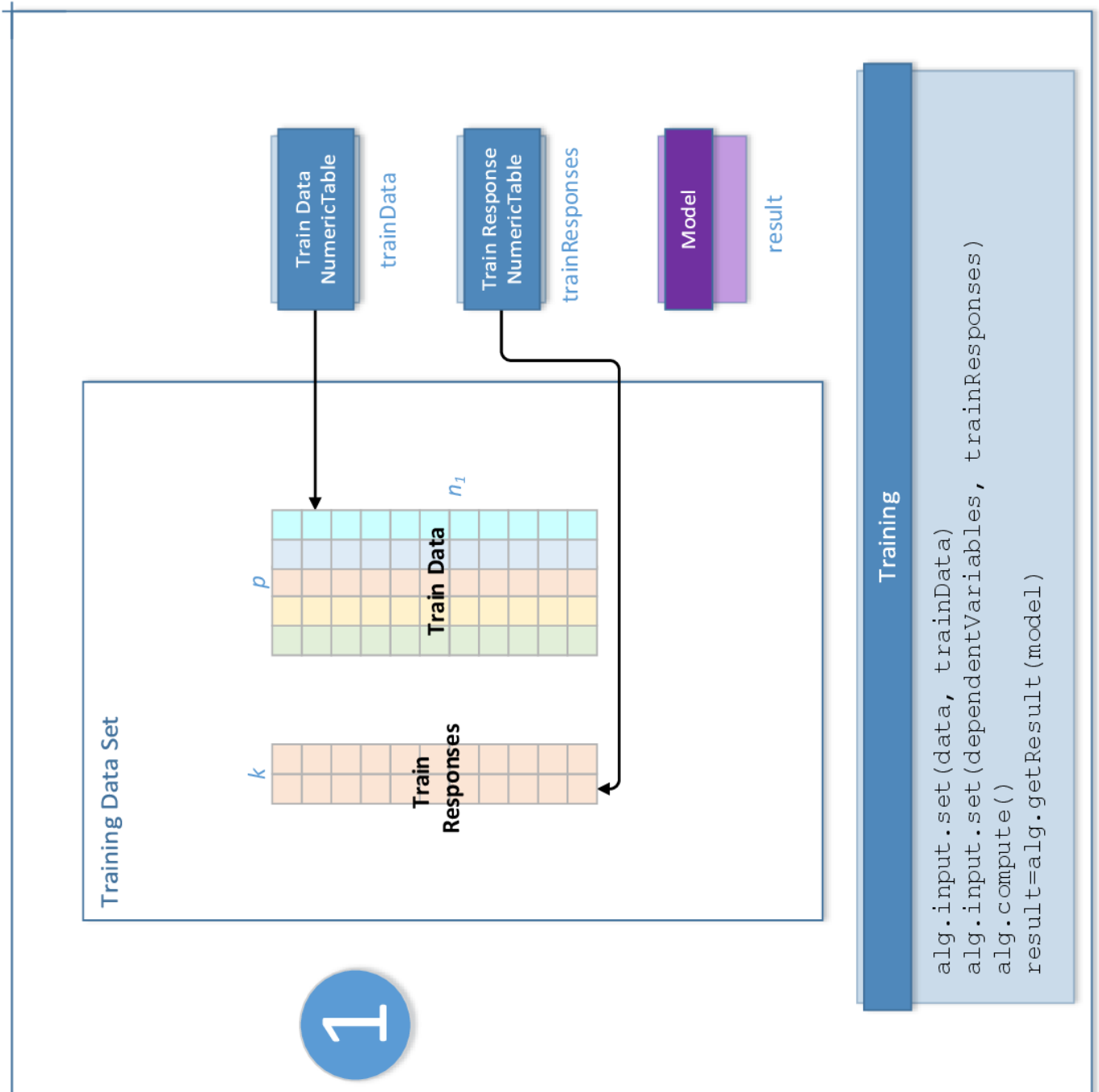
Usage Model: Training and Prediction

A typical workflow for regression methods includes training and prediction, as explained below.

Algorithm-Specific Parameters

The parameters used by regression algorithms at each stage depend on a specific algorithm. For a list of these parameters, refer to the description of an appropriate regression algorithm.

Training Stage



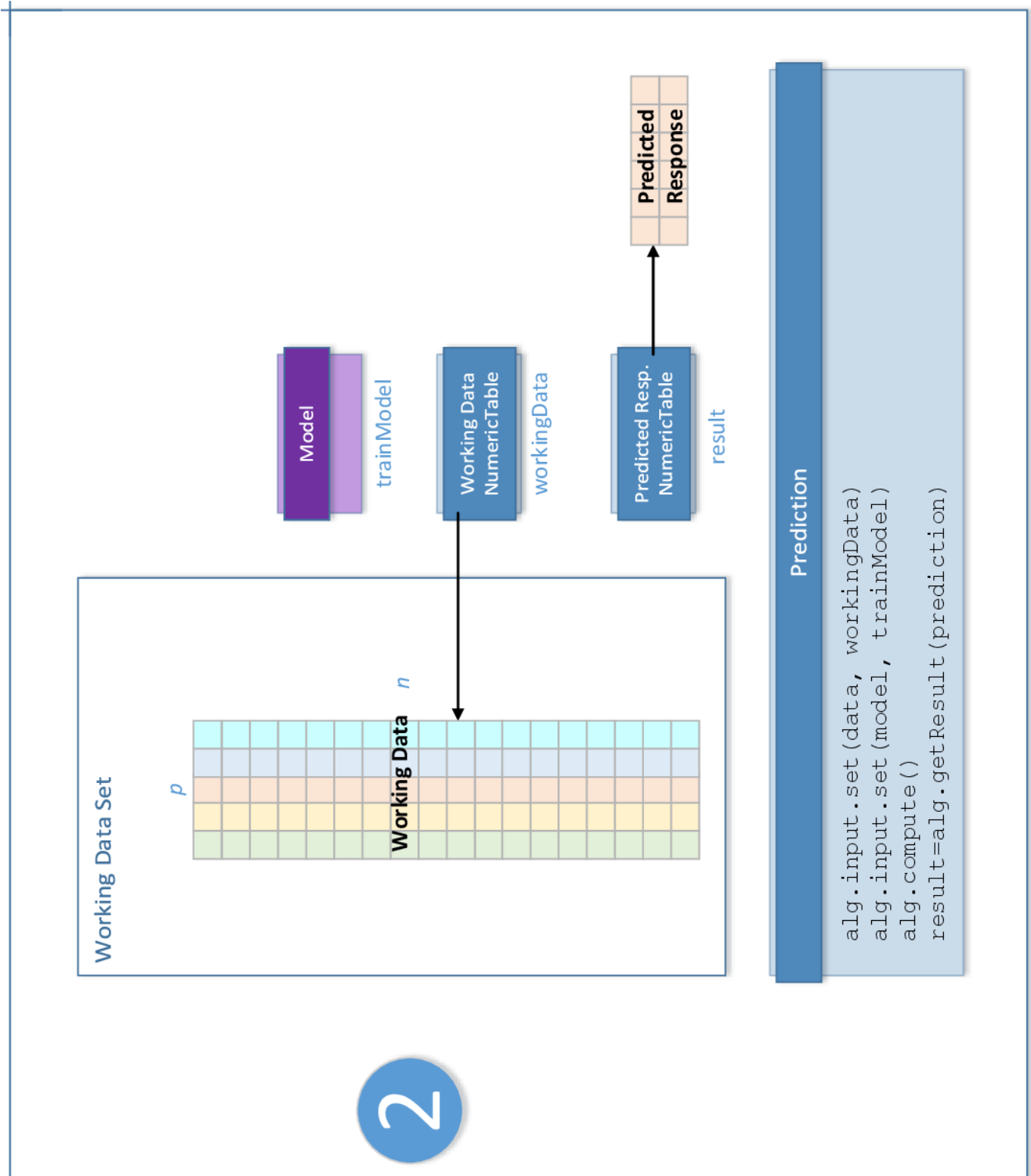
At the training stage, regression algorithms accept the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
<code>data</code>	Pointer to the $n \times p$ numeric table with the training data set. This table can be an object of any class derived from <code>NumericTable</code> .
<code>dependentVariables</code>	Pointer to the $n \times k$ numeric table with responses (k dependent variables). This table can be an object of any class derived from <code>NumericTable</code> .

At the training stage, regression algorithms calculate the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
<code>model</code>	Pointer to the regression model being trained. The result can only be an object of the <code>Model</code> class.

Prediction Stage



At the prediction stage, regression algorithms accept the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
<code>data</code>	Pointer to the $n \times p$ numeric table with the working data set. This table can be an object of any class derived from <code>NumericTable</code> .
<code>model</code>	Pointer to the trained regression model. This input can only be an object of the <code>Model</code> class.

At the prediction stage, regression algorithms calculate the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
<code>prediction</code>	Pointer to the $n \times k$ numeric table with responses (k dependent variables). By default, this table is an object of the <code>HomogenNumericTable</code> class, but you can define it as an object of any class derived from <code>NumericTable</code> except <code>PackedSymmetricMatrix</code> , <code>PackedTriangularMatrix</code> , and <code>CSRNumericTable</code> .

Linear Regression

Linear regression is a method for modeling the relationship between a dependent variable (which may be a vector) and one or more explanatory variables by fitting linear equations to observed data. The case of one explanatory variable is called Simple Linear Regression. For several explanatory variables the method is called Multiple Linear Regression.

Details

Let (x_1, \dots, x_p) be a vector of input variables and $y = (y_1, \dots, y_k)$ be the response. For each $j = 1, \dots, k$, the linear regression model has the format [\[Hastie2009\]](#):

$$y_j = \beta_{0j} + \beta_{1j}x_1 + \dots + \beta_{pj}x_p$$

Here x_i , $i = 1, \dots, p$, are referred to as independent variables, and y_j are referred to as dependent variables or responses.

The linear regression is multiple if the number of input variables $p > 1$.

Training Stage

Let $(x_{11}, \dots, x_{1p}, y_1), \dots, (x_{n1}, \dots, x_{np}, y_n)$ be a set of training data, $n \gg p$. The matrix X of size $n \times p$ contains observations x_{ij} , $i = 1, \dots, n$, $j = 1, \dots, p$, of independent variables.

To estimate the coefficients $(\beta_{0j}, \dots, \beta_{pj})$ one these methods can be used:

- Normal Equation system
- QR matrix decomposition

Prediction Stage

Linear regression based prediction is done for input vector (x_1, \dots, x_p) using the equation $y_j = \beta_{0j} + \beta_{1j}x_1 + \dots + \beta_{pj}x_p$ for each $j = 1, \dots, k$.

See Also

[Quality Metrics for Linear Regression](#)

Ridge Regression

The ridge regression method is similar to the least squares procedure except that it penalizes the sizes of the regression coefficients. Ridge regression is one of the most commonly used methods to overcome data multicollinearity.

Details

Let (x_1, \dots, x_p) be a vector of input variables and $y = (y_1, \dots, y_k)$ be the response. For each $j = 1, \dots, k$, the ridge regression model has the form similar to the linear regression model [Hoerl70], except that the coefficients are estimated by minimizing a different objective function [James2013]:

$$y_j = \beta_{0j} + \beta_{1j}x_1 + \dots + \beta_{pj}x_p.$$

Here x_i , $i = 1, \dots, p$, are referred to as independent variables, and y_j are referred to as dependent variables or responses.

Training Stage

Let $(x_{11}, \dots, x_{1p}, y_{11}, \dots, y_{1k}), \dots, (x_{n1}, \dots, x_{np}, y_{n1}, \dots, y_{nk})$ be a set of training data, $n > p$. The matrix X of size $n \times p$ contains observations x_{ij} , $i = 1, \dots, n$, $j = 1, \dots, p$, of independent variables.

For each y_j , $j = 1, \dots, k$, the ridge regression estimates $(\beta_{0j}, \beta_{1j}, \dots, \beta_{pj})$ by minimizing the objective function:

$$\sum_{i=1}^n \left(y_{ij} - \beta_{0j} - \sum_{q=1}^p \beta_{qj} x_{iq} \right)^2 + \lambda_j \sum_{q=1}^p \beta_{qj}^2,$$

where $\lambda_j \geq 0$ are ridge parameters [Hoerl70, James2013].

Prediction Stage

Ridge regression based prediction is done for input vector (x_1, \dots, x_p) using the equation $y_j = \beta_{0j} + \beta_{1j}x_1 + \dots + \beta_{pj}x_p$ for each $j = 1, \dots, k$.

Computation

Batch Processing

Linear and ridge regressions in the batch processing mode follow the general workflow described in [Usage Model: Training and Prediction](#).

Training

For a description of the input and output, refer to [Usage Model: Training and Prediction](#).

The following table lists parameters of linear and ridge regressions at the training stage. Some of these parameters or their values are specific to a linear or ridge regression algorithm.

Parameter	Algorithm	Default Value	Description
<code>algorithmFPTType</code>	any	double	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<code>method</code>	linear regression	defaultDense	Available methods for linear regression training: <ul style="list-style-type: none"> <code>defaultDense</code> - the normal equations method <code>qrDense</code> - the method based on QR decomposition

Parameter	Algorithm	Default Value	Description
	ridge regression		Default computation method used by the ridge regression. The only method supported at the training stage is the normal equations method.
<i>ridgeParameters</i>	ridge regression	Numeric table of size 1 x 1 that contains the default ridge parameter equal to 1.	<p>The numeric table of size 1 x k (k is the number of dependent variables) or 1 x 1. The contents of the table depend on its size:</p> <ul style="list-style-type: none"> size = 1 x k: values of the ridge parameters λ_j for $j = 1, \dots, k$. size = 1 x 1: the value of the ridge parameter for each dependent variable $\lambda_1 = \dots = \lambda_k$. <p>This parameter can be an object of any class derived from <code>NumericTable</code>, except for <code>PackedTriangularMatrix</code>, <code>PackedSymmetricMatrix</code>, and <code>CSRNumericTable</code>.</p>
<i>interceptFlag</i>	any	true	A flag that indicates a need to compute β_{0j} .

Prediction

For a description of the input and output, refer to [Usage Model: Training and Prediction](#).

At the prediction stage, linear and ridge regressions have the following parameters:

Parameter	Default Value	Description
<i>algorithmFPTYPE</i>	double	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<i>method</i>	defaultDense	Default performance-oriented computation method, the only method supported by the regression based prediction.
<i>interceptFlag</i>	true	A flag that indicates a need to compute β_{0j} .

Examples

Refer to the following examples in your Intel DAAL directory:

C++:

- `./examples/cpp/source/linear_regression/lin_reg_norm_eq_dense_batch.cpp`
- `./examples/cpp/source/linear_regression/lin_reg_qr_dense_batch.cpp`
- `./examples/cpp/source/ridge_regression/ridge_reg_norm_eq_dense_batch.cpp`
- `./examples/cpp/source/quality_metrics/lin_reg_metrics_dense_batch.cpp`

Java*:

- `./examples/java/source/com/intel/daal/examples/linear_regression/LinRegNormEqDenseBatch.java`
- `./examples/java/source/com/intel/daal/examples/linear_regression/LinRegQRDenseBatch.java`
- `./examples/java/source/com/intel/daal/examples/ridge_regression/RidgeRegNormEqDenseBatch.java`
- `./examples/java/source/com/intel/daal/examples/quality_metrics/LinRegMetricsDenseBatch.java`

Python*:

- `./examples/python/source/linear_regression/linear_regression_norm_eq_dense_batch.py`
- `./examples/python/source/linear_regression/lin_reg_qr_dense_batch.py`
- `./examples/python/source/ridge_regression/ridge_reg_norm_eq_dense_batch.py`
- `./examples/python/source/quality_metrics/lin_reg_metrics_dense_batch.py`

Online Processing

You can use linear or ridge regression in the online processing mode only at the training stage.

This computation mode assumes that the data arrives in blocks $i = 1, 2, 3, \dots nblocks$.

Training

Linear or ridge regression training in the online processing mode follows the general workflow described in [Usage Model: Training and Prediction](#).

Linear or ridge regression training in the online processing mode accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
<code>data</code>	Pointer to the $n_i \times p$ numeric table that represents the current, i -th, data block. This table can be an object of any class derived from <code>NumericTable</code> .
<code>dependentVariables</code>	Pointer to the $n_i \times k$ numeric table with responses associated with the current, i -th, data block. This table can be an object of any class derived from <code>NumericTable</code> .

The following table lists parameters of linear and ridge regressions at the training stage in the online processing mode. Some of these parameters or their values are specific to a linear or ridge regression algorithm.

Parameter	Algorithm	Default Value	Description
<code>algorithmFPTYPE</code>	any	double	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<code>method</code>	linear regression	defaultDense	Available methods for linear regression training: <ul style="list-style-type: none"> • <code>defaultDense</code> - the normal equations method

Parameter	Algorithm	Default Value	Description
	ridge regression		<ul style="list-style-type: none"> qrDense - the method based on QR decomposition <p>Default computation method used by the ridge regression. The only method supported at the training stage is the normal equations method.</p>
<i>ridgeParameters</i>	ridge regression	Numeric table of size 1 x 1 that contains the default ridge parameter equal to 1.	<p>The numeric table of size 1 x k (k is the number of dependent variables) or 1 x 1. The contents of the table depend on its size:</p> <ul style="list-style-type: none"> size = 1 x k: values of the ridge parameters λ_j for $j = 1, \dots, k$. size = 1 x 1: the value of the ridge parameter for each dependent variable $\lambda_1 = \dots = \lambda_k$. <p>This parameter can be an object of any class derived from <code>NumericTable</code>, except for <code>PackedTriangularMatrix</code>, <code>PackedSymmetricMatrix</code>, and <code>CSRNumericTable</code>.</p>
<i>interceptFlag</i>	any	true	A flag that indicates a need to compute β_{0j} .

For a description of the output, refer to [Usage Model: Training and Prediction](#).

Examples

Refer to the following examples in your Intel DAAL directory:

C++:

- `./examples/cpp/source/linear_regression/lin_reg_norm_eq_dense_online.cpp`
- `./examples/cpp/source/linear_regression/lin_reg_qr_dense_online.cpp`
- `./examples/cpp/source/ridge_regression/ridge_reg_norm_eq_dense_online.cpp`

Java*:

- `./examples/java/source/com/intel/daal/examples/linear_regression/LinRegNormEqDenseOnline.java`
- `./examples/java/source/com/intel/daal/examples/linear_regression/LinRegQRDenseOnline.java`
- `./examples/java/source/com/intel/daal/examples/ridge_regression/RidgeRegNormEqDenseOnline.java`

Python*:

- `./examples/python/source/linear_regression/lin_reg_norm_eq_dense_online.py`
- `./examples/python/source/linear_regression/lin_reg_qr_dense_online.py`
- `./examples/python/source/ridge_regression/ridge_reg_norm_eq_dense_online.py`

Distributed Processing

You can use linear or ridge regression in the distributed processing mode only at the training stage. This computation mode assumes that the data set is split in *nblocks* blocks across computation nodes.

Training

Algorithm Parameters

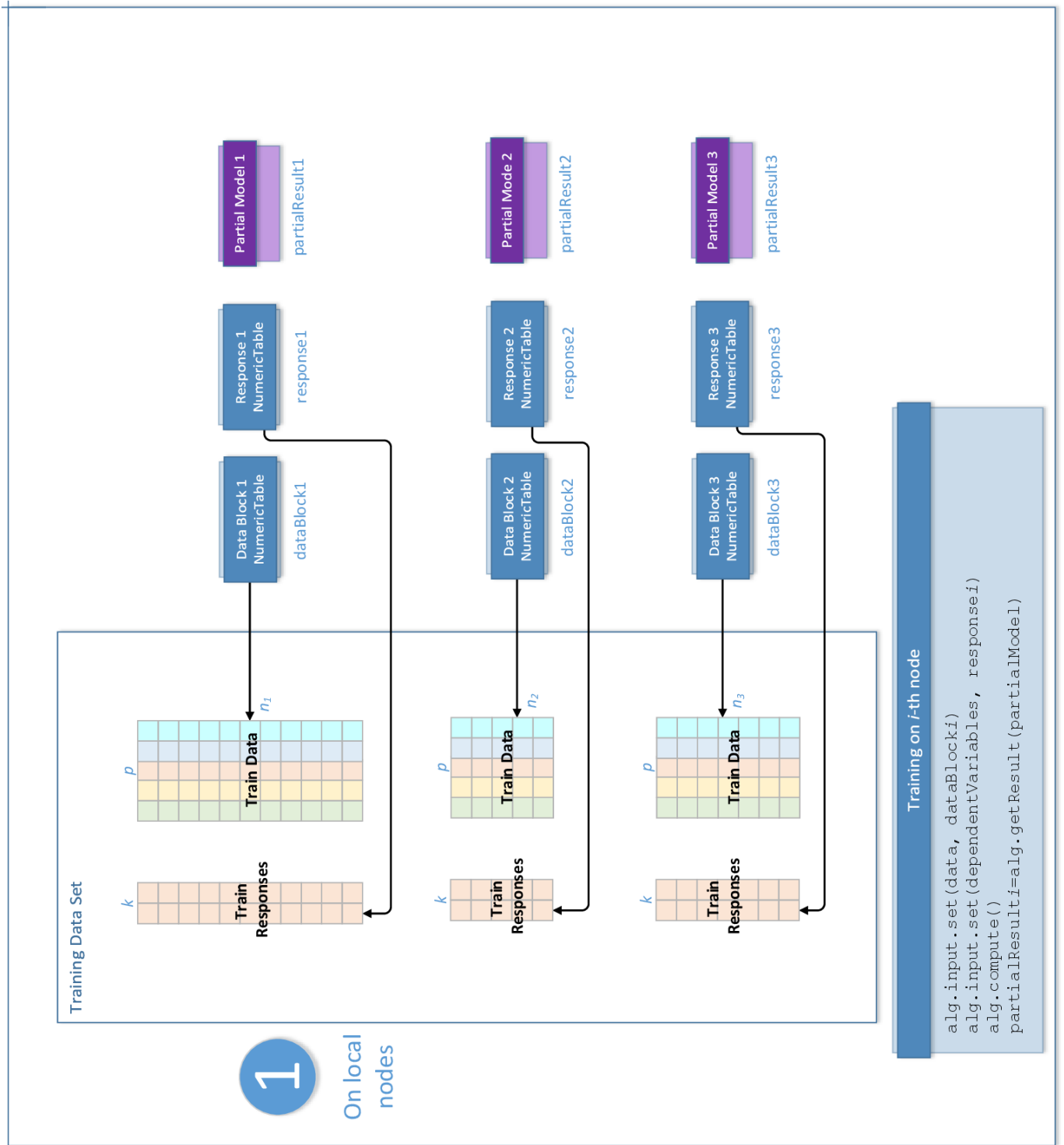
The following table lists parameters of linear and ridge regressions at the training stage in the distributed processing mode. Some of these parameters or their values are specific to a linear or ridge regression algorithm.

Parameter	Algorithm	Default Value	Description
<i>computeStep</i>	any	Not applicable	The parameter required to initialize the algorithm. Can be: <ul style="list-style-type: none"> <i>step1Local</i> - the first step, performed on local nodes <i>step2Master</i> - the second step, performed on a master node
<i>algorithmFPType</i>	any	double	The floating-point type that the algorithm uses for intermediate computations. Can be <i>float</i> or <i>double</i> .
<i>method</i>	linear regression		Available methods for linear regression training: <ul style="list-style-type: none"> <i>defaultDense</i> - the normal equations method <i>qrDense</i> - the method based on QR decomposition
	ridge regression	<i>defaultDense</i>	Default computation method used by the ridge regression. The only method supported at the training stage is the normal equations method.
<i>ridgeParameters</i>	ridge regression	Numeric table of size 1 x 1 that contains the default ridge parameter equal to 1.	The numeric table of size 1 x <i>k</i> (<i>k</i> is the number of dependent variables) or 1 x 1. The contents of the table depend on its size: <ul style="list-style-type: none"> size = 1 x <i>k</i>: values of the ridge parameters λ_j for $j = 1, \dots, k$. size = 1 x 1: the value of the ridge parameter for each dependent variable $\lambda_1 = \dots = \lambda_k$. This parameter can be an object of any class derived from <i>NumericTable</i> , except for

			PackedTriangularMatrix, PackedSymmetricMatrix, and CSRNumericTable.
<i>interceptFlag</i>	any	true	A flag that indicates a need to compute β_{0j} .

Use the two-step computation schema for linear or ridge regression training in the distributed processing mode, as illustrated below:

Step 1 - on Local Nodes



In this step, linear or ridge regression training accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

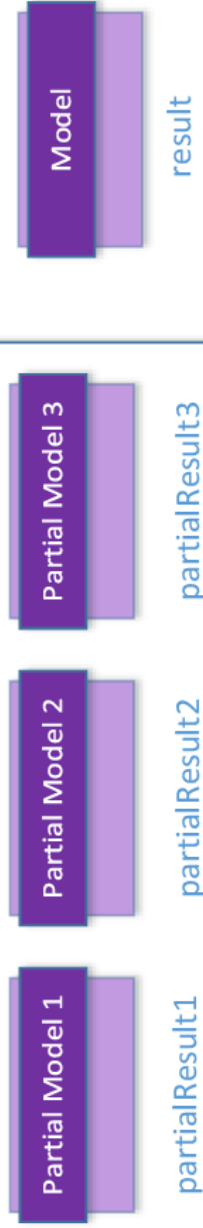
Input ID	Input
<code>data</code>	Pointer to the $n_i \times p$ numeric table that represents the i -th data block on the local node. This table can be an object of any class derived from <code>NumericTable</code> .
<code>dependentVariables</code>	Pointer to the $n_i \times k$ numeric table with responses associated with the i -th data block. This table can be an object of any class derived from <code>NumericTable</code> .
In this step, linear or ridge regression training calculates the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see Algorithms .	
Result ID	Result
<code>partialModel</code>	Pointer to the partial linear regression model that corresponds to the i -th data block. The result can only be an object of the <code>Model</code> class.

Step 2 - on Master Node

2

On
master
nodes

Collection: Partial Models



Training

```
alg.input.add(partialModels, partialResult1)
alg.input.add(partialModels, partialResult2)
alg.input.add(partialModels, partialResult3)
alg.compute()
alg.finalizeCompute()
result=alg.getResult(model)
```

In this step, linear or ridge regression training accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
<code>partialModels</code>	A collection of partial models computed on local nodes in Step 1 . The collection contains objects of the <code>Model</code> class.

In this step, linear or ridge regression training calculates the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
<code>model</code>	Pointer to the linear or ridge regression model being trained. The result can only be an object of the <code>Model</code> class.

Examples

Refer to the following examples in your Intel DAAL directory:

C++:

- `./examples/cpp/source/linear_regression/lin_reg_norm_eq_dense_distr.cpp`
- `./examples/cpp/source/linear_regression/lin_reg_qr_dense_distr.cpp`
- `./examples/cpp/source/ridge_regression/ridge_reg_norm_eq_dense_distr.cpp`

Java*:

- `./examples/java/source/com/intel/daal/examples/linear_regression/LinRegNormEqDenseDistr.java`
- `./examples/java/source/com/intel/daal/examples/linear_regression/LinRegQRDenseDistr.java`
- `./examples/java/source/com/intel/daal/examples/ridge_regression/RidgeRegNormEqDenseDistr.java`

Python*:

- `./examples/python/source/linear_regression/lin_reg_norm_eq_dense_distr.py`
- `./examples/python/source/linear_regression/lin_reg_qr_dense_distr.py`
- `./examples/python/source/ridge_regression/ridge_reg_norm_eq_distr.py`

Classification

Classification methods split observations within a data set into a set of distinct classes by assigning class labels. Because classification is a supervised machine learning method, the training stage requires a data set that consists of both feature vectors and class labels that indicate membership of observations in a particular class. The prediction stage takes a data set (labeled or unlabeled) on input and assigns a class label to each observation.

See Also

[Quality Metrics](#) for classifier evaluation metrics

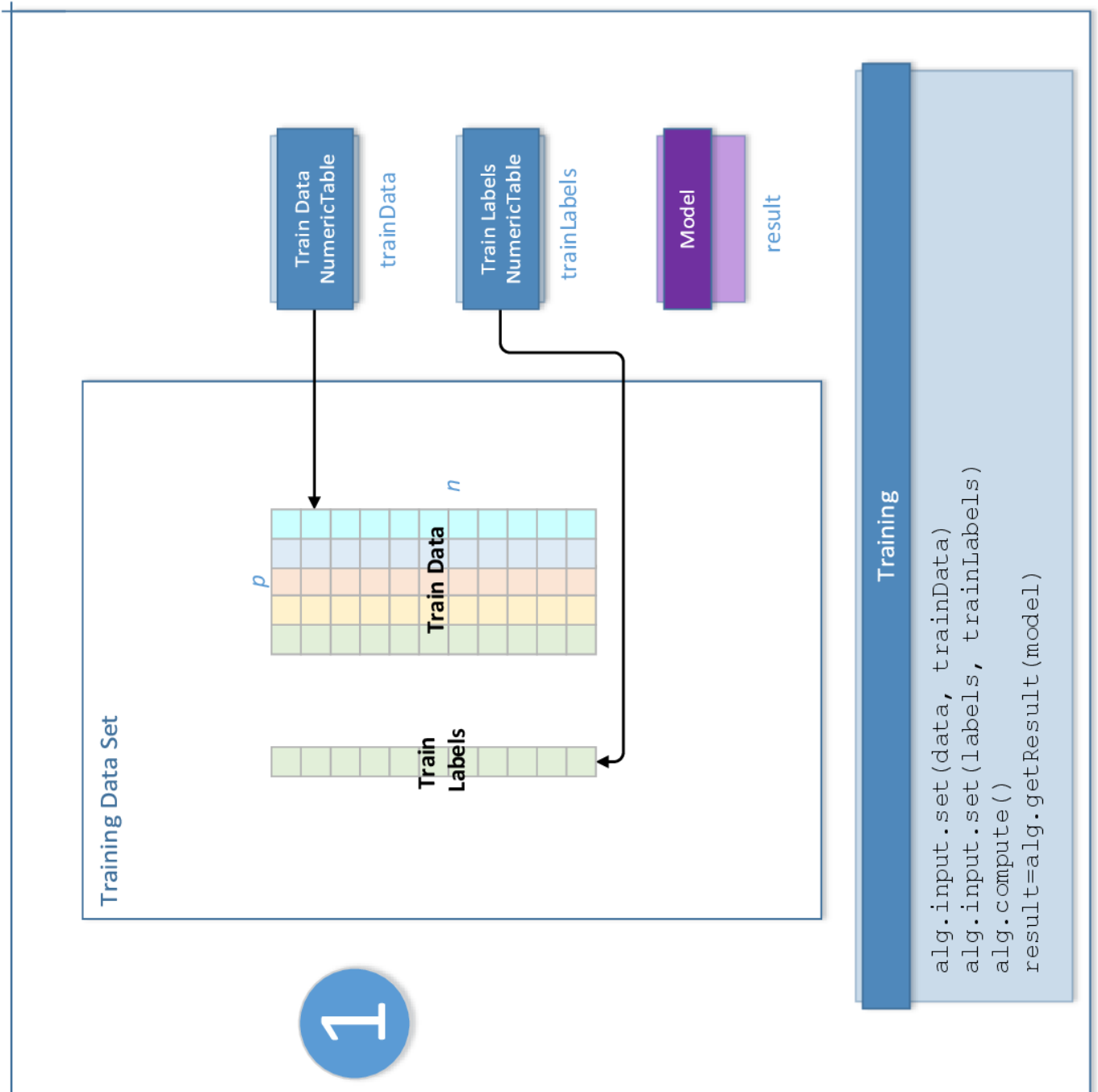
Usage Model: Training and Prediction

A typical workflow for classification methods includes training and prediction, as explained below.

Algorithm-Specific Parameters

The parameters used by classification algorithms at each stage depend on a specific algorithm. For a list of these parameters, refer to the description of an appropriate classification algorithm.

Training Stage



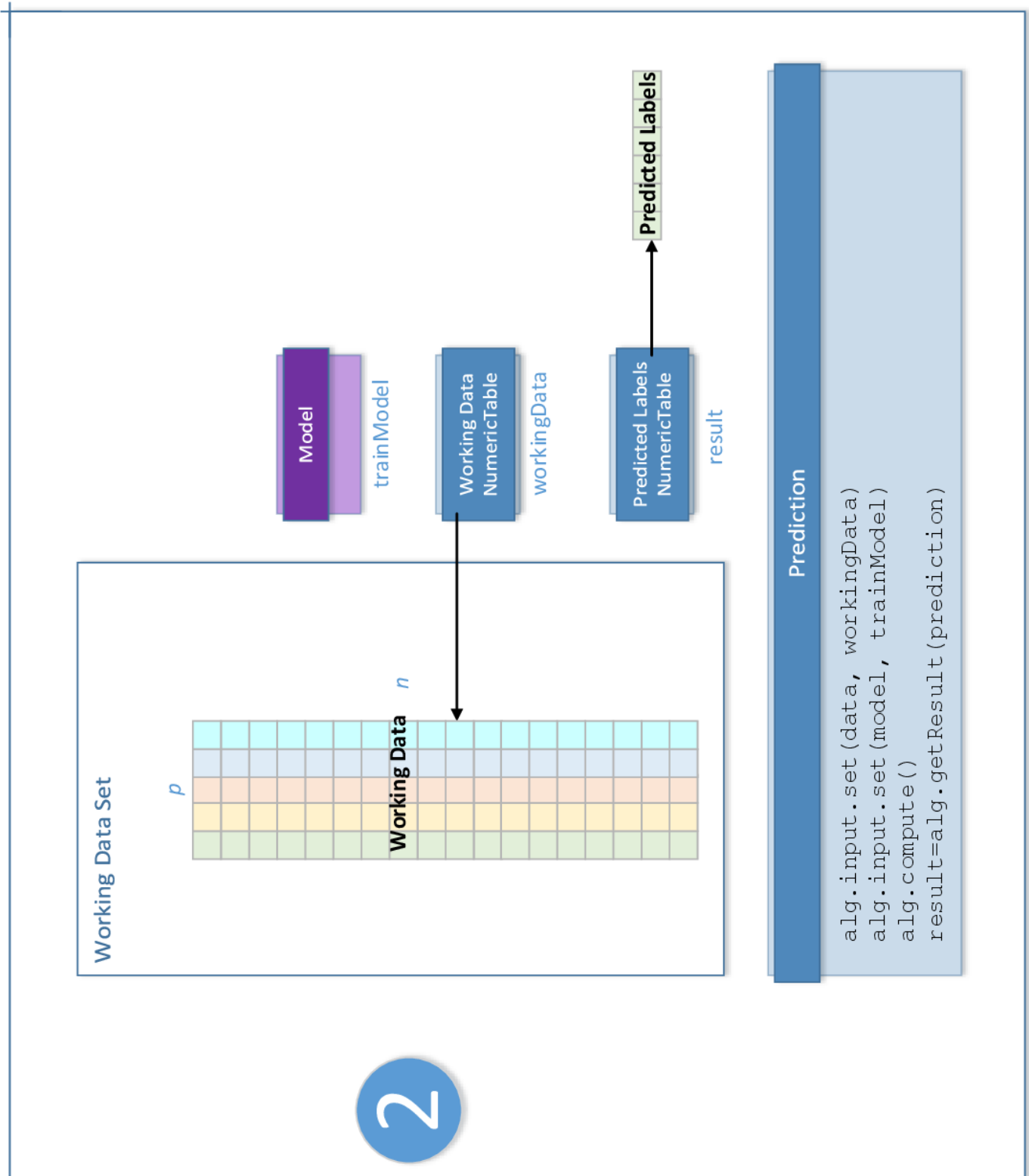
At the training stage, classification algorithms accept the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
data	Pointer to the $n \times p$ numeric table with the training data set. This table can be an object of any class derived from <code>NumericTable</code> .
labels	Pointer to the $n \times 1$ numeric table with class labels. This table can be an object of any class derived from <code>NumericTable</code> .

At the training stage, classification algorithms calculate the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
model	Pointer to the classification model being trained. The result can only be an object of the <code>Model</code> class.

Prediction Stage



At the prediction stage, classification algorithms accept the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
<code>data</code>	Pointer to the $n \times p$ numeric table with the working data set. This table can be an object of any class derived from <code>NumericTable</code> .
<code>model</code>	Pointer to the trained classification model. This input can only be an object of the <code>Model</code> class.

At the prediction stage, classification algorithms calculate the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
<code>prediction</code>	Pointer to the $n \times 1$ numeric table with classification results (class labels or confidence levels). By default, this table is an object of the <code>HomogenNumericTable</code> class, but you can define it as an object of any class derived from <code>NumericTable</code> except <code>PackedSymmetricMatrix</code> , <code>PackedTriangularMatrix</code> , and <code>CSRNumericTable</code> .

Naïve Bayes Classifier

Naïve Bayes is a set of simple and powerful classification methods often used for text classification, medical diagnosis, and other classification problems. In spite of their main assumption about independence between features, Naïve Bayes classifiers often work well when this assumption does not hold. An advantage of this method is that it requires only a small amount of training data to estimate model parameters.

Details

The library provides Multinomial Naïve Bayes classifier [\[Renie03\]](#).

Let J be the number of classes, indexed $0, 1, \dots, J-1$. The integer-valued feature vector $x_i = (x_{i1}, \dots, x_{ip})$, $i=1, \dots, n$, contains scaled frequencies: the value of x_{ik} is the number of times the k -th feature is observed in the vector x_i (in terms of the document classification problem, x_{ik} is the number of occurrences of the word indexed k in the document x_i). For a given data set (a set of n documents), (x_1, \dots, x_n) , the problem is to train a Naïve Bayes classifier.

Training Stage

The Training stage involves calculation of these parameters:

- $$\log(\theta_{jk}) = \log\left(\frac{N_{jk} + \alpha_k}{N_j + \alpha}\right)$$

where N_{jk} is the number of occurrences of the feature k in the class j , N_j is the total number of occurrences of all features in the class, the α_k parameter is the imagined number of occurrences of the feature k (for example, $\alpha_k=1$), and α is the sum of all α_k .
- $\log(p(\theta_j))$, where $p(\theta_j)$ is the prior class estimate.

Prediction Stage

Given a new feature vector x_i , the classifier determines the class the vector belongs to:

$$class(x_i) = \operatorname{argmax}_j (\log(p(\theta_j)) + \sum_k \log(\theta_{jk})).$$

Batch Processing

Naïve Bayes classifier in the batch processing mode follows the general workflow described in [Usage Model: Training and Prediction](#).

Training

For a description of the input and output, refer to [Usage Model: Training and Prediction](#).

At the training stage, Naïve Bayes classifier has the following parameters:

Parameter	Default Value	Description
<i>algorithmFPType</i>	double	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<i>method</i>	defaultDense	Available computation methods for the Naïve Bayes classifier: <ul style="list-style-type: none"> <code>defaultDense</code> - default performance-oriented method <code>fastCSR</code> - performance-oriented method for CSR numeric tables
<i>nClasses</i>	Not applicable	The number of classes, a required parameter.
<i>priorClassEstimates</i>	$1/nClasses$	Vector of size <i>nClasses</i> that contains prior class estimates. The default value applies to each vector element.
<i>alpha</i>	1	Vector of size <i>p</i> that contains the imagined occurrences of features. The default value applies to each vector element.

Prediction

For a description of the input and output, refer to [Usage Model: Training and Prediction](#).

At the prediction stage, Naïve Bayes classifier has the following parameters:

Parameter	Default Value	Description
<i>algorithmFPType</i>	double	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<i>method</i>	defaultDense	Performance-oriented computation method, the only method supported by the algorithm.
<i>nClasses</i>	Not applicable	The number of classes, a required parameter.

Examples

Refer to the following examples in your Intel DAAL directory:

C++:

- `./examples/cpp/source/naive_bayes/multinomial_naive_bayes_dense_batch.cpp`
- `./examples/cpp/source/naive_bayes/multinomial_naive_bayes_csr_batch.cpp`

Java*:

- `./examples/java/source/com/intel/daal/examples/naive_bayes/MultinomialNaiveBayesDenseBatch.java`
- `./examples/java/source/com/intel/daal/examples/naive_bayes/MultinomialNaiveBayesCSRBatch.java`

Python*:

- `./examples/python/source/naive_bayes/multinomial_naive_bayes_dense_batch.py`
- `./examples/python/source/naive_bayes/multinomial_naive_bayes_csr_batch.py`

Online Processing

You can use the Naïve Bayes classifier algorithm in the online processing mode only at the training stage.

This computation mode assumes that the data arrives in blocks $i = 1, 2, 3, \dots, nblocks$.

Training

Naïve Bayes classifier training in the online processing mode follows the general workflow described in [Usage Model: Training and Prediction](#).

Naïve Bayes classifier in the online processing mode accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
<code>data</code>	Pointer to the $n_i \times p$ numeric table that represents the current data block. This table can be an object of any class derived from <code>NumericTable</code> .
<code>labels</code>	Pointer to the $n_i \times 1$ numeric table with class labels associated with the current data block. This table can be an object of any class derived from <code>NumericTable</code> .

Naïve Bayes classifier in the online processing mode has the following parameters:

Parameter	Default Value	Description
<code>algorithmFPType</code>	<code>double</code>	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<code>method</code>	<code>defaultDense</code>	Available computation methods for the Naïve Bayes classifier: <ul style="list-style-type: none"> • <code>defaultDense</code> - default performance-oriented method • <code>fastCSR</code> - performance-oriented method for CSR numeric tables
<code>nClasses</code>	Not applicable	The number of classes, a required parameter.
<code>priorClassEstimates</code>	<code>1/nClasses</code>	Vector of size <code>nClasses</code> that contains prior class estimates. The default value applies to each vector element.
<code>alpha</code>	<code>1</code>	Vector of size p that contains the imagined occurrences of features. The default value applies to each vector element.

For a description of the output, refer to [Usage Model: Training and Prediction](#).

Examples

Refer to the following examples in your Intel DAAL directory:

C++:

- `./examples/cpp/source/naive_bayes/multinomial_naive_bayes_dense_online.cpp`
- `./examples/cpp/source/naive_bayes/multinomial_naive_bayes_csr_online.cpp`

Java*:

- `./examples/java/source/com/intel/daal/examples/naive_bayes/MultinomialNaiveBayesDenseOnline.java`
- `./examples/java/source/com/intel/daal/examples/naive_bayes/MultinomialNaiveBayesCSROnline.java`

Python*:

- `./examples/python/source/naive_bayes/multinomial_naive_bayes_dense_online.py`
- `./examples/python/source/naive_bayes/multinomial_naive_bayes_csr_online.py`

Distributed Processing

You can use the Naïve Bayes classifier algorithm in the distributed processing mode only at the training stage.

This computation mode assumes that the data set is split in *nblocks* blocks across computation nodes.

Training

Algorithm Parameters

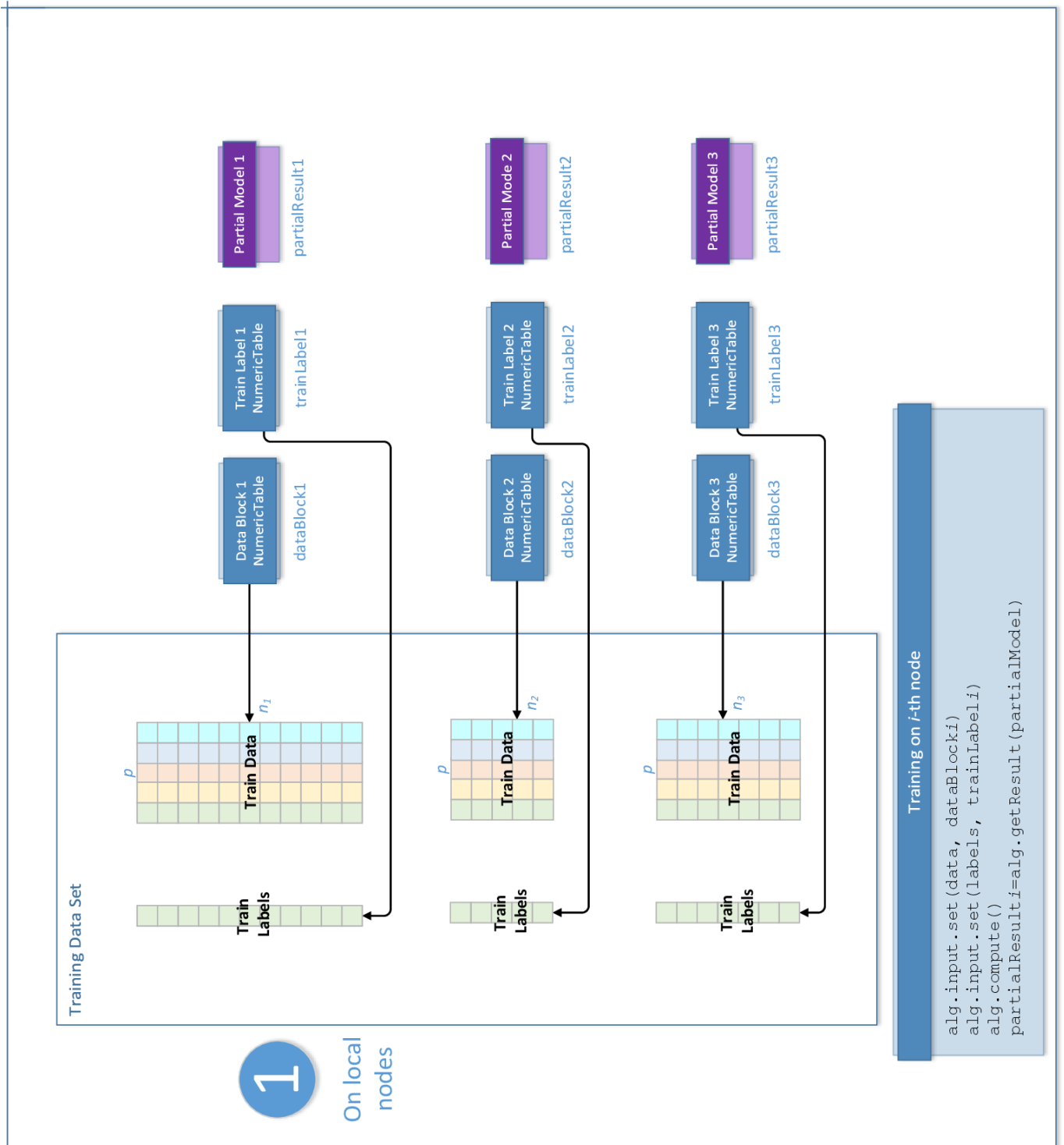
At the training stage, Naïve Bayes classifier in the distributed processing mode has the following parameters:

Parameter	Default Value	Description
<i>computeStep</i>	Not applicable	The parameter required to initialize the algorithm. Can be: <ul style="list-style-type: none"> • <code>step1Local</code> - the first step, performed on local nodes • <code>step2Master</code> - the second step, performed on a master node
<i>algorithmFPType</i>	double	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<i>method</i>	defaultDense	Available computation methods for the Naïve Bayes classifier: <ul style="list-style-type: none"> • <code>defaultDense</code> - default performance-oriented method • <code>fastCSR</code> - performance-oriented method for CSR numeric tables
<i>nClasses</i>	Not applicable	The number of classes, a required parameter.
<i>priorClassEstimates</i>	$1/nClasses$	Vector of size <i>nClasses</i> that contains prior class estimates. The default value applies to each vector element.

Parameter	Default Value	Description
<i>alpha</i>	1	Vector of size p that contains the imagined occurrences of features. The default value applies to each vector element.

Use the two-step computation schema for Naïve Bayes classifier training in the distributed processing mode, as illustrated below:

Step 1 - on Local Nodes



In this step, Naïve Bayes classifier training accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
data	Pointer to the $n_i \times p$ numeric table that represents the i -th data block on the local node. This table can be an object of any class derived from <code>NumericTable</code> .
labels	Pointer to the $n_i \times 1$ numeric table with class labels associated with the i -th data block. This table can be an object of any class derived from <code>NumericTable</code> .

In this step, Naïve Bayes classifier training calculates the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
partialModel	Pointer to the partial Naïve Bayes classifier model that corresponds to the i -th data block. The result can only be an object of the <code>Model</code> class.

Step 2 - on Master Node

2

On
master
nodes

Collection: Partial Models

Partial Model 1

partialResult1

Partial Model 2

partialResult2

Partial Model 3

partialResult3

Model

result

Training

```
alg.input.add(partialModels, partialResult1)
alg.input.add(partialModels, partialResult2)
alg.input.add(partialModels, partialResult3)
alg.compute()
alg.finalizeCompute()
result=alg.getResult(model)
```


In this step, Naïve Bayes classifier training accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
<code>partialModels</code>	A collection of partial models computed on local nodes in Step 1 . The collection contains objects of the <code>Model</code> class.

In this step, Naïve Bayes classifier training calculates the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
<code>model</code>	Pointer to the Naïve Bayes classifier model being trained. The result can only be an object of the <code>Model</code> class.

Examples

Refer to the following examples in your Intel DAAL directory:

C++:

- `./examples/cpp/source/naive_bayes/multinomial_naive_bayes_dense_distributed.cpp`
- `./examples/cpp/source/naive_bayes/multinomial_naive_bayes_csr_distributed.cpp`

Java*:

- `./examples/java/source/com/intel/daal/examples/naive_bayes/MultinomialNaiveBayesDenseDistributed.java`
- `./examples/java/source/com/intel/daal/examples/naive_bayes/MultinomialNaiveBayesCSR Distributed.java`

Python*:

- `./examples/python/source/naive_bayes/multinomial_naive_bayes_dense_distributed.py`
- `./examples/python/source/naive_bayes/multinomial_naive_bayes_csr_distributed.py`

Performance Considerations

Training Stage

To get the best overall performance at the Naïve Bayes classifier training stage:

- If input data is homogeneous:
 - For the training data set, use a homogeneous numeric table of the same type as specified in the `algorithmFPTYPE` class template parameter.
 - For class labels, use a homogeneous numeric table of type `int`.
- If input data is non-homogeneous, use AOS layout rather than SOA layout.

The training stage of the Naïve Bayes classifier algorithm is memory access bound in most cases. Therefore, use efficient data layout whenever possible.

Prediction Stage

To get the best overall performance at the Naïve Bayes classifier prediction stage:

- For the working data set, use a homogeneous numeric table of the same type as specified in the `algorithmFPTYPE` class template parameter.

- For predicted labels, use a homogeneous numeric table of type `int`.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Boosting

Boosting is a set of algorithms intended to build a strong classifier from an ensemble of weighted weak classifiers by iterative re-weighting according to some accuracy measure for weak classifiers. A weak learner is a classifier that has only slightly better performance than random guessing. Weak learners are usually very simple and fast, and they focus on classification of very specific features.

Boosting algorithms include LogitBoost, BrownBoost, AdaBoost, and others. A Decision Stump classifier is one of popular weak learners.

In Intel DAAL, a Weak Learner is a set of interface classes that define methods to enable use of a weak learner classifier. The Weak Learner classes include `Training`, `Prediction`, and `Model`. Specific weak learner classifiers, such as `stump`, implement those methods.

Intel DAAL boosting algorithms pass pointers to weak learner training and prediction objects through the parameters of boosting algorithms. Use the `getNumberOfWeakLearners()` method to determine the number of weak learners trained.

You can implement your own weak learners by deriving from the appropriate interface classes.

NOTE

When defining your own weak learners to use with boosting classifiers, make sure the prediction component of your weak learner returns the number from the interval $[-1.0, 1.0]$.

Performance Considerations

Formally you can use any classifier as a weak classifier in boosting algorithms. However, be aware about the following tradeoffs:

- Try to use a weak learner model with lower bias, but avoid over-fitting.
- Ensure the training time of a weak learner is reasonably low. Because an ensemble of weak learners is used for boosting, the overall training time may be hundreds or thousands times greater than the training time of a single weak learner.
- Ensure the prediction time of a weak learner is low. The boosting prediction rate is a lot slower than a single weak learner prediction rate.

In most cases, to achieve the best performance of a boosting algorithm, use the layout of an input and output numeric tables that is preferable for the underlying weak learner.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain

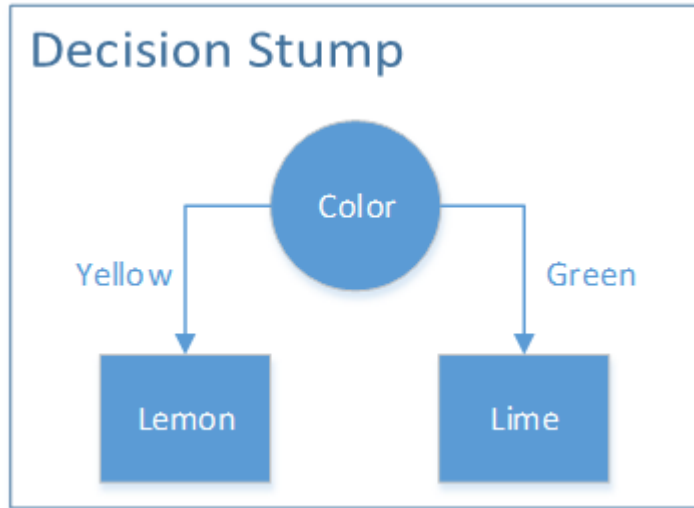
Optimization Notice

optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Stump Weak Learner Classifier

A decision stump is a model that consists of a one-level decision tree [Iba92] where the root is connected to terminal nodes (leaves). The library only supports stumps with two leaves.

*Batch Processing*

A stump weak learner classifier follows the general workflow described in [Usage Model: Training and Prediction](#).

Training

For a description of the input and output, refer to [Usage Model: Training and Prediction](#).

At the training stage, a stump weak learner classifier has the following parameters:

Parameter	Default Value	Description
<code>algorithmFPType</code>	<code>double</code>	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<code>method</code>	<code>defaultDense</code>	Performance-oriented computation method, the only method supported by the algorithm.

Prediction

For a description of the input and output, refer to [Usage Model: Training and Prediction](#).

At the prediction stage, a stump weak learner classifier has the following parameters:

Parameter	Default Value	Description
<code>algorithmFPTType</code>	<code>double</code>	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<code>method</code>	<code>defaultDense</code>	Performance-oriented computation method, the only method supported by the algorithm.

Examples

Refer to the following examples in your Intel DAAL directory:

C++: `./examples/cpp/source/stump/stump_batch.cpp`

Java*: `./examples/java/source/com/intel/daal/examples/stump/StumpBatch.java`

Python*: `./examples/python/source/stump/stump_batch.py`

Performance Considerations

For the best performance of a stump weak learner classifier:

- Whenever possible, represent the decision stump using the SOA layout.
- For input and output numeric tables, use the same data type as specified in the `algorithmFPTType` template parameter for the model representation and computations.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

AdaBoost Classifier

AdaBoost (short for "Adaptive Boosting") is a popular boosting classification algorithm. AdaBoost algorithm performs well on a variety of data sets except some noisy data [Freund99].

AdaBoost is a binary classifier. For a multi-class case, use [Multi-Class Classifier](#) framework of the library.

See Also

[Quality Metrics for Binary Classification Algorithms](#)

Details

Given n feature vectors $x_1=(x_{11},\dots,x_{1p}),\dots,x_n=(x_{n1},\dots,x_{np})$ of size p , a vector of class labels $y=(y_1,\dots,y_n)$, where $y_i\in K=\{-1, 1\}$ describes the class to which the feature vector x_i belongs, and a weak learner algorithm, the problem is to build an AdaBoost classifier.

Training Stage

The following scheme shows the major steps of the algorithm:

1. Initialize weights $D_1(i) = 1/n$ for $i = 1,\dots,n$

2. For $t = 1, \dots, T$:
 - i. Train the weak learner $h_t(t) \in \{-1, 1\}$ using weights D_t
 - ii. Choose a confidence value α_t
 - iii. Update

$$D_{t+1}(i) = \frac{D_t(i) \exp(-\alpha_t Y_i h_t(x_i))}{Z_t},$$

where Z_t is a normalization factor

3. Output the final hypothesis:

$$H(x) = \text{sign} \left(\sum_{t=1}^T \alpha_t h_t(x) \right)$$

Prediction Stage

Given the AdaBoost classifier and r feature vectors x_1, \dots, x_r , the problem is to calculate the final class

$$H(x_i) = \text{sign} \left(\sum_{t=1}^T \alpha_t h_t(x_i) \right).$$

Batch Processing

AdaBoost classifier follows the general workflow described in [Usage Model: Training and Prediction](#).

Training

For a description of the input and output, refer to [Usage Model: Training and Prediction](#).

At the training stage, an AdaBoost classifier has the following parameters:

Parameter	Default Value	Description
<code>algorithmFPType</code>	double	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<code>method</code>	defaultDense	The computation method used by the AdaBoost classifier. The only training method supported so far is the Y. Freund's method.
<code>weakLearnerTraining</code>	Pointer to an object of the stump training class	Pointer to the training algorithm of the weak learner. By default, a stump weak learner is used.
<code>weakLearnerPrediction</code>	Pointer to an object of the stump prediction class	Pointer to the prediction algorithm of the weak learner. By default, a stump weak learner is used.
<code>accuracyThreshold</code>	0.01	AdaBoost training accuracy.
<code>maxIterations</code>	100	The maximal number of iterations for the algorithm.

Prediction

For a description of the input and output, refer to [Usage Model: Training and Prediction](#).

At the prediction stage, an AdaBoost classifier has the following parameters:

Parameter	Default Value	Description
<code>algorithmFPTType</code>	double	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<code>method</code>	defaultDense	Performance-oriented computation method, the only method supported by the AdaBoost classifier at the prediction stage.
<code>weakLearnerPrediction</code>	Pointer to an object of the stump prediction class	Pointer to the prediction algorithm of the weak learner. By default, a stump weak learner is used.

Examples

Refer to the following examples in your Intel DAAL directory:

C++: `./examples/cpp/source/boosting/adaboost_batch.cpp`

Java*: `./examples/java/source/com/intel/daal/examples/boosting/AdaBoostBatch.java`

Python*: `./examples/python/source/boosting/adaboost_batch.py`

BrownBoost Classifier

BrownBoost is a boosting classification algorithm. It is more robust to noisy data sets than other boosting classification algorithms [Freund99].

BrownBoost is a binary classifier. For a multi-class case, use [Multi-Class Classifier](#) framework of the library.

See Also

Quality Metrics for Binary Classification Algorithms

Details

Given n feature vectors $x_1=(x_{11},\dots,x_{1p}),\dots,x_n=(x_{n1},\dots,x_{np})$ of size p , a vector of class labels $y=(y_1,\dots,y_n)$, where $y_i \in K = \{-1, 1\}$ describes the class to which the feature vector x_i belongs, and a weak learner algorithm, the problem is to build a two-class BrownBoost classifier.

Training Stage

The model is trained using the Freund method [Freund01] as follows:

1. Calculate $c = \operatorname{erfinv}^2(1 - \varepsilon)$, where
 $\operatorname{erfinv}(x)$ is an inverse error function,
 ε is a target classification error of the algorithm defined as

$$\frac{1}{n} \sum_{i=1}^n |p(x_i) - y_i|,$$

$$p(x) = \operatorname{erf}\left(\frac{\sum_{i=1}^M a_i h_i(x)}{\sqrt{c}}\right),$$

$\operatorname{erf}(x)$ is the error function,

$h_i(x)$ is a hypothesis formulated by the i -th weak learner, $i = 1, \dots, M$,

a_i is the weight of the hypothesis.

2. Set initial prediction values: $r_1(x, y) = 0$.

3. Set "remaining timing": $s_1 = c$.

4. Do for $i=1,2,\dots$ until $s_{i+1} \leq 0$

i. With each feature vector and its label of positive weight, associate

$$W_i(x, y) = e^{-\left(r_i(x, y) + s_i\right)^2 / c}$$

ii. Call the weak learner with the distribution defined by normalizing $W_i(x, y)$ to receive a hypothesis $h_i(x)$

iii. Solve the differential equation

$$\frac{dt}{d\alpha} = \gamma = \frac{\sum_{(x, y)} \exp\left(-\frac{1}{c}\left(r_i(x, y) + \alpha h_i(x)y + s_i - t\right)^2\right) h_i(x)y}{\sum_{(x, y)} \exp\left(-\frac{1}{c}\left(r_i(x, y) + \alpha h_i(x)y + s_i - t\right)^2\right)}$$

with given boundary conditions $t = 0$ and $\alpha = 0$ to find $t_i = t^* > 0$ and $\alpha_i = \alpha^*$ such that either $\gamma \leq v$ or $t^* = s_i$, where v is a given small constant needed to avoid degenerate cases

iv. Update the prediction values: $r_{i+1}(x, y) = r_i(x, y) + \alpha_i h_i(x)y$

v. Update "remaining time": $s_{i+1} = s_i - t_i$

End do

The result of the model training is the array of M weak learners h_i .

Prediction Stage

Given the BrownBoost classifier and r feature vectors x_1, \dots, x_r , the problem is to calculate the final classification confidence, a number from the interval $[-1, 1]$, using the rule

$$p(x) = \operatorname{erf}\left(\frac{\sum_{i=1}^M \alpha_i h_i(x)}{\sqrt{c}}\right).$$

Batch Processing

BrownBoost classifier follows the general workflow described in [Usage Model: Training and Prediction](#).

Training

For a description of the input and output, refer to [Usage Model: Training and Prediction](#).

At the training stage, a BrownBoost classifier has the following parameters:

Parameter	Default Value	Description
<i>algorithmFPType</i>	double	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<i>method</i>	defaultDense	The computation method used by the BrownBoost classifier. The only training method supported so far is the Y. Freund's method.
<i>weakLearnerTraining</i>	Pointer to an object of the weak learner training class	Pointer to the training algorithm of the weak learner. By default, a stump weak learner is used.
<i>weakLearnerPrediction</i>	Pointer to an object of the weak learner prediction class	Pointer to the prediction algorithm of the weak learner. By default, a stump weak learner is used.

Parameter	Default Value	Description
<i>accuracyThreshold</i>	0.01	BrownBoost training accuracy ϵ .
<i>maxIterations</i>	100	The maximal number of iterations for the BrownBoost algorithm.
<i>newtonRaphsonAccuracyThreshold</i>	1.0e-3	Accuracy threshold of the Newton-Raphson method used underneath the BrownBoost algorithm.
<i>newtonRaphsonMaxIterations</i>	100	The maximal number of Newton-Raphson iterations in the algorithm.
<i>degenerateCasesThreshold</i>	1.0e-2	The threshold used to avoid degenerate cases.

Prediction

For a description of the input and output, refer to [Usage Model: Training and Prediction](#).

At the prediction stage, a BrownBoost classifier has the following parameters:

Parameter	Default Value	Description
<i>algorithmFPTType</i>	double	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<i>method</i>	defaultDense	Performance-oriented computation method, the only method supported by the BrownBoost classifier.
<i>weakLearnerPrediction</i>	Pointer to an object of the weak learner prediction class	Pointer to the prediction algorithm of the weak learner. By default, a stump weak learner is used.
<i>accuracyThreshold</i>	0.01	BrownBoost training accuracy ϵ .

Examples

Refer to the following examples in your Intel DAAL directory:

C++: `./examples/cpp/source/boosting/brownboost_batch.cpp`

Java*: `./examples/java/source/com/intel/daal/examples/boosting/BrownBoostBatch.java`

Python*: `./examples/python/source/boosting/brownboost_batch.py`

LogitBoost Classifier

LogitBoost is a boosting classification algorithm. LogitBoost and AdaBoost are close to each other in the sense that both perform an additive logistic regression. The difference is that AdaBoost minimizes the exponential loss, whereas LogitBoost minimizes the logistic loss.

LogitBoost within Intel DAAL implements a multi-class classifier.

See Also

[Quality Metrics for Multi-class Classification Algorithms](#)

Details

Given n feature vectors $x_1=(x_{11},\dots,x_{1p}),\dots,x_n=(x_{n1},\dots,x_{np})$ of size p and a vector of class labels $y=(y_1,\dots,y_n)$, where $y_i \in K = \{0, \dots, J-1\}$ describes the class to which the feature vector x_i belongs and J is the number of classes, the problem is to build a multi-class LogitBoost classifier.

Training Stage

The LogitBoost model is trained using the Friedman method [Friedman00].

Let $y_{i,j} = I\{x_i \in j\}$ is the indicator that the i -th feature vector belongs to class j . The scheme below, which uses the stump weak learner, shows the major steps of the algorithm:

1. Start with weights $w_{ij} = 1/n$, $F_j(x) = 0$, $p_j(x) = 1/J$, $i = 1, \dots, n$, $j = 0, \dots, J-1$
2. For $m=1, \dots, M$

Do

For $j = 1, \dots, J$

Do

(i) Compute working responses and weights in the j -th class:

$$w_{ij} = p_i(x_i)(1-p_i(x_i)), w_{ij} = \max(z_{ij}, \text{Thr1})$$

$$z_{ij} = (y_{ij} - p_i(x_i)) / w_{ij}, z_{ij} = \min(\max(z_{ij}, -\text{Thr2}), \text{Thr2})$$

(ii) Fit the function $f_{mj}(x)$ by a weighted least-squares regression of z_{ij} to x_i with weights w_{ij} using the stump-based approach.

End do

$$f_{mj}(x) = \frac{J-1}{J} \left(f_{mj}(x) - \frac{1}{J} \sum_{k=1}^J f_{mk}(x) \right)$$

$$F_j(x) = F_j(x) + f_{mj}(x)$$

$$p_j(x) = \frac{e^{F_j(x)}}{\sum_{k=1}^J e^{F_k(x)}}$$

End do

The result of the model training is a set of M stumps.

Prediction Stage

Given the LogitBoost classifier and r feature vectors x_1, \dots, x_r , the problem is to calculate the labels $\text{argmax}_j F_j(x)$ of the classes to which the feature vectors belong.

Batch Processing

LogitBoost classifier follows the general workflow described in [Usage Model: Training and Prediction](#).

Training

For a description of the input and output, refer to [Usage Model: Training and Prediction](#).

At the training stage, a LogitBoost classifier has the following parameters:

Parameter	Default Value	Description
<i>algorithmFPTType</i>	double	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<i>method</i>	defaultDense	The computation method used by the LogitBoost classifier. The only training method supported so far is the Friedman method.
<i>weakLearnerTraining</i>	Pointer to an object of the stump training class	Pointer to the training algorithm of the weak learner. By default, a stump weak learner is used.
<i>weakLearnerPrediction</i>	Pointer to an object of the stump prediction class	Pointer to the prediction algorithm of the weak learner. By default, a stump weak learner is used.
<i>accuracyThreshold</i>	0.01	LogitBoost training accuracy.
<i>maxIterations</i>	100	The maximal number of iterations for the LogitBoost algorithm.
<i>nClasses</i>	Not applicable	The number of classes, a required parameter.
<i>weightsDegenerateCasesThreshold</i>	1e-10	The threshold to avoid degenerate cases when calculating weights w_{ij} .
<i>responsesDegenerateCasesThreshold</i>	1e-10	The threshold to avoid degenerate cases when calculating responses z_{ij} .

Prediction

For a description of the input and output, refer to [Usage Model: Training and Prediction](#).

At the prediction stage, a LogitBoost classifier has the following parameters:

Parameter	Default Value	Description
<i>algorithmFPTType</i>	double	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<i>method</i>	defaultDense	Performance-oriented computation method, the only method supported by the LogitBoost classifier at the prediction stage.
<i>weakLearnerPrediction</i>	Pointer to an object of the stump prediction class	Pointer to the prediction algorithm of the weak learner. By default, a stump weak learner is used.
<i>nClasses</i>	Not applicable	The number of classes, a required parameter.

NOTE

The algorithm terminates if it achieves the specified accuracy or reaches the specified maximal number of iterations. To determine the actual number of iterations performed, call the `getNumberOfWeakLearners()` method of the `LogitBoostModel` class and divide it by *nClasses*.

Examples

Refer to the following examples in your Intel DAAL directory:

C++: `./examples/cpp/source/boosting/logitboost_batch.cpp`

Java*: `./examples/java/source/com/intel/daal/examples/boosting/LogitBoostBatch.java`

Python*: `./examples/python/source/boosting/logitboost_batch.py`

Support Vector Machine Classifier

Support Vector Machine (SVM) is among popular classification algorithms. It belongs to a family of generalized linear classification problems. Because SVM covers binary classification problems only in the multi-class case, SVM must be used in conjunction with multi-class classifier methods.

SVM is a binary classifier. For a multi-class case, use [Multi-Class Classifier](#) framework of the library.

See Also

[Quality Metrics for Binary Classification Algorithms](#)

Details

Given n feature vectors $x_1 = (x_{11}, \dots, x_{1p})$, ..., $x_n = (x_{n1}, \dots, x_{np})$ of size p and a vector of class labels $y = (y_1, \dots, y_n)$, where $y_i \in \{-1, 1\}$ describes the class to which the feature vector x_i belongs, the problem is to build a two-class Support Vector Machine (SVM) classifier.

Training Stage

The SVM model is trained using the Boser method [[Boser92](#)] reduced to the solution of the quadratic optimization problem

$$\min_{\alpha} \frac{1}{2} \alpha^T Q \alpha - e^T \alpha$$

with $0 \leq \alpha_i \leq C$, $i = 1, \dots, n$, $y^T \alpha = 0$,

where e is the vector of ones, C is the upper bound of the coordinates of the vector α , Q is a symmetric matrix of size $n \times n$ with $Q_{ij} = y_i y_j K(x_i, x_j)$, and $K(x, y)$ is a kernel function.

Working subset of α updated on each iteration of the algorithm is based on the Working Set Selection (WSS) 3 scheme [[Fan05](#)]. The scheme can be optimized using one of these techniques or both:

- *Cache*.

The implementation can allocate a predefined amount of memory to store intermediate results of the kernel computation.

- *Shrinking*.

The implementation can try to decrease the amount of kernel related computations (see [[Joachims99](#)]).

The solution of the problem defines the separating hyperplane and corresponding decision function $D(x) = \sum_k y_k \alpha_k K(x_k, x) + b$ where only those x_k that correspond to non-zero α_k appear in the sum, and b is a bias. Each non-zero α_k is called a *classification coefficient* and the corresponding x_k is called a *support vector*.

Prediction Stage

Given the SVM classifier and r feature vectors x_1, \dots, x_r , the problem is to calculate the signed value of the decision function $D(x_i)$, $i=1, \dots, r$. The sign of the value defines the class of the feature vector, and the absolute value of the function is a multiple of the distance between the feature vector and separating hyperplane.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Batch Processing

SVM classifier follows the general workflow described in [Usage Model: Training and Prediction](#).

Training

For a description of the input and output, refer to [Usage Model: Training and Prediction](#).

At the training stage, SVM classifier has the following parameters:

Parameter	Default Value	Description
<i>algorithmFPType</i>	double	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<i>method</i>	defaultDense	The computation method used by the SVM classifier. The only training method supported so far is the Boser method.
<i>C</i>	1	Upper bound in conditions of the quadratic optimization problem.
<i>accuracyThreshold</i>	0.001	The training accuracy.
<i>tau</i>	1.0e-6	Tau parameter of the WSS scheme.
<i>maxIterations</i>	1000000	Maximal number of iterations for the algorithm.
<i>cacheSize</i>	8000000	Size of cache in bytes for storing values of the kernel matrix. A non-zero value enables use of a cache optimization technique.
<i>doShrinking</i>	true	A flag that enables use of a shrinking optimization technique.
<i>kernel</i>	Pointer to an object of the <code>KernelIface</code> class	The kernel function. By default, the algorithm uses a linear kernel. For details, see Kernel Functions .

Prediction

For a description of the input and output, refer to [Usage Model: Training and Prediction](#).

At the prediction stage, SVM classifier has the following parameters:

Parameter	Default Value	Description
<i>algorithmFPTYPE</i>	double	The floating-point type that the algorithm uses for intermediate computations. Can be float or double.
<i>method</i>	defaultDense	Performance-oriented computation method, the only prediction method supported by the algorithm.
<i>kernel</i>	Pointer to object of the <code>KernelIface</code> class	The kernel function. By default, the algorithm uses a linear kernel.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Examples

Refer to the following examples in your Intel DAAL directory:

C++:

- `./examples/cpp/source/svm/svm_two_class_dense_batch.cpp`
- `./examples/cpp/source/svm/svm_two_class_csr_batch.cpp`
- `./examples/cpp/source/quality_metrics/svm_two_class_quality_metric_set_batch.cpp`

Java*:

- `./examples/java/source/com/intel/daal/examples/svm/SVMTwoClassDenseBatch.java`
- `./examples/java/source/com/intel/daal/examples/svm/SVMTwoClassCSRBatch.java`
- `./examples/java/source/com/intel/daal/examples/quality_metrics/SVMTwoClassQualityMetricSetBatchExample.java`

Python*:

- `./examples/python/source/svm/svm_two_class_dense_batch.py`
- `./examples/python/source/svm/svm_two_class_csr_batch.py`
- `./examples/python/source/quality_metrics/svm_two_class_quality_metric_set_batch.py`

Performance Considerations

For the best performance of the SVM classifier, use homogeneous numeric tables if your input data set is homogeneous or SOA numeric tables otherwise.

Performance of the SVM algorithm greatly depends on the cache size *cacheSize*. Larger cache size typically results in greater performance. For the best SVM algorithm performance, use *cacheSize* equal to $n^2 * \text{sizeof}(\text{algorithmFPTYPE})$. However, avoid setting the cache size to a larger value than the number of bytes required to store n^2 data elements because the algorithm does not fully utilize the cache in this case.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Multi-class Classifier

While some classification algorithms naturally permit the use of more than two classes, some algorithms, such as Support Vector Machines (SVM), are by nature solving a two-class problem only. These two-class (or binary) classifiers can be turned into multi-class classifiers by using different strategies, such as One-Against-Rest or One-Against-One.

Intel DAAL implements a Multi-Class Classifier using the One-Against-One strategy.

Multi-class classifiers, such as SVM, are based on two-class classifiers, which are integral components of the models trained with the corresponding multi-class classifier algorithms.

See Also

[Quality Metrics for Multi-class Classification Algorithms](#)

Details

Given n feature vectors $x_1=(x_{11},\dots,x_{1p}),\dots,x_n=(x_{n1},\dots,x_{np})$ of size p , the number of classes K , and a vector of class labels $y=(y_1,\dots,y_n)$, where $y_i\in\{0, 1, \dots, K-1\}$, the problem is to build a multi-class classifier using a two-class (binary) classifier, such as a two-class SVM.

Training Stage

The model is trained with the One-Against-One method that uses the binary classification described in [Hsu02] as follows (for more references, see the Bibliography in [Hsu02]):

- For each pair of classes (i, j) , train a binary classifier, such as SVM. The total number of such binary classifiers is $K(K-1)/2$.
- If the binary classifier predicts the feature vector to be in i -th class, the number of votes for the class i is increased by one, otherwise the vote is given to the j -th class. If two classes have equal numbers of votes, the class with the smallest index is selected.

Prediction Stage

Given a new feature vector x_i , the classifier determines the class to which the vector belongs according to the algorithm 2 for computation of the class probabilities described in [Wu04]. The library returns the index of the class with the largest probability.

Batch Processing

Multi-class classifier follows the general workflow described in [Usage Model: Training and Prediction](#).

Training

For a description of the input and output, refer to [Usage Model: Training and Prediction](#).

At the training stage, a multi-class classifier has the following parameters:

Parameter	Default Value	Description
<i>algorithmFPTType</i>	double	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<i>method</i>	defaultDense	The computation method used by the multi-class classifier. The only training method supported so far is One-Against-One.
<i>training</i>	Pointer to an object of the SVM training class	Pointer to the training algorithm of the two-class classifier. By default, the SVM two-class classifier is used.
<i>prediction</i>	Pointer to an object of the SVM prediction class	Pointer to the prediction algorithm of the two-class classifier. By default, the SVM two-class classifier is used.
<i>nClasses</i>	Not applicable	The number of classes, a required parameter.

Prediction

For a description of the input and output, refer to [Usage Model: Training and Prediction](#).

At the prediction stage, a multi-class classifier has the following parameters:

Parameter	Default Value	Description
<i>algorithmFPTType</i>	double	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<i>pmethod</i>	defaultDense	The computation method used by the multi-class classifier. The only prediction method supported so far is the multi-class classifier proposed by Ting-Fan Wu et al.
<i>tmethod</i>	training::oneAgainstOne	The computation method that was used to train the multi-class classifier model.
<i>prediction</i>	Pointer to an object of the SVM prediction class	Pointer to the prediction algorithm of the two-class classifier. By default, the SVM two-class classifier is used.
<i>nClasses</i>	Not applicable	The number of classes, a required parameter.
<i>maxIterations</i>	100	The maximal number of iterations for the algorithm.
<i>accuracyThreshold</i>	1.0e-12	The prediction accuracy.

Examples

Refer to the following examples in your Intel DAAL directory:

C++:

- `./examples/cpp/source/svm/svm_multi_class_dense_batch.cpp`
- `./examples/cpp/source/svm/svm_multi_class_csr_batch.cpp`
- `./examples/cpp/source/quality_metrics/svm_multi_class_quality_metric_set_batch.cpp`

Java*:

- `./examples/java/source/com/intel/daal/examples/svm/SVMMultiClassDenseBatch.java`
- `./examples/java/source/com/intel/daal/examples/svm/SVMMultiClassCSRBatch.java`
- `./examples/java/source/com/intel/daal/examples/quality_metrics/SVMMultiClassQualityMetricSetBatchExample.java`

Python*:

- `./examples/python/source/svm/svm_multi_class_dense_batch.py`
- `./examples/python/source/svm/svm_multi_class_csr_batch.py`
- `./examples/python/source/quality_metrics/svm_multi_class_quality_metric_set_batch.py`

k-Nearest Neighbors (kNN) Classifier

k-Nearest Neighbors (kNN) classification is a non-parametric classification algorithm. The model of the kNN classifier is based on feature vectors and class labels from the training data set. This classifier induces the class of the query vector from the labels of the feature vectors in the training data set to which the query vector is similar. A similarity between feature vectors is determined by the type of distance (for example, Euclidian) in a multidimensional feature space.

Details

The library provides kNN classification based on multidimensional binary search tree (K-D tree, where D means the dimension and K means the number of dimensions in the feature space). For more details, see [James2013, Patwary2016].

Given n feature vectors $x_1 = (x_{11}, \dots, x_{1p})$, ..., $x_n = (x_{n1}, \dots, x_{np})$ of size p and a vector of class labels $y = (y_1, \dots, y_n)$, where $y_i \in \{0, 1, \dots, C - 1\}$ and C is the number of classes, describes the class to which the feature vector x_i belongs, the problem is to build a kNN classifier.

Given a positive integer parameter k and a test observation x_0 , the kNN classifier does the following:

1. Identifies the set N_0 of the k feature vectors in the training data that are closest to x_0 according to the Euclidian distance
2. Estimates the conditional probability for the class j as the fraction of vectors in N_0 whose labels y are equal to j
3. Assigns the class with the largest probability to the test observation x_0

Intel DAAL version of the kNN algorithm uses the PANDA algorithm [Patwary2016] that uses a space-partitioning data structure known as K-D tree. Each non-leaf node of a tree contains the identifier of a feature along which to split the feature space and an appropriate feature value (a cut-point) that defines the splitting hyperplane to partition the feature space into two parts. Each leaf node of the tree has an associated subset (a bucket) of elements of the training data set. Feature vectors from any bucket belong to the region of the space defined by tree nodes on the path from the root node to the respective leaf.

Training Stage

For each non-leaf node, the process of building a K-D tree involves the choice of the feature (that is, dimension in the feature space) and the value for this feature (a cut-point) to split the feature space. This procedure starts with the entire feature space for the root node of the tree, and for every next level of the tree deals with ever smaller part of the feature space.

The PANDA algorithm constructs the K-D tree by choosing the dimension with the maximum variance for splitting [Patwary2016]. Therefore, for each new non-leaf node of the tree, the algorithm computes the variance of values that belong to the respective region of the space for each of the features and chooses the feature with the largest variance. Due to high computational cost of this operation, PANDA uses a subset of feature values to compute the variance.

PANDA uses a sampling heuristic to estimate the data distribution for the chosen feature and chooses the median estimate as the cut-point.

PANDA generates new K-D tree levels until the number of feature vectors in a leaf node gets less or equal to a predefined threshold. Once the threshold is reached, PANDA stops growing the tree and associates the feature vectors with the bucket of the respective leaf node.

Prediction Stage

Given kNN classifier and query vectors x_0, \dots, x_r , the problem is to calculate the labels for those vectors. To solve the problem for each given query vector x_i , the algorithm traverses the K-D tree to find feature vectors associated with a leaf node that are closest to x_i . During the search, the algorithm limits exploration of the nodes for which the distance between the query vector and respective part of the feature space is not less than the distance from the k^{th} neighbor. This distance is progressively updated during the tree traverse.

Batch Processing

kNN classification follows the general workflow described in [Usage Model: Training and Prediction](#).

Training

For a description of the input and output, refer to [Usage Model: Training and Prediction](#).

At the training stage, K-D tree based kNN classifier has the following parameters:

Parameter	Default Value	Description
<i>algorithmFPType</i>	double	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<i>method</i>	defaultDense	The computation method used by the K-D tree based kNN classification. The only training method supported so far is the default dense method.
<i>seed</i>	777	The seed for random number generators, which are used internally to perform sampling needed to choose dimensions and cut-points for the K-D tree.
<i>dataUseInModel</i>	doNotUse	A parameter to enable/disable use of the input data set in the kNN model. Possible values: <ul style="list-style-type: none"> <code>doNotUse</code> - the algorithm does not include the input data and labels in the trained kNN model but creates a copy of the input data set. <code>doUse</code> - the algorithm includes the input data and labels in the trained kNN model.

Parameter	Default Value	Description
		The algorithm reorders feature vectors and corresponding labels in the input data set or its copy to improve performance at the prediction stage.
		If the value is <code>doUse</code> , do not deallocate the memory for input data and labels.

Prediction

For a description of the input and output, refer to [Usage Model: Training and Prediction](#).

At the prediction stage, K-D tree based kNN classifier has the following parameters:

Parameter	Default Value	Description
<code>algorithmFPType</code>	<code>double</code>	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<code>method</code>	<code>defaultDense</code>	The computation method used by the K-D tree based kNN classification. The only prediction method supported so far is the default dense method.
<code>k</code>	<code>1</code>	The number of neighbors.

Examples

Refer to the following examples in your Intel DAAL directory:

C++: `./examples/cpp/source/k_nearest_neighbors/kdtree_knn_dense_batch.cpp`

Java*: `./examples/java/source/com/intel/daal/examples/k_nearest_neighbors/KDTreeKNN.DenseBatch.java`

Python*: `./examples/python/source/k_nearest_neighbors/kdtree_knn_dense_batch.py`

Recommendation Systems

Recommendation systems track users' behavior, such as purchase habits, in order to model users' preferences and give recommendations based on their choices, actions, and reviews. Recommendation systems use different approaches, such as collaborative filtering or content-based filtering.

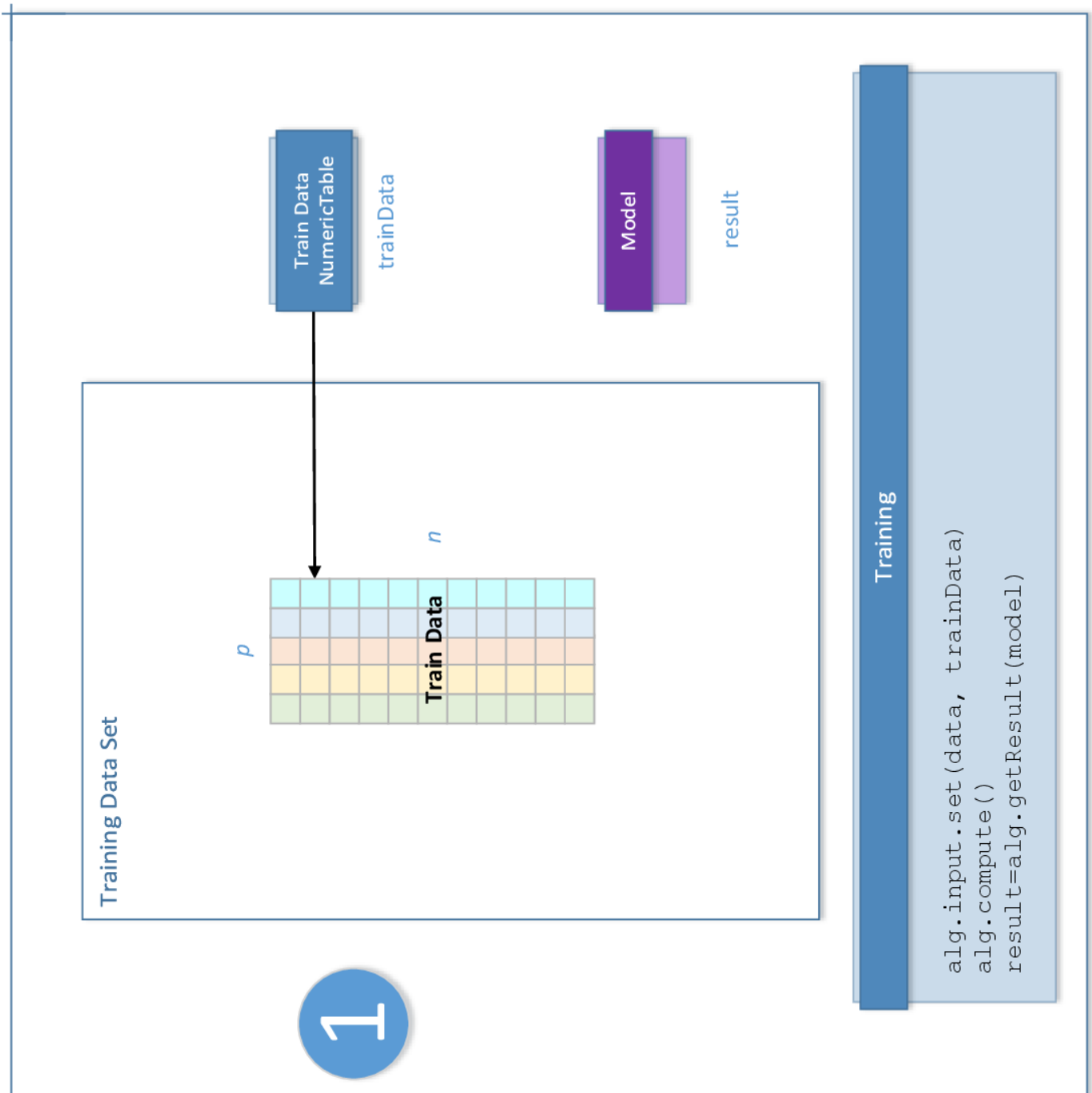
Usage Model: Training and Prediction

A typical workflow for methods of recommendation systems includes training and prediction, as explained below.

Algorithm-Specific Parameters

The parameters used by recommender algorithms at each stage depend on a specific algorithm. For a list of these parameters, refer to the description of an appropriate recommender algorithm.

Training Stage



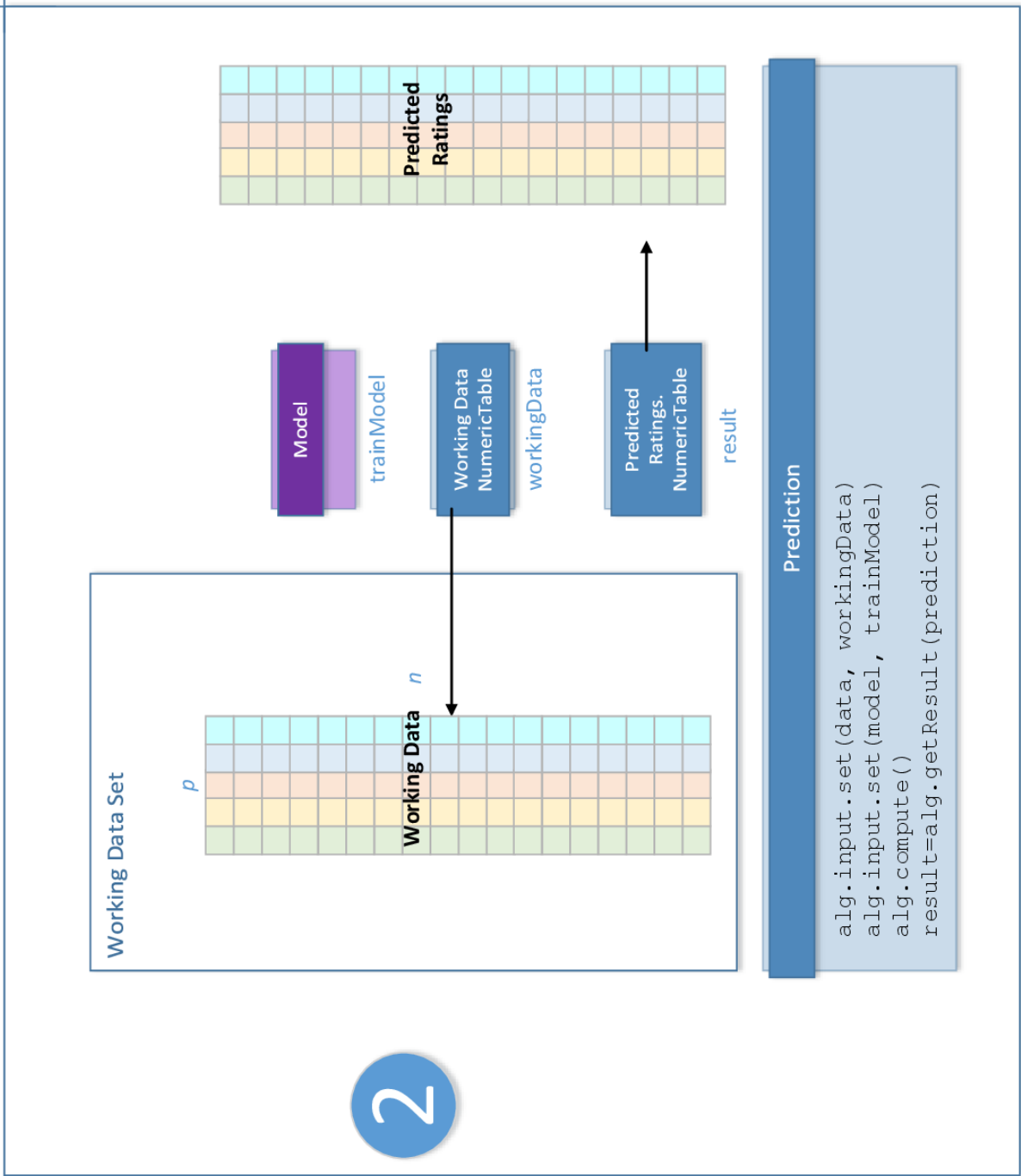
At the training stage, recommender algorithms accept the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
data	Pointer to the $m \times n$ numeric table with the mining data. This table can be an object of any class derived from <code>NumericTable</code> except <code>PackedTriangularMatrix</code> and <code>PackedSymmetricMatrix</code> .

At the training stage, recommender algorithms calculate the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
model	Model with initialized item factors. The result can only be an object of the <code>Model</code> class.

Prediction Stage



At the prediction stage, recommender algorithms accept the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
model	Model with initialized item factors. This input can only be an object of the <code>Model</code> class.

At the prediction stage, recommender algorithms calculate the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
prediction	Pointer to the $m \times n$ numeric table with predicted ratings. By default, this table is an object of the <code>HomogenNumericTable</code> class, but you can define it as an object of any class derived from <code>NumericTable</code> except <code>PackedSymmetricMatrix</code> , <code>PackedTriangularMatrix</code> , and <code>CSRNumericTable</code> .

Implicit Alternating Least Squares

The library provides the Implicit Alternating Least Squares (implicit ALS) algorithm [Fleischer2008], based on collaborative filtering.

Details

Training Stage

Given the matrix of observations R of size $m \times n$, where n is the number of users and m is the number of items, the problem is to find the matrix X of size $m \times f$ and the matrix Y of size $f \times n$, where f is the number of factors that minimize the following cost function:

$$\min_{x_*, y_*} \sum_{u, i} c_{ui} (p_{ui} - x_u^T y_i)^2 + \lambda (\sum_u n_x \|x_u\|^2 + \sum_i m_y \|y_i\|^2),$$

$$p_{ui} = \begin{cases} 1, & r_{ui} > 0 \\ 0, & r_{ui} = 0 \end{cases}, c_{ui} = 1 + \alpha r_{ui},$$

where

- c_{ui} measures the confidence in observing p_{ui}
- α is the rate of confidence
- r_{ui} is the element of the matrix R
- λ is the parameter of the regularization
- n_x, m_y denote the number of ratings of user u and item i respectively

Prediction Stage

Prediction of Ratings

Given user factors X , item factors Y , and the matrix D that describes for which pairs of factors X and Y the rating should be computed, the system calculates the matrix of recommended ratings Res :

$$res_{ui} = \sum_{j=1}^f x_{uj} y_{ji} \text{ if } d_{ui} \neq 0, u = 1, \dots, m, i = 1, \dots, n.$$

Initialization

Batch Processing

Input

Initialization of item factors for the implicit ALS algorithm accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
<code>data</code>	Pointer to the $m \times n$ numeric table with the mining data. The input can be an object of any class derived from <code>NumericTable</code> except <code>PackedTriangularMatrix</code> and <code>PackedSymmetricMatrix</code> .

Parameters

Initialization of item factors for the implicit ALS algorithm has the following parameters:

Parameter	Default Value	Description
<code>algorithmFPType</code>	<code>double</code>	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<code>method</code>	<code>defaultDense</code>	Available computation methods: <ul style="list-style-type: none"> <code>defaultDense</code> - performance-oriented method <code>fastCSR</code> - performance-oriented method for CSR numeric tables
<code>nFactors</code>	10	The total number of factors.
<code>seed</code>	777777	The seed for the random number generation in the initialization step.

Output

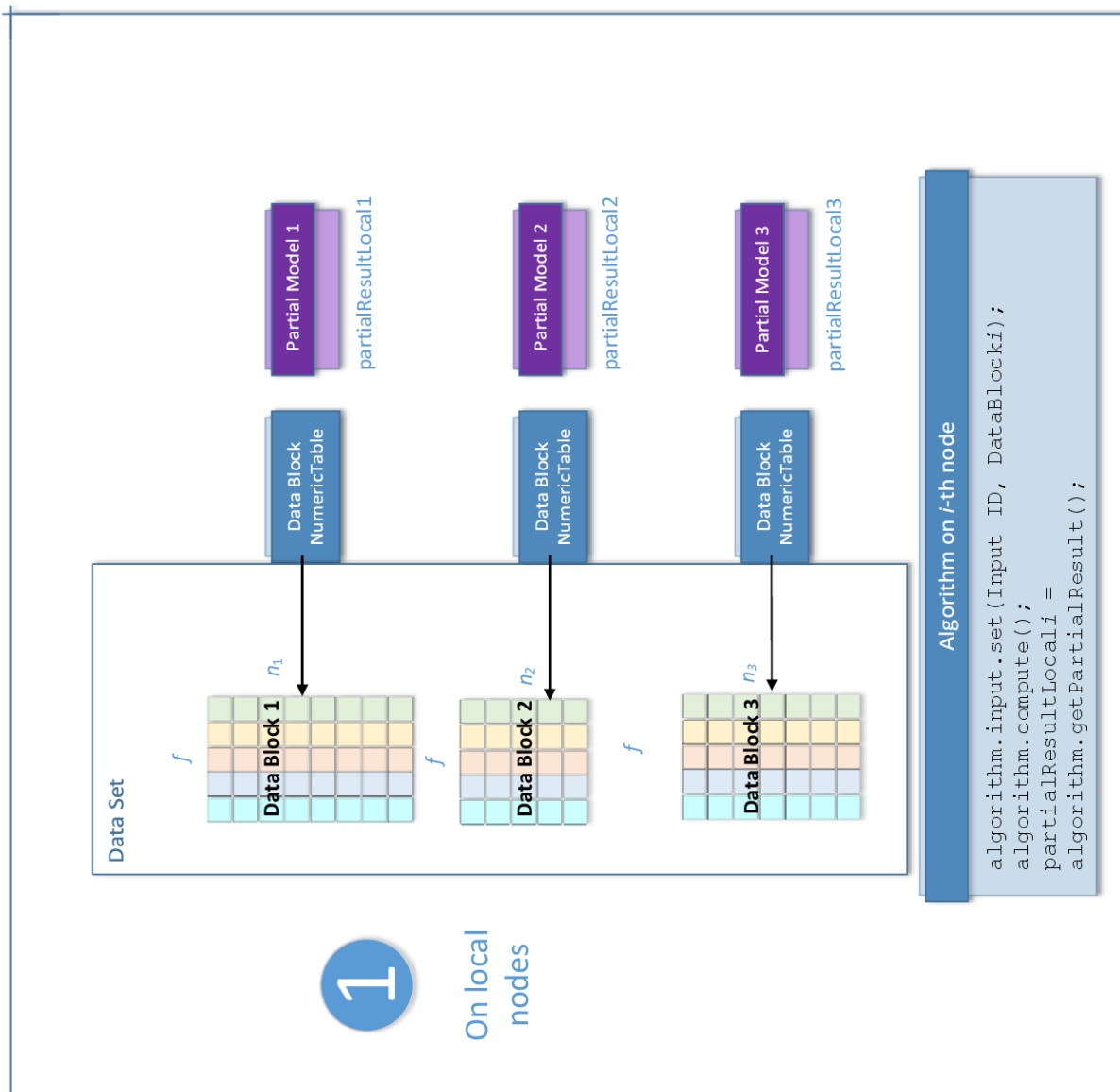
Initialization of item factors for the implicit ALS algorithm calculates the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
<code>model</code>	The model with initialized item factors. The result can only be an object of the <code>Model</code> class.

Distributed Processing

The distributed processing mode assumes that the data set is split in `nblocks` blocks across computation nodes.

To initialize the implicit ALS algorithm in the distributed processing mode, use the one-step process illustrated by the following diagram for `nblocks=3`:



Input

In the distributed processing mode, initialization of item factors for the implicit ALS algorithm accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
data	Pointer to the $m \times n$ numeric table with the mining data. The input can be an object of any class derived from <code>NumericTable</code> except <code>PackedTriangularMatrix</code> and <code>PackedSymmetricMatrix</code> .

Parameters

In the distributed processing mode, initialization of item factors for the implicit ALS algorithm has the following parameters:

Parameter	Default Value	Description
<i>algorithmFPTType</i>	double	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<i>method</i>	fastCSR	Performance-oriented computation method for CSR numeric tables, the only method supported by the algorithm.
<i>nFactors</i>	10	The total number of factors.
<i>seed</i>	777777	The seed for the random number generation in the initialization step.
<i>fullNUsers</i>	0	The total number of users.

Output

In the distributed processing mode, initialization of item factors for the implicit ALS algorithm calculates the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
model	The model with initialized item factors. The result can only be an object of the <code>Model</code> class.

Computation

Batch Processing

Training

For a description of the input and output, refer to [Usage Model: Training and Prediction](#).

At the training stage, the implicit ALS recommender has the following parameters:

Parameter	Default Value	Description
<i>algorithmFPTType</i>	double	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<i>method</i>	defaultDense	Available computation methods: <ul style="list-style-type: none"> <code>defaultDense</code> - performance-oriented method <code>fastCSR</code> - performance-oriented method for CSR numeric tables
<i>nFactors</i>	10	The total number of factors.
<i>maxIterations</i>	5	The number of iterations.
<i>alpha</i>	40	The rate of confidence.
<i>lambda</i>	0.01	The parameter of the regularization.

Parameter	Default Value	Description
<code>preferenceThreshold</code>	0	Threshold used to define preference values. 0 is the only threshold supported so far.

Prediction

For a description of the input and output, refer to [Usage Model: Training and Prediction](#).

At the prediction stage, the implicit ALS recommender has the following parameters:

Parameter	Default Value	Description
<code>algorithmFPType</code>	double	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<code>method</code>	defaultDense	Performance-oriented computation method, the only method supported by the algorithm.

Examples

Refer to the following examples in your Intel DAAL directory:

C++:

- `./examples/cpp/source/implicit_als/implicit_als_dense_batch.cpp`
- `./examples/cpp/source/implicit_als/implicit_als_csr_batch.cpp`

Java*:

- `./examples/java/source/com/intel/daal/examples/implicit_als/ImplicitAlsDenseBatch.java`
- `./examples/java/source/com/intel/daal/examples/implicit_als/ImplicitAlsCSRBatch.java`

Python*:

- `./examples/python/source/implicit_als/implicit_als_dense_batch.py`
- `./examples/python/source/implicit_als/implicit_als_csr_batch.py`

Distributed Processing

Training

The distributed processing mode assumes that the data set is split in *nblocks* blocks across computation nodes.

Algorithm Parameters

At the training stage, implicit ALS recommender in the distributed processing mode has the following parameters:

Parameter	Default Value	Description
<code>computeStep</code>	Not applicable	The parameter required to initialize the algorithm. Can be: <ul style="list-style-type: none"> • <code>step1Local</code> - the first step, performed on local nodes

Parameter	Default Value	Description
		<ul style="list-style-type: none"> <code>step2Master</code> - the second step, performed on a master node <code>step3Local</code> - the third step, performed on local nodes <code>step4Local</code> - the fourth step, performed on local nodes
<code>algorithmFPType</code>	double	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<code>method</code>	fastCSR	Performance-oriented computation method for CSR numeric tables, the only method supported by the algorithm.
<code>nFactors</code>	10	The total number of factors.
<code>maxIterations</code>	5	The number of iterations.
<code>alpha</code>	40	The rate of confidence.
<code>lambda</code>	0.01	The parameter of the regularization.
<code>preferenceThreshold</code>	0	Threshold used to define preference values. 0 is the only threshold supported so far.

Computation Process

At each iteration, the implicit ALS training algorithm alternates between re-computing user factors (X) and item factors (Y). These computations split each iteration into the following parts:

1. Re-compute all user factors using the input data sets and item factors computed previously.
2. Re-compute all item factors using input data sets in the transposed format and item factors computed previously.

Important

In Intel DAAL, computation of the implicit ALS in the distributed processing mode requires that you provide the matrices X and Y and their transposed counterparts.

Each part includes four steps executed either on local nodes or on the master node, as explained below and illustrated by graphics for `nblocks=3`. The main loop of the implicit ALS training stage is executed on the master node. The following pseudocode illustrates the entire computation:

```
for (iteration = 0; iteration < maxIterations; iteration++)
{
    /* Update partial user factors */
    computeStep1Local(itemsPartialResultLocal);
    computeStep2Master();
    computeStep3Local(itemsOffsetTable, itemsPartialResultLocal, itemsFactorsToNodes);
    usersPartialResultLocal = computeStep4Local(itemsPartialResultLocal);

    /* Update partial item factors */
    computeStep1Local(usersPartialResultLocal);
    computeStep2Master();
}
```

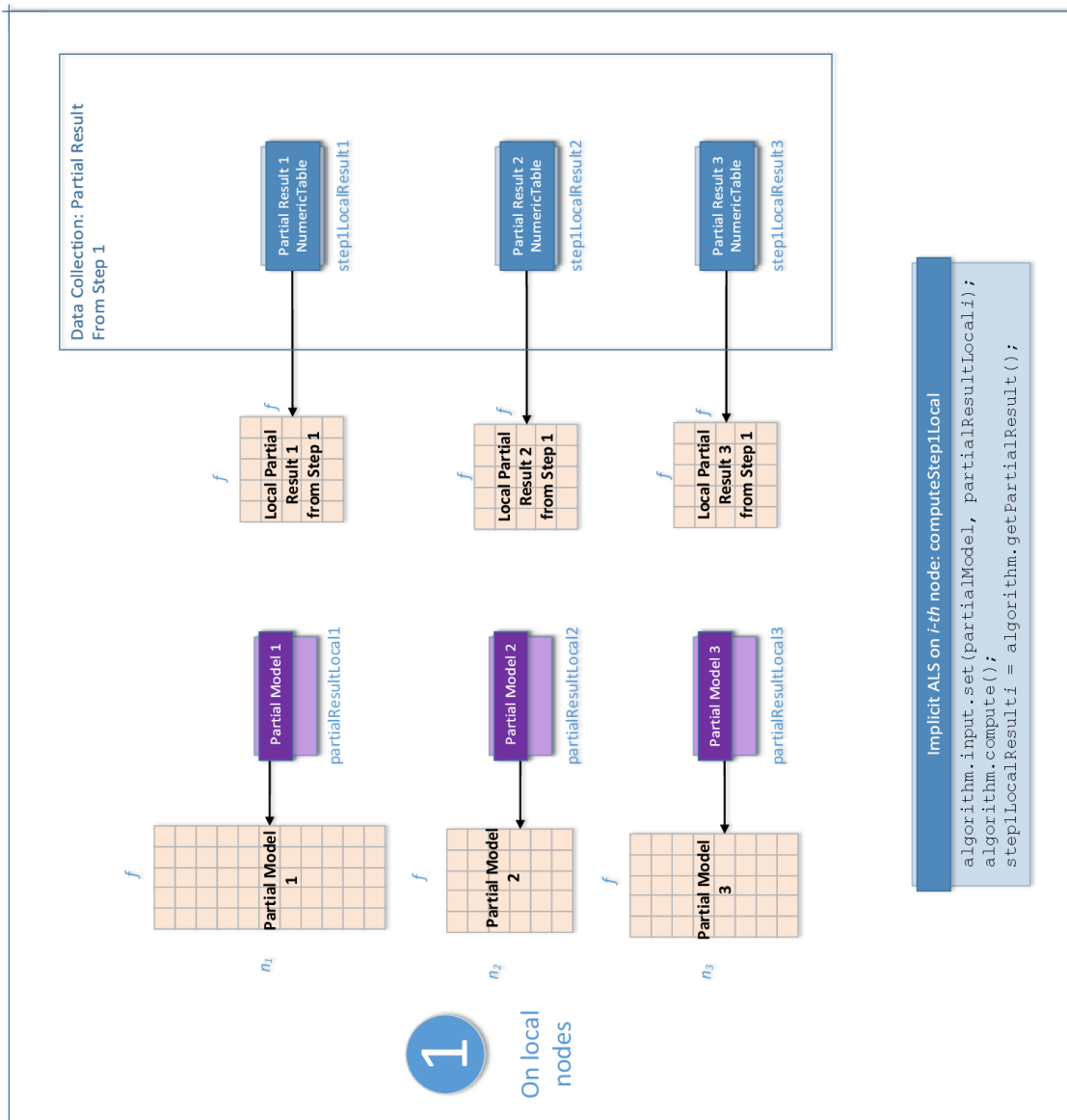
```
computeStep3Local(usersOffsetTable, usersPartialResultLocal, usersFactorsToNodes);
itemsPartialResultLocal = computeStep4Local(usersPartialResultLocal);
}
```

Step 1 - on Local Nodes

This step works with the matrix:

- Y in part 1 of the iteration
- X in part 2 of the iteration

Parts of this matrix are used as input partial models.



In this step, implicit ALS recommender training accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

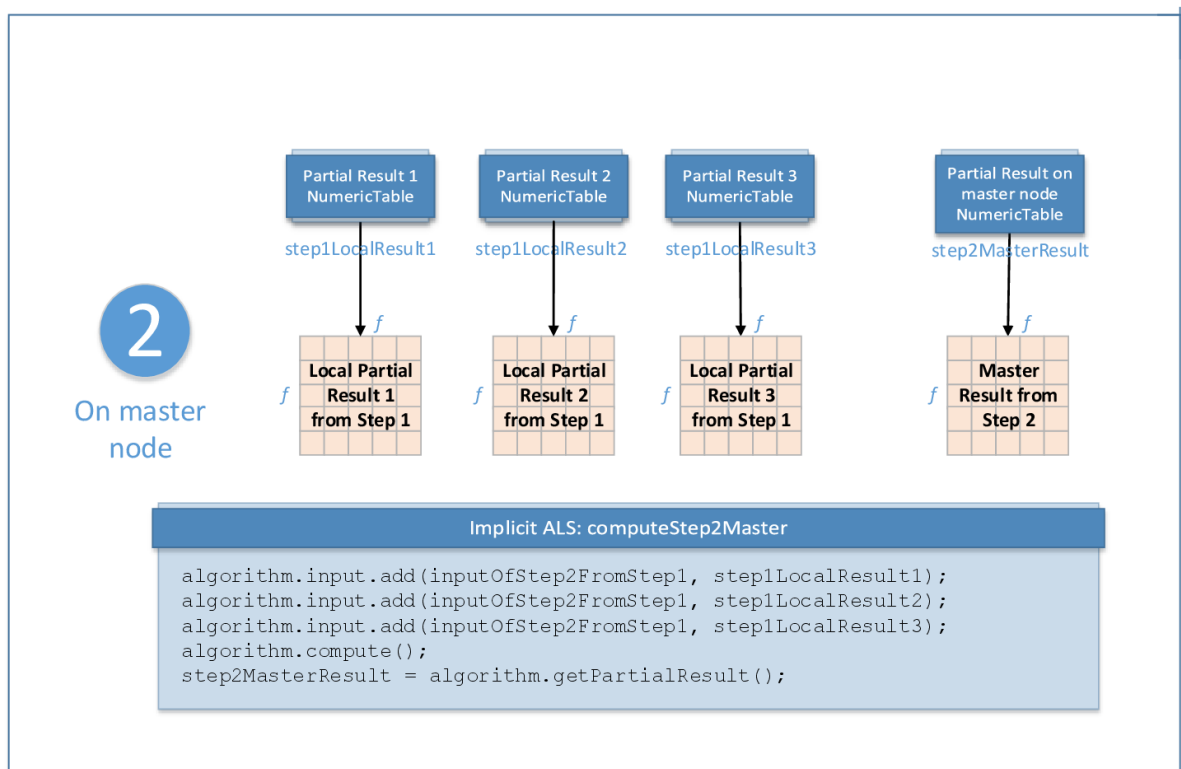
Input ID	Input
<code>partialModel</code>	Partial model computed on the local node.

In this step, implicit ALS recommender training calculates the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
<code>outputOfStep1ForStep2</code>	Pointer to the $f \times f$ numeric table with the sum of numeric tables calculated in Step 1.

Step 2 - on Master Node

This step uses local partial results from [Step 1](#) as input.



In this step, implicit ALS recommender training accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
<code>inputOfStep2FromStep1</code>	A collection of numeric tables computed on local nodes in Step 1 . The collection may contain objects of any class derived from <code>NumericTable</code> except the <code>PackedTriangularMatrix</code> class with the <code>lowerPackedTriangularMatrix</code> layout.

In this step, implicit ALS recommender training calculates the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
outputOfStep2ForStep4	Pointer to the $f \times f$ numeric table with merged cross-products.

Step 3 - on Local Nodes

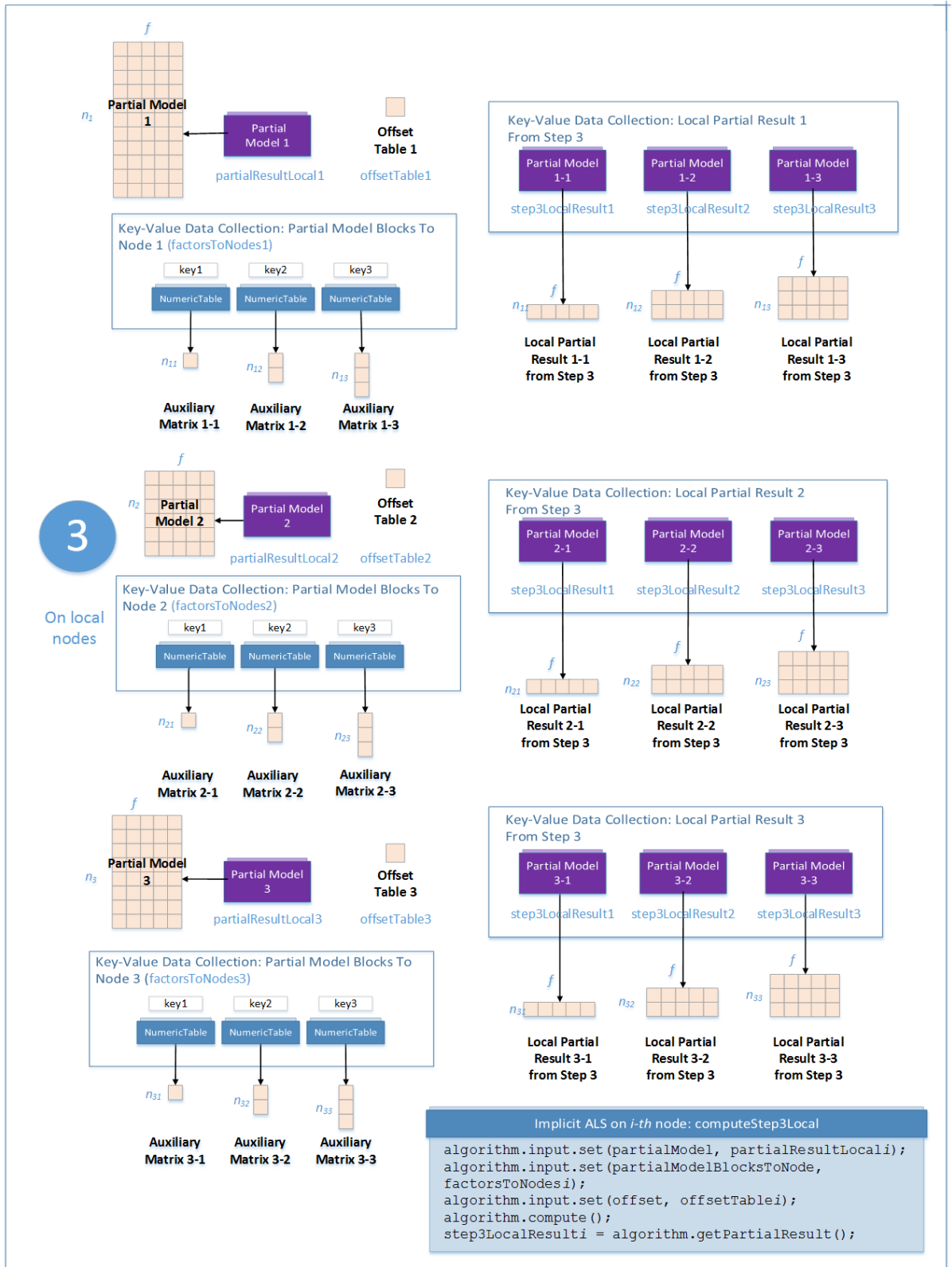
On each node i , this step uses results of the previous steps and requires that you provide two extra matrices **Offset Table i** and **Partial Model Blocks To Node i** .

The only element of the **Offset Table i** table refers to the matrix:

- Y in part 1 of the iteration
- X in part 2 of the iteration

The **Offset Table i** table must hold the index in the matrix row of the first element in the partial model obtained in [Step 1](#). Fill this table with the index of the starting row of the part of the input data set located on the i -th node.

The **Partial Model Blocks To Node i** table contains a key-value data collection that specifies the node where a subset of the partial model must to be transferred. For example, if $key[i]$ (such as *key1* in the diagram) contains indices $\{j1, j2, j3\}$, rows $j1, j2$, and $j3$ of the partial model must be transferred to the computation node i .



In this step, implicit ALS recommender training accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
<code>partialModel</code>	Partial model computed on the local node.
<code>partialModelBlocksToNode</code>	A key-value data collection that maps components of the partial model to local nodes: i -th element of this collection is a numeric table that contains indices of the factors to be transferred to the i -th node.
<code>offset</code>	Pointer to 1x1 numeric table that holds the global index of the starting row of the input partial model.

In this step, implicit ALS recommender training calculates the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

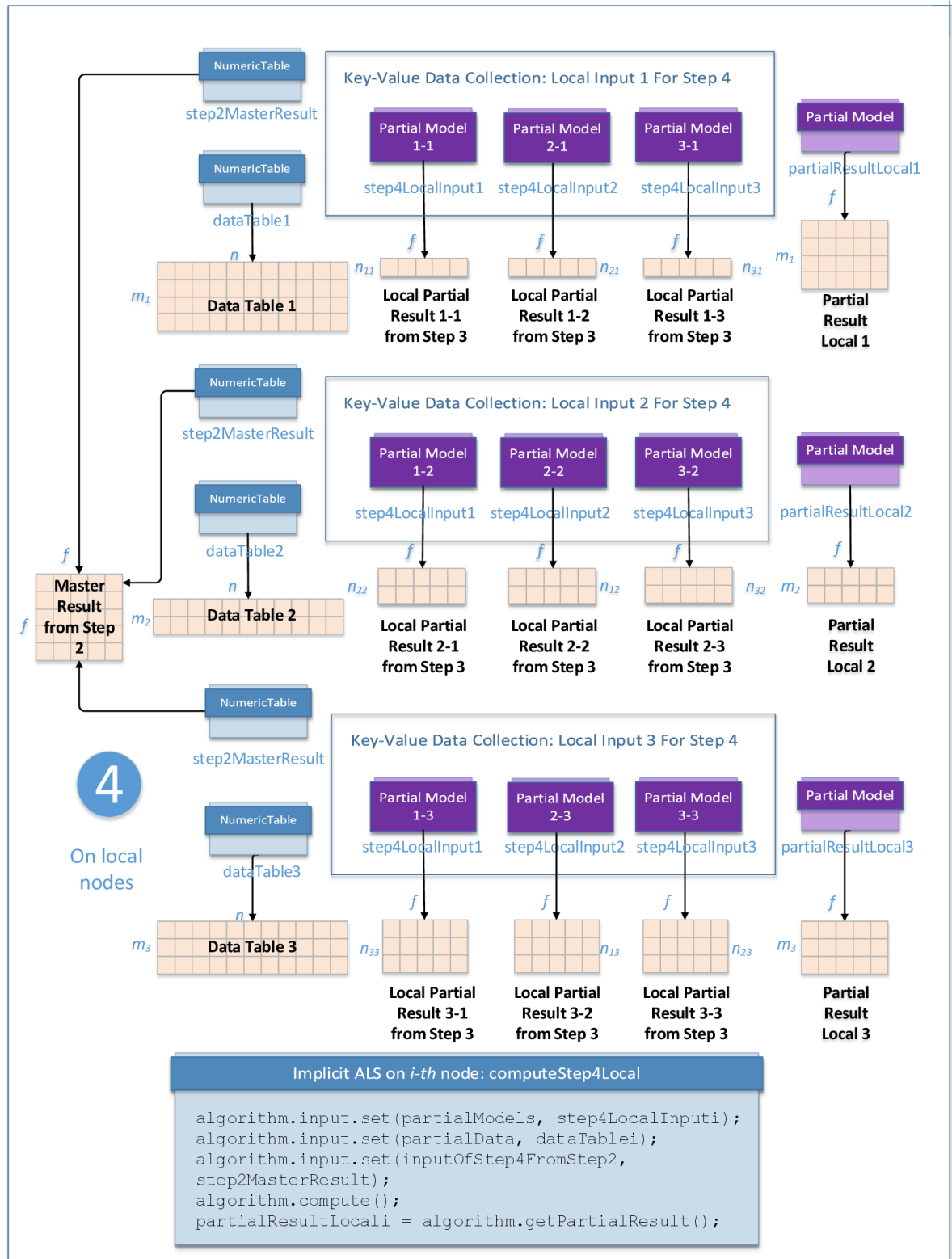
Result ID	Result
<code>outputOfStep3ForStep4</code>	A key-value data collection that contains partial models to be used in Step 4 . Each element of the collection contains an object of the <code>PartialModel</code> class.

Step4 - on Local Nodes

This step uses the results of the previous steps and parts of the following matrix in the transposed format:

- X in part [1](#) of the iteration
- Y in part [2](#) of the iteration

The results of the step are the re-computed parts of this matrix.



In this step, implicit ALS recommender training accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
<code>partialModels</code>	A key-value data collection with partial models that contain user factors/item factors computed in Step 3 . Each element of the collection contains an object of the <code>PartialModel</code> class.
<code>partialData</code>	Pointer to the CSR numeric table that holds the i -th part of ratings data, assuming that the data is divided by users/items.
<code>inputOfStep4FromStep2</code>	Pointer to the $f \times f$ numeric table computed in Step 2 .

In this step, implicit ALS recommender training calculates the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
<code>outputOfStep4ForStep1</code>	Pointer to the partial implicit ALS model that corresponds to the i -th data block. The partial model stores user factors/item factors.
<code>outputOfStep4ForStep3</code>	

Examples

Refer to the following examples in your Intel DAAL directory:

C++: `./examples/cpp/source/implicit_als/implicit_als_csr_distributed.cpp`

Java*: `./examples/java/source/com/intel/daal/examples/implicit_als/ImplicitAlsCSR Distributed.java`

Python*: `./examples/python/source/implicit_als/implicit_als_csr_distributed.py`

Prediction of Ratings

The distributed processing mode assumes that the data set is split in $nblocks$ blocks across computation nodes.

Algorithm Parameters

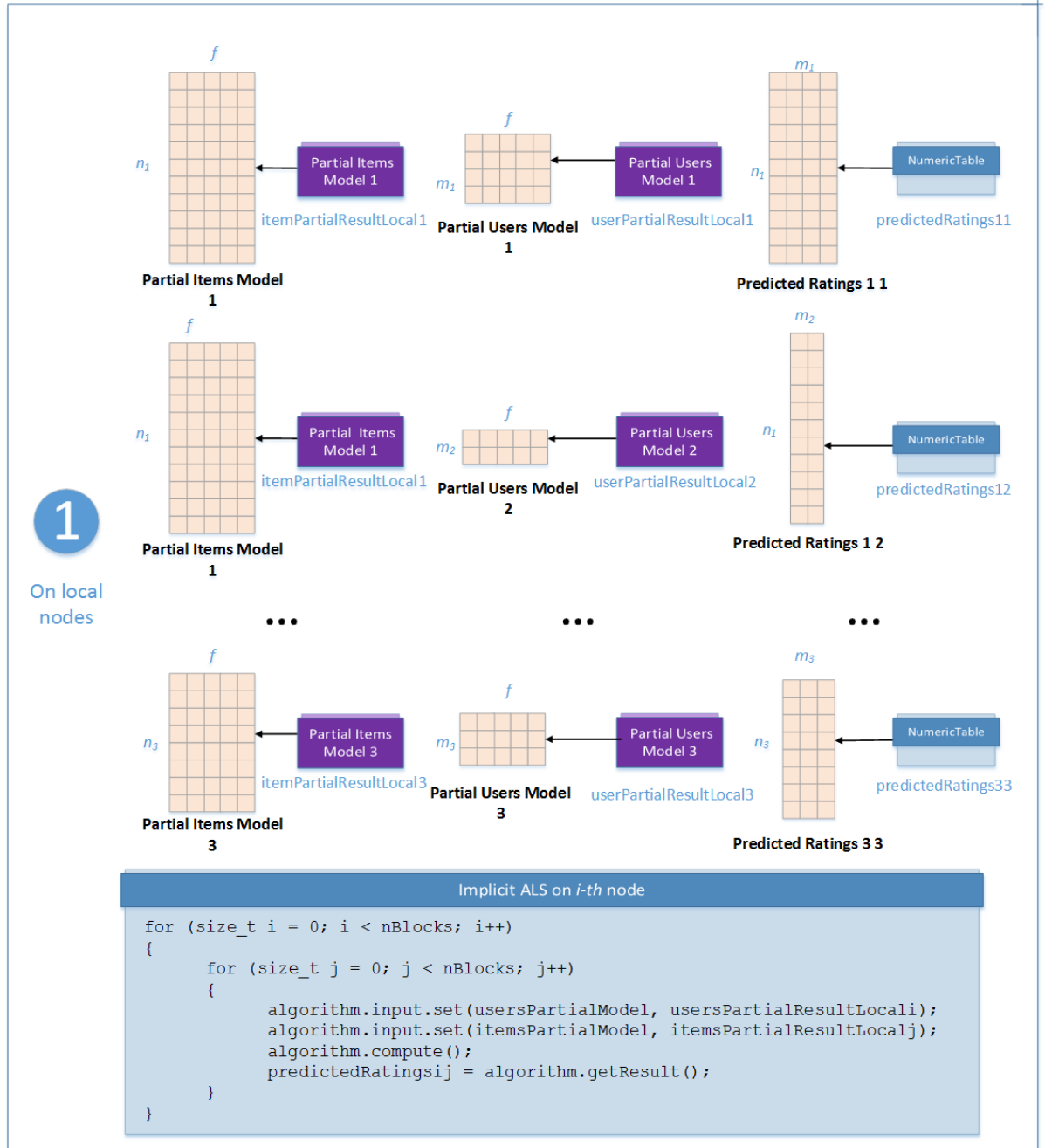
At the prediction stage, implicit ALS recommender in the distributed processing mode has the following parameters:

Parameter	Default Value	Description
<code>computeStep</code>	Not applicable	The parameter required to initialize the algorithm. Can be: <ul style="list-style-type: none"> <code>step1Local</code> - the first step, performed on local nodes
<code>algorithmFPType</code>	<code>double</code>	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<code>method</code>	<code>defaultDense</code>	Performance-oriented computation method, the only method supported by the algorithm.
<code>nFactors</code>	10	The total number of factors.

Use the one-step computation schema for implicit ALS recommender prediction in the distributed processing mode, as explained below and illustrated by the graphic for $nblocks=3$:

Step 1 - on Local Nodes

Prediction of rating uses partial models, which contain the parts of user factors $X_1, X_2, \dots, X_{nblocks}$ and item factors $Y_1, Y_2, \dots, Y_{nblocks}$ produced at the [training stage](#). Each pair of partial models (X_i, Y_j) is used to compute a numeric table with ratings R_{ij} that correspond to the user factors and item factors from the input partial models.



In this step, implicit ALS recommender-based prediction accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
<code>usersPartialModel</code>	The partial model trained by the implicit ALS algorithm in the distributed processing mode. Stores user factors that correspond to the i -th data block.
<code>itemsPartialModel</code>	The partial model trained by the implicit ALS algorithm in the distributed processing mode. Stores item factors that correspond to the j -th data block.

In this step, implicit ALS recommender-based prediction calculates the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
<code>prediction</code>	Pointer to the $m_i \times n_j$ numeric table with predicted ratings. By default this table is an object of the <code>HomogenNumericTable</code> class, but you can define it as an object of any class derived from <code>NumericTable</code> except <code>PackedTriangularMatrix</code> , <code>PackedSymmetricMatrix</code> , and <code>CSRNumericTable</code> .

Examples

Refer to the following examples in your Intel DAAL directory:

C++: `./examples/cpp/source/implicit_als/implicit_als_csr_distributed.cpp`

Java*: `./examples/java/source/com/intel/daal/examples/implicit_als/ImplicitAlsCSRdistributed.java`

Python*: `./examples/python/source/implicit_als/implicit_als_csr_distributed.py`

Performance Considerations

To get the best overall performance of the implicit ALS recommender:

- If input data is homogeneous, provide the input data and store results in homogeneous numeric tables of the same type as specified in the `algorithmFPTType` class template parameter.
- If input data is sparse, use CSR numeric tables.

Optimization Notice

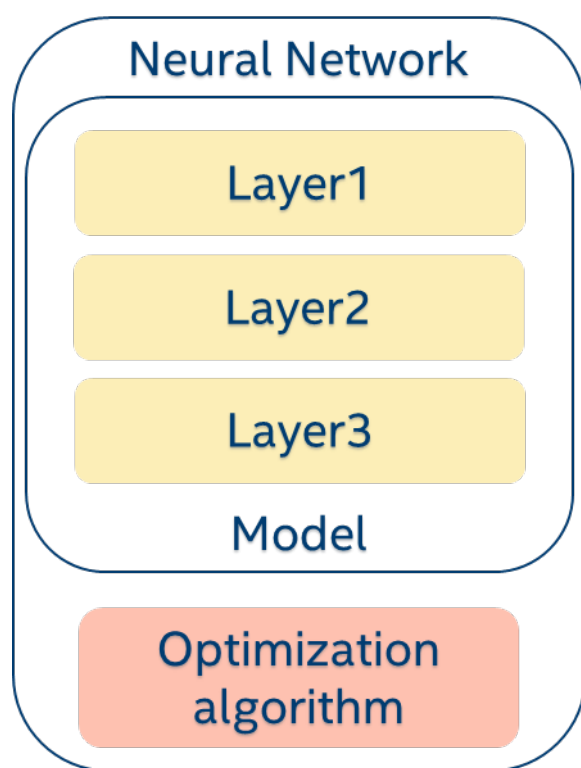
Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Neural Networks

Artificial neural networks are mathematical models inspired by their biological counterparts. A neural network is an information processing system used to approximate functions of large numbers of arguments. Neural networks are applied to solve different kinds of machine learning tasks, such as image and video processing [Szegedy13], classification, text recognition or natural language understanding, and others [LeCun15]. At a high level, in Intel DAAL a neural network comprises:

- Interconnected computation devices, or *layers* of neurons.
- The *topology*, which defines the logical structure of the network. The topology defines layers with their parameters and describes the connections between those layers.
- The *model*, which contains information about layer structures and all the data associated with the layers and about weights and biases to be optimized.
- The *optimization algorithm*. It updates weights and biases at the training stage, and it is not used at the prediction stage.



A layer can perform forward and backward computations. For a list of available layers and descriptions of their usage, see [Layers](#). The information about the order of layers is stored in the model. Intel DAAL supports graph organization of layers and has special layers that used for that.

In addition to the order of layers, the model also contains information about weights and biases being optimized in each computation step using the optimization solver.

Intel DAAL provides different optimization solvers. For the descriptions and usages of the solvers, see [Optimization Solvers](#).

For data storage and computations, neural networks use *tensors*, multidimensional data structures. For more details, see [Tensors](#).

Flexibility of Intel DAAL allows you to create your own neural network configuration by choosing the topology, as well as number of layers and layer types. You can even create your own layers or optimization solvers by deriving from relevant classes.

Usage Model: Training and Prediction

Training

Given a $(p+1)$ -dimensional tensor q of size $n_1 \times n_2 \times \dots \times n_p \times n_{p+1}$ where each element is a sample, a $(p+1)$ -dimensional tensor y of size $n_1 \times n_2 \times \dots \times n_p \times n_{p+1}$ where each element is a stated result for the corresponding sample, and a neural network that consists of n layers, the problem is to train the neural network. For more details, see [Training and Prediction](#).

NOTE

Intel DAAL supports only supervised learning with a known vector of class labels.

The key mechanism used to train a neural network is a backward propagation of errors [\[Rumelhart86\]](#). During the training stage the algorithm performs forward and backward computations.

The training stage consists of one or several epochs. An *epoch* is the time interval when the network processes the entire input data set performing several forward passes, backward passes, and updates of weights and biases in the neural network model.

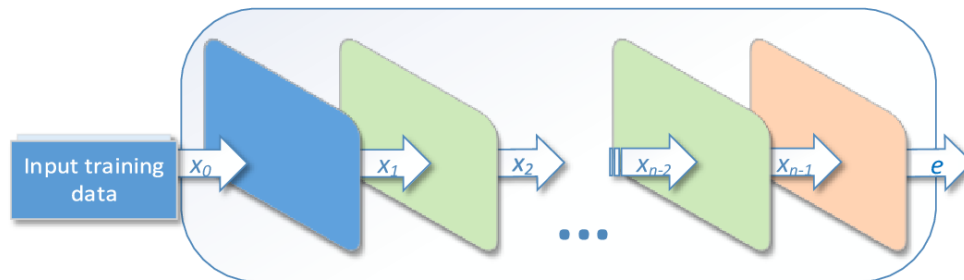
Each epoch consists of several iterations. An iteration is the time interval when the network performs one forward and one backward pass using a part of the input data set called a *batch*. At each iteration, the optimization solver performs an optimization step and updates weights and biases in the model.

Forward Computation

Follow these steps:

1. Provide the neural network with the input data for training. You can provide either one sample or a set of samples. The *batchSize* parameter specifies the number of simultaneously processed samples.
2. Compute $x_{i+1} = f(x_i)$, where:
 - x_i is the input data for the layer i
 - x_{i+1} is the output value of the layer i
 - $f_i(x)$ is the function corresponding to the layer i .
 - $i = 0, \dots, n-1$ is the index of the layer

For some layers, the computation can also use weights w and biases b . For more details, see [Layers](#).



3. Compute an error as the result of a loss layer: $e = f_{loss}(x_{n-1}, y)$. For available loss layers, see [Layers](#).

NOTE

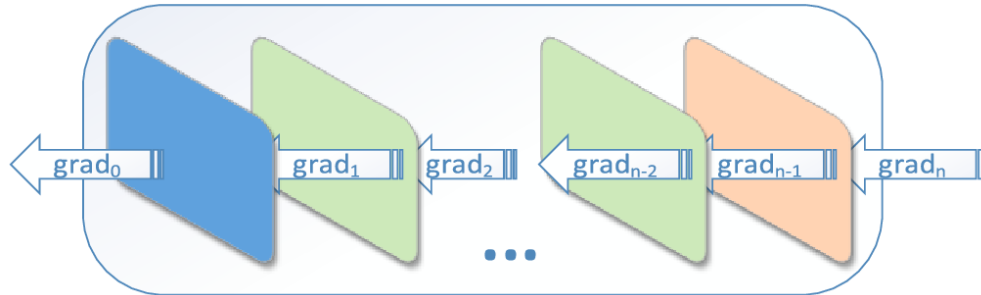
In the descriptions of specific forward layers in the [Layers](#) section, the preceding layer for the layer i is the layer $i-1$.

Backward Computation

Follow these steps:

1. Compute the input gradient for the penultimate layer as the gradient of the loss layer $grad_n = \nabla f_{loss}(x_{n-1}, y)$.
2. Compute $grad_i = \nabla f_i(x_i) * grad_{i+1}$, where:
 - $grad_i$ is the gradient obtained at the i -the layer
 - $grad_{i+1}$ is the gradient obtained at the $(i+1)$ -the layer
 - $i = n - 1, \dots, 0$

For some layers, the computation can also use weights w and biases b . For more details, see [Layers](#).



3. Apply one of the optimization methods to the results of the previous step. Compute $w, b = \text{optimizationSolver}(w, b, grad_0)$, where $w = (w_0, w_1, \dots, w_{n-1})$, $b = (b_0, b_1, \dots, b_{n-1})$. For available optimization solver algorithms, see [Optimization Solvers](#).

As a result of the training stage, you receive the trained model with the optimum set of weights and biases. Use the `getPredictionModel` method to get the model you can use at the prediction stage. This method performs the following steps to produce the prediction model from the training model:

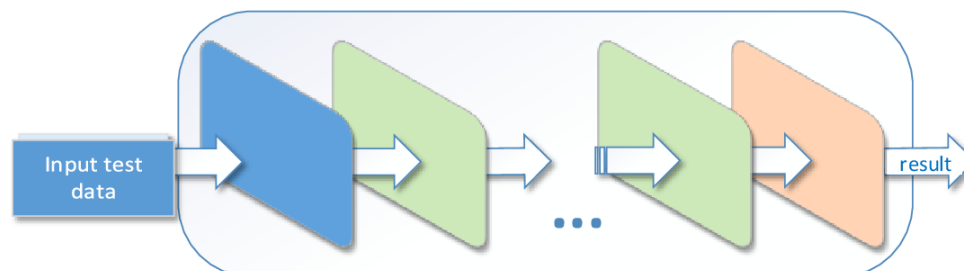
1. Clones all the forward layers of the training model except the loss layer.
2. Replaces the loss layer with the layer returned by the `getLayerForPrediction` method of the forward loss layer. For example, the loss softmax cross-entropy forward layer is replaced with the softmax forward layer.

NOTE

In the descriptions of specific backward layers in the [Layers](#) section, the preceding layer for the layer i is the layer $i+1$.

Prediction

Given the trained network (with optimum set of weights w and biases b) and a new $(p+1)$ -dimensional tensor x of size $n_1 \times n_2 \times \dots \times n_p \times n_{p+1}$, the algorithm determines the result for each sample (one of elements of the tensor y). Unlike the training stage, during prediction the algorithm performs the forward computation only.



Batch Processing

For the description of the neural network training algorithm, see [Usage Model: Training and Prediction](#).

Training

Algorithm Input

Neural network training in the batch processing mode accepts the following input. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
<code>data</code>	Pointer to the tensor of size $n_1 \times n_2 \times \dots \times n_p$ that stores the neural network input data. This input can be an object of any class derived from <code>Tensor</code> .
<code>groundTruth</code>	Pointer to the tensor of size n_1 that stores stated results associated with the input data. This input can be an object of any class derived from <code>Tensor</code> .

Algorithm Parameters

Neural network training in the batch processing mode has the following parameters:

Parameter	Default Value	Description
<code>algorithmFPType</code>	<code>float</code>	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<code>method</code>	<code>defaultDense</code>	Performance-oriented computation method.
<code>batchSize</code>	<code>1</code>	The number of samples simultaneously used for training. NOTE Because the first dimension of the input data tensor represents the data samples, the library computes the number of batches by dividing n_1 by the value of <code>batchSize</code> . After processing each batch the library updates the parameters of the model. If n_1 is not a multiple of <code>batchSize</code> , the algorithm ignores data samples at the end of the data tensor.
<code>optimization Solver</code>	<code>SharedPtr< optimization_solver::sgd::Batch< algorithmFPType, defaultDense> ></code>	The optimization procedure used at the training stage.

Algorithm Output

Neural network training in the batch processing mode calculates the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
<code>model</code>	Trained model with the optimum set of weights and biases. The result can only be an object of the <code>Model</code> class.

Examples

Refer to the following examples in your Intel DAAL directory:

C++: `./examples/cpp/source/neural_networks/neural_network_batch.cpp`

Java*: `./examples/java/source/com/intel/daal/examples/neural_networks/NeuralNetworkBatch.java`

Python*: `./examples/python/source/neural_networks/neural_net_dense_batch.py`

Prediction

Algorithm Input

Neural network prediction algorithm in the batch processing mode accepts the following input. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
<code>data</code>	Pointer to the tensor of size $n_1 \times n_2 \times \dots \times n_p$ that stores the neural network input data. This input can be an object of any class derived from <code>Tensor</code> .
<code>model</code>	Trained model with the optimum set of weights and biases. The result can only be an object of the <code>Model</code> class.

Algorithm Parameters

Neural network prediction algorithm in the batch processing mode has the following parameters:

Parameter	Default Value	Description
<code>algorithmFPType</code>	<code>float</code>	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<code>method</code>	<code>defaultDense</code>	Performance-oriented computation method.
<code>nIterations</code>	<code>1000</code>	The number of iterations.
<code>batchSize</code>	<code>1</code>	The number of samples simultaneously used for prediction.

Algorithm Output

Neural network prediction algorithm in the batch processing mode calculates the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
prediction	Pointer to the tensor of size n_1 that stores the predicted result for each sample. This input can be an object of any class derived from <code>Tensor</code> .

Examples

Refer to the following examples in your Intel DAAL directory:

C++: `./examples/cpp/source/neural_networks/neural_network_prediction_batch.cpp`

Java*: `./examples/java/source/com/intel/daal/examples/neural_networks/NeuralNetworkPredictionBatch.java`

Python*: `./examples/python/source/neural_networks/neural_net_predict_dense_batch.py`

Distributed Processing

You can use the distributed processing mode for neural network training. Approaches to neural network training in the distributed mode are based on the following kinds of parallelization:

- Data based
- Model based
- Hybrid (data+model based)

The library supports data based parallelization.

Data Based Parallelization

The data based parallelization approach has the following features:

- The training data is split across local nodes.
- Instances of the same model are used by local nodes to compute local derivatives.
- The master node updates the weights and biases parameters of the model using the local derivatives and delivers them back to the local nodes.

The library supports the following ways to update the parameters of the neural network model for data based parallelization:

- **Synchronous.**

The master node updates the model only after all local nodes deliver the local derivatives for a given iteration of the training.

- **Asynchronous.**

The master node:

- Immediately sends the latest version of the model to the local node that delivered the local derivatives.
- Updates the model as soon as the master node accumulates a sufficient amount of partial results. This amount is defined by the requirements of the application.

Computation

The flow of the neural network model training using data based parallelization involves these steps:

1. Initialize the neural network model using the `initialize()` method on the master node and propagate the model to local nodes.
2. Run the training algorithm on local nodes as described in the [Usage Model: Training and Prediction > Training](#) section with the following specifics of the distributed computation mode:
 - Provide each j -th node of the neural network with the local data set of `localDataSizej` size.

- Specify the required `batchSizej` parameter.
- Split the data set on a local node into `localDataSizej/batchSizej` data blocks, each to be processed by the local algorithm separately.
- The `batchSizej` parameters and `localDataSizej` parameters must be the same on all local nodes for synchronous computations and can be different for asynchronous computations.

See the figure below to visualize an i -th iteration, corresponding to the i -th data block. After the computations for the i -th data block on a local node are finished, send the derivatives of local weights and biases to the master node.

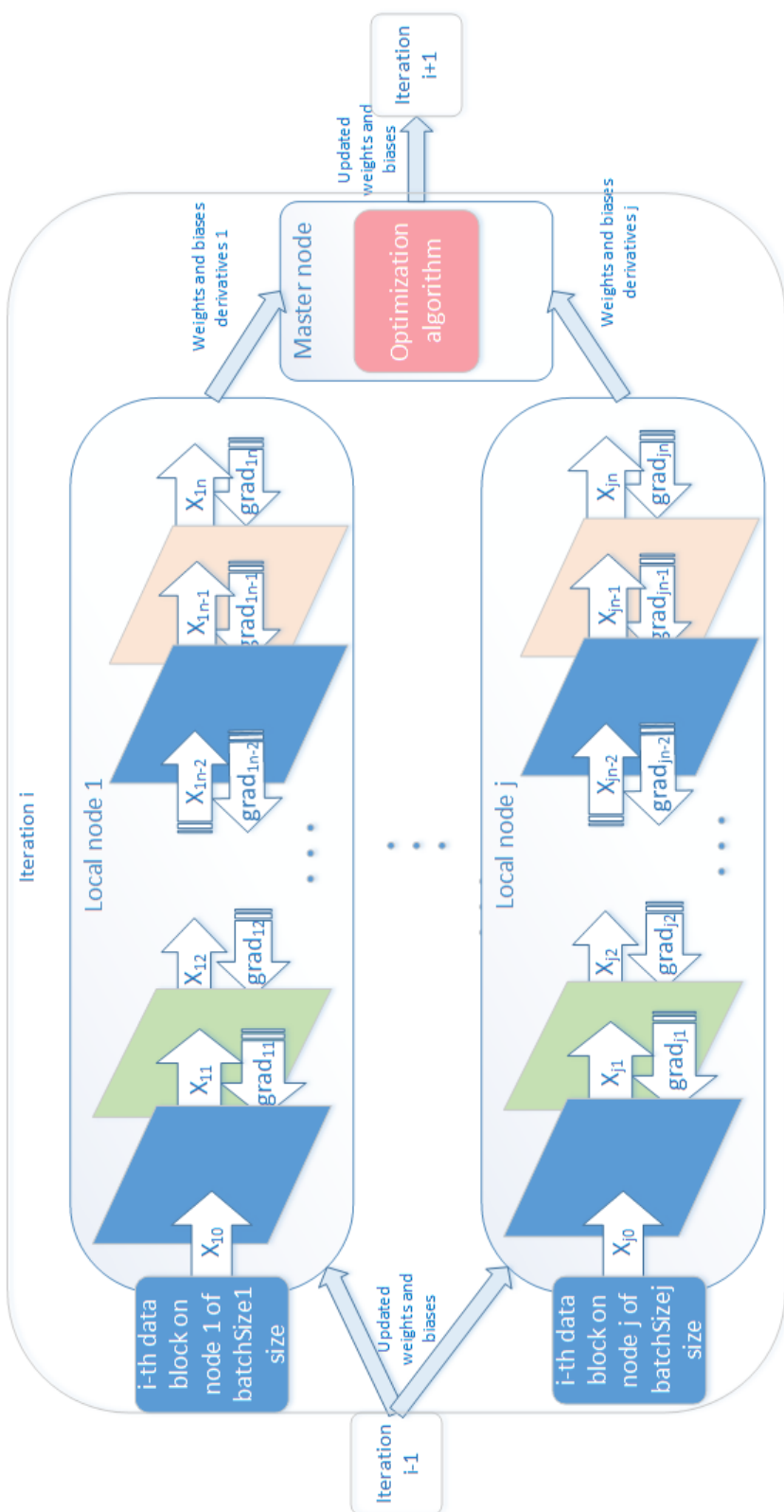
NOTE

The training algorithm on local nodes does not require an optimization solver.

3. Run the training algorithm on the master node by providing the local derivatives from all local nodes. The algorithm uses the optimization solver provided in its `optimizationSolver` parameter. For available algorithms, see [Optimization Solvers](#). After the computations are completed, send the updated weights and biases parameters of the model to all local nodes.

You can get the latest version of the model by calling the `finalizeCompute()` method after each run of the training algorithm on the master or local node.

4. Perform computations 2 - 3 for all data blocks. Call the `getPredictionModel()` method of the trained model on the master to get the model to be used for validation and prediction after the training process is completed.



Training

Algorithm Parameters

Neural network training in the distributed processing mode has the following parameters:

Parameter	Default Value	Description
<i>computeStep</i>	Not applicable	The parameter required to initialize the algorithm. Can be: <ul style="list-style-type: none"> <code>step1Local</code> - the first step, performed on local nodes <code>step2Master</code> - the second step, performed on a master node
<i>algorithmFPType</i>	<code>float</code>	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<i>method</i>	<code>defaultDense</code>	Performance-oriented computation method.
<i>batchSize</i>	128	The number of samples, simultaneously used for training on each node. The values of this parameter must be the same on all nodes for synchronous computations and may be different for asynchronous computations. This parameter is used by algorithms running on local nodes.
<i>optimization Solver</i>	<code>SharedPtr<optimization_solver::sgd::Batch<float, defaultDense >>()</code>	The optimization procedure used at the training stage. This parameter is used by the algorithm running on the master node.

Initialization

Neural network initialization

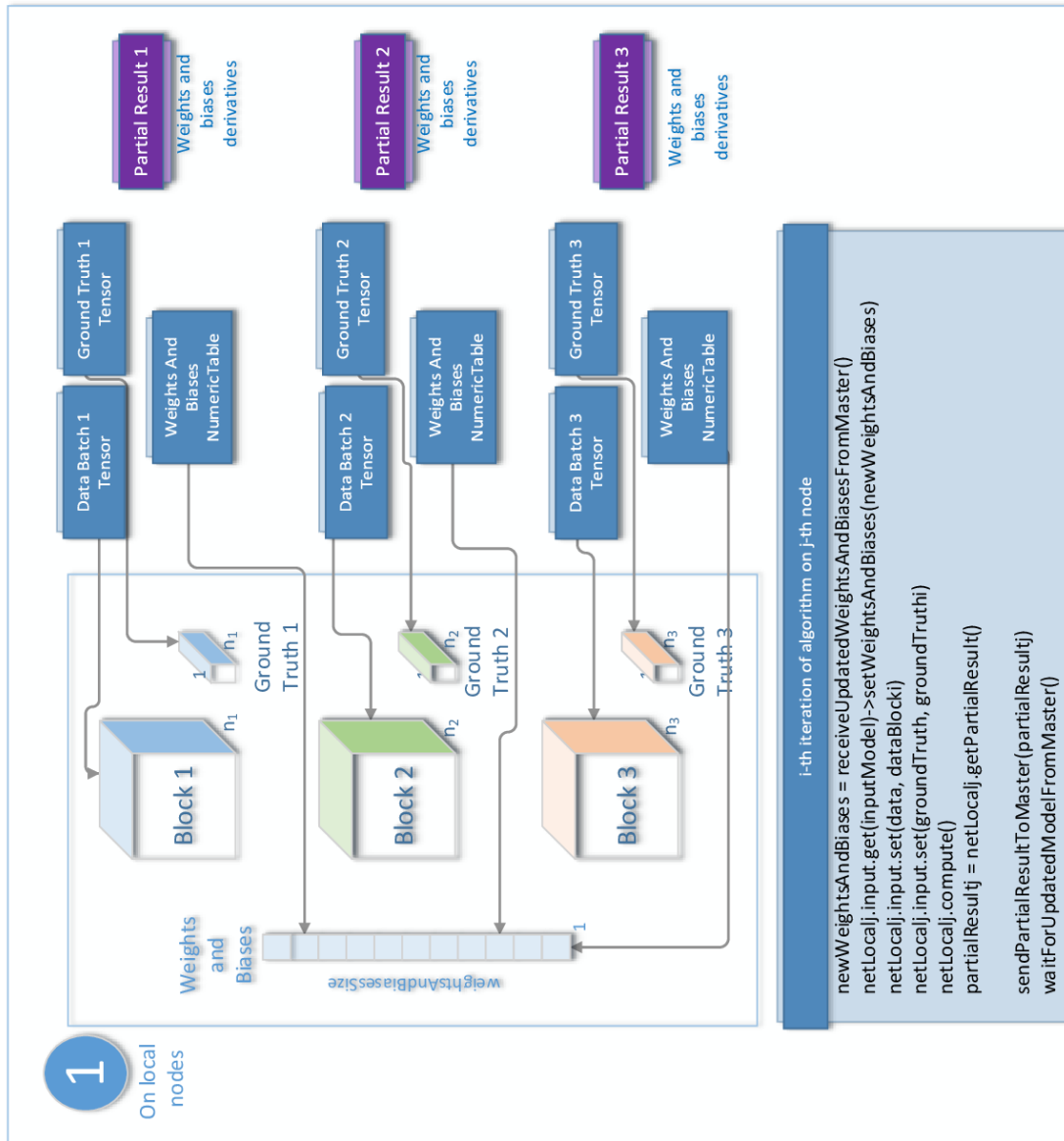
```
netMaster.initialize(inputDataDimensions, networkTopology)
trainingModel = netMaster.getResult()->get(model)
netLocal1.input.set(inputModel, trainingModel)
netLocal2.input.set(inputModel, trainingModel)
netLocal3.input.set(inputModel, trainingModel)
netLocal1.parameter.batchSize = batchSize1
netLocal2.parameter.batchSize = batchSize2
netLocal3.parameter.batchSize = batchSize3
```

Initialize batch sizes depending on whether the computation is synchronous or asynchronous:

- In the synchronous case, set `batchSize1 = batchSize2 = batchSize3`.
- In the asynchronous case, you can use different batch sizes.

Use the two-step computation schema for neural network training in the distributed processing mode, as illustrated below.

Step1 - on Local Nodes



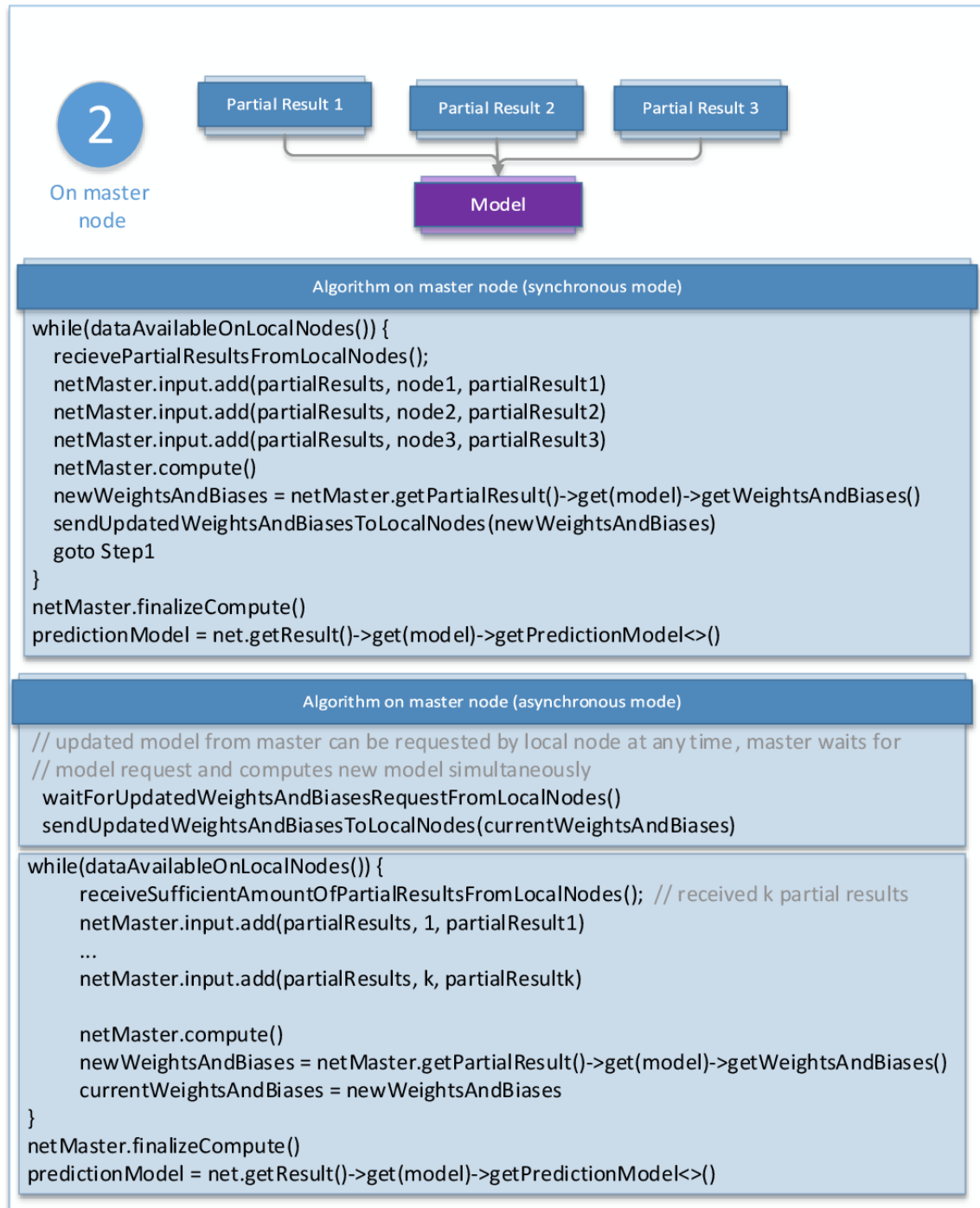
In this step, the neural network training algorithm accepts the following input. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
<code>data</code>	Pointer to the tensor of size $batchSize \times n_2 \times \dots \times n_p$ that represents the i -th data block on the local node. This input can be an object of any class derived from <code>Tensor</code> .
<code>groundTruth</code>	Pointer to the tensor of size $batchSize$ that stores i -th stated results associated with the input data. This input can be an object of any class derived from <code>Tensor</code> .
<code>inputModel</code>	The neural network model updated on the master node. This input can only be an object of the <code>Model</code> class.
Important Update the model parameters using the <code>setWeightsAndBiases()</code> method of <code>inputModel</code> after the master node delivers them to local nodes.	

In this step, the neural network training algorithm calculates partial results described below. Pass the Partial Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

PartialResultID	Partial Result
<code>derivatives</code>	Pointer to the numeric table of size $weightsAndBiasesSize \times 1$, where $weightsAndBiasesSize$ is the number of model parameters. By default this result is an object of the <code>HomogenNumericTable</code> class, but you can define this result as an object of any class derived from <code>NumericTable</code> except <code>PackedTriangularMatrix</code> , <code>PackedSymmetricMatrix</code> , and <code>CSRNumericTable</code> .
<code>batchSize</code>	Pointer to the numeric table of size 1×1 that contains the number of samples simultaneously used for training on each node. By default this partial result is an object of the <code>HomogenNumericTable</code> class, but you can define this numeric table as an object of any class derived from <code>NumericTable</code> except <code>PackedTriangularMatrix</code> , <code>PackedSymmetricMatrix</code> , and <code>CSRNumericTable</code> .

Step2 - on Master Node



In this step, the neural network training algorithm accepts the following input. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
partialResults	Results computed on local nodes in Step 1 (derivatives and batchSize). This input contains objects of the PartialResult class.

Input ID	Input
	<p>NOTE</p> <p>You can update the model on the master node incrementally, upon availability of partial results from local nodes, but be aware that in synchronous computations, the master node sends out the updated model only after it processes partial results from all local nodes.</p>

In this step, the neural network training algorithm calculates the results described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
model	Trained model with a set of weights and biases. This result can only be an object of the <code>Model</code> class.

Initializers

Initializer is an algorithm for early initialization of a p -dimensional tensor $W \in R^{n_1 \times \dots \times n_p}$ of a neural network model. In Intel DAAL, an initializer represents an algorithm interface that runs in-place initialization of memory according to the predefined method. This interface initializes parameters of a neural network. For more details, see [Usage Model: Training and Prediction](#).

Algorithm Parameters

Initializer algorithm has the following parameters:

Parameter	Default Value	Description
<code>layer</code>	<code>SharedPtr<layers::forward::LayerInterface>()</code>	Pointer to the layer whose weights and biases are initialized by the initializer. The initializer uses this pointer to get layer parameters such as sizes of the input and result.

Algorithm Input

Initializer algorithm accepts the following input. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
<code>tensorToInitialize</code>	Pointer to the tensor W of size $n_1 \times \dots \times n_p$ to initialize. This input can be an object of any class derived from <code>Tensor</code> .

Algorithm Output

Initializer algorithm calculates the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
value	Pointer to the initialized tensor W of size $n_1 \times \dots \times n_p$. This input can be an object of any class derived from <code>Tensor</code> . In Intel DAAL, the initialization is in-place, which means that the initializer does not allocate the result and always returns the pointer to the initialized input.

Uniform Initializer

A uniform initializer is an [initializer](#) algorithm to initialize a p -dimensional tensor $W \in R^{n_1 \times \dots \times n_p}$ with random numbers uniformly distributed on the interval $[a, b)$, where $a, b \in R$ and $a < b$.

Algorithm Parameters

In addition to common parameters of the initializer interface, a uniform initializer has the following parameters:

Parameter	Default Value	Description
<code>algorithmFPTType</code>	float	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<code>method</code>	defaultDense	Performance-oriented computation method, the only method supported by the algorithm.
<code>a</code>	-0.5	The left bound of the interval, a .
<code>b</code>	0.5	The right bound of the interval, b .
<code>seed</code>	777	A seed for random generation of uniformly distributed variates.

Xavier Initializer

A Xavier initializer is an [initializer](#) algorithm to initialize a p -dimensional tensor $W \in R^{n_1 \times \dots \times n_p}$ that represents weights and biases of the appropriate layer. The algorithm initializes this tensor with random numbers uniformly distributed on the interval $[-\alpha, \alpha)$. The value of α is defined using the sizes of the r -dimensional input tensor $X \in R^{n \times m_2 \times \dots \times m_r}$ and q -dimensional value tensor $Y \in R^{n \times k_2 \times \dots \times k_q}$ for the layer:

$$\alpha = \sqrt{\frac{6}{n_{in} + n_{out}}},$$

where:

- $n_{in} = m_2 \cdot \dots \cdot m_r$
- $n_{out} = k_2 \cdot \dots \cdot k_q$
- It is assumed without loss of generality that tensors X and Y have batch dimension of size n

For more details, see [\[Glorot2010\]](#).

Algorithm Parameters

In addition to common parameters of the initializer interface, a Xavier initializer has the following parameters:

Parameter	Default Value	Description
<i>algorithmFPType</i>	<code>float</code>	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<i>method</i>	<code>defaultDense</code>	Performance-oriented computation method, the only method supported by the algorithm.
<i>seed</i>	<code>777</code>	A seed for random generation of uniformly distributed variates.

Gaussian Initializer

A Gaussian initializer is an [initializer](#) algorithm to initialize a p -dimensional tensor $X \in \mathbb{R}^{n_1 \times \dots \times n_p}$ with Gaussian random numbers having mean a and standard deviation σ , where $a, \sigma \in \mathbb{R}$ and $\sigma > 0$.

Algorithm Parameters

In addition to common parameters of the initializer interface, a Gaussian initializer has the following parameters:

Parameter	Default Value	Description
<i>algorithmFPType</i>	<code>float</code>	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<i>method</i>	<code>defaultDense</code>	Performance-oriented computation method, the only method supported by the algorithm.
<i>a</i>	<code>0</code>	The mean a .
<i>sigma</i>	<code>0.01</code>	The standard deviation σ .
<i>seed</i>	<code>777</code>	A seed for random generation of Gaussian variates.

Truncated Gaussian Initializer

A truncated Gaussian initializer is an [initializer](#) algorithm to initialize a p -dimensional tensor $X \in \mathbb{R}^{n_1 \times \dots \times n_p}$ with variates that have the Gaussian probability density function with mean μ and standard deviation σ and belong to the truncation range $[a, b]$, where $b > a$, $\mu, \sigma \in \mathbb{R}$, and $\sigma > 0$.

The bounds of the truncation range define the following cases:

- No truncation: $a = -\infty, b = +\infty$
- Left truncation: $a > -\infty, b = +\infty$
- Right truncation: $a = -\infty, b < +\infty$
- Two-sided truncation: $a > -\infty, b < +\infty$

Algorithm Parameters

In addition to common parameters of the initializer interface, a truncated Gaussian initializer has the following parameters:

Parameter	Default Value	Description
<i>algorithmFPTYPE</i>	float	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<i>method</i>	defaultDense	Inverse cumulative distribution function (CDF) transform method. Performance-oriented computation method, the only method supported by the algorithm.
<i>mean</i>	0	The mean μ .
<i>sigma</i>	1	The standard deviation σ .
<i>a</i>	$mean - 2 * sigma$	The left bound of the truncation range. If it is set to the negative infinity, there is no left truncation.
<i>b</i>	$mean + 2 * sigma$	The right bound of the truncation range. If it is set to the positive infinity, there is no right truncation.
<i>seed</i>	777	A seed for random generation of truncated Gaussian variates.

Layers

Types of Layers Implemented

Intel DAAL provides the following types of layers:

- Fully-connected layers,

which compute the inner product of all weighed inputs plus bias.

Fully-connected Layers	Forward Backward
------------------------	--

- Activation layers,

which apply a transform to the input data.

Absolute Value (Abs) Layers	Forward Backward
Logistic Layers	Forward Backward
Parametric Rectifier Linear Unit (pReLU) Layers	Forward Backward
Rectifier Linear Unit (ReLU) Layers	Forward Backward
Smooth Rectifier Linear Unit (SmoothReLU) Layers	Forward Backward
Hyperbolic Tangent Layers	Forward Backward

- Normalization layers,

which normalize the input data.

Batch Normalization Layers	Forward Backward
Local Response Normalization Layers	Forward Backward
Local Contrast Normalization Layers	Forward Backward

- Anti-overfitting layers,

which prevent the neural network from overfitting.

Dropout Layers	Forward Backward
----------------	--

- Pooling layers,

which apply a form of non-linear downsampling to input data.

1D Max Pooling Layers	Forward Backward
2D Max Pooling Layers	Forward Backward
3D Max Pooling Layers	Forward Backward
1D Average Pooling Layers	Forward Backward
2D Average Pooling Layers	Forward Backward
3D Average Pooling Layers	Forward Backward
2D Stochastic Pooling Layers	Forward Backward
2D Spatial Pyramid Pooling Layers	Forward Backward

- Convolutional and locally-connected layers,
which apply filters to input data.

2D Convolution Layers	Forward Backward
2D Transposed Convolution Layers	Forward Backward
2D Locally-connected Layers	Forward Backward

- Service layers,
which apply service operations to the input tensors.

Reshape Layers	Forward Backward
Concat Layers	Forward Backward
Split Layers	Forward Backward

- Softmax layers,
which measure confidence of the output of the neural network.

Softmax Layers	Forward Backward
----------------	--

- Loss layers,
which measure the difference between the output of the neural network and ground truth.

Loss Layers	Forward Backward
Loss Softmax Cross-entropy Layers	Forward Backward
Loss Logistic Cross-entropy Layers	Forward Backward

NOTE

In the descriptions of specific layers, the preceding layer for the layer i is the layer:

- $i - 1$ for forward layers.
- $i + 1$ for backward layers.

Assumptions

When using Intel DAAL neural networks, be aware of the following assumptions:

- In Intel DAAL, numbering of data samples is scalar.
- For neural network layers, the first dimension of the input tensor represents the data samples.

While the actual layout of the data can be different, the access methods of the tensor return the data in the assumed layout. Therefore, for a tensor containing the input to the neural network, it is your responsibility to change logical indexing of tensor dimensions so that the first dimension represents the sample data. To do this, use the `shuffleDimensions()` method of the `Tensor` class.

Common Parameters

Neural network layers have the following common parameters:

Parameter	Default Value	Description
<i>weights Initializer</i>	<code>services::SharedPtr<initializers::InitializerIface>(new initializers::uniform::Batch<>())</code>	Weights initializer for the layer.
<i>biases Initializer</i>	<code>services::SharedPtr<initializers::InitializerIface>(new initializers::uniform::Batch<>())</code>	Biases initializer for the layer.
<i>predictionStage</i>	<code>false</code>	A flag that specifies whether the layer is used at the prediction stage. Applies to forward layers.
<i>weightsAndBiases Initialized</i>	<code>false</code>	Initialization status of weights and biases: <ul style="list-style-type: none"> <code>true</code> if the user provides weights and biases. It is your responsibility to set the flag to <code>true</code> so that the layer uses the user-defined weights and biases. <code>false</code> if the layer uses the default initializer or the one selected by the user from the list of supported initializers to initialize weights and biases.

Fully-connected Forward Layer

The forward fully-connected layer computes values

$$y_i(x_1, \dots, x_n) = \sum_{k=1}^n s_{ki} w_{ki} x_k + b_i$$

for n input arguments x_k , weights w_{ki} , weights mask s_{ki} , and biases b_i , where $k \in \{1, \dots, n\}$, $i \in \{1, \dots, m\}$, and m is the number of layer outputs.

Problem Statement

Given:

- Dimension k
- p -dimensional tensor X of size $n_1 \times \dots \times n_k \dots \times n_p$
- p -dimensional tensor W of size $n_1 \times \dots \times n_{k-1} \times m \times n_{k+1} \dots \times n_p$
- p -dimensional tensor S of size $n_1 \times \dots \times n_{k-1} \times m \times n_{k+1} \dots \times n_p$
- 1-dimensional tensor B of size m

The problem is to compute the 2-dimensional tensor Y of size $n_k \times m$ such that:

$$y_{i_k j} = \sum_{i_1=1}^{n_1} \dots \sum_{i_{k-1}=1}^{n_{k-1}} \sum_{i_{k+1}=1}^{n_{k+1}} \dots \sum_{i_p=1}^{n_p} s_{i_1 \dots i_{k-1} j i_{k+1} \dots i_p} w_{i_1 \dots i_{k-1} j i_{k+1} \dots i_p} x_{i_1 \dots i_k \dots i_p} + b_j,$$

$$i_k = \{1, \dots, n_k\},$$

$$j = \{1, \dots, m\}.$$

Batch Processing

Layer Input

The forward fully-connected layer accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
<code>data</code>	Pointer to the tensor of size $n_1 \times \dots \times n_k \times \dots \times n_p$ that stores the input data for the forward fully-connected layer. This input can be an object of any class derived from <code>Tensor</code> .
<code>weights</code>	Pointer to the tensor of size $n_1 \times \dots \times n_{k-1} \times m \times n_{k+1} \times \dots \times n_p$ that stores a set of weights. This input can be an object of any class derived from <code>Tensor</code> .
<code>biases</code>	Pointer to the tensor of size m that stores a set of biases. This input can be an object of any class derived from <code>Tensor</code> .
<code>mask</code>	Pointer to the tensor of size $n_1 \times \dots \times n_{k-1} \times m \times n_{k+1} \times \dots \times n_p$ that holds 1 for the corresponding weights used in computations and 0 for the weights not used in computations. If no mask is provided, the library uses all the weights.

Layer Parameters

For common parameters of neural network layers, see [Common Parameters](#).

In addition to the common parameters, the forward fully-connected layer has the following parameters:

Parameter	Default Value	Description
<code>algorithmFPType</code>	<code>float</code>	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<code>method</code>	<code>defaultDense</code>	Performance-oriented computation method, the only method supported by the layer.
<code>nOutputs</code>	Not applicable	Number of layer outputs m . Required to initialize the algorithm.
<code>dimension</code>	0	Dimension k for which the forward propagation step of the fully-connected layer is performed.

Layer Output

The forward fully-connected layer calculates the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
<code>value</code>	Pointer to the tensor of size $n_k \times m$ that stores the result of the forward fully-connected layer. This input can be an object of any class derived from <code>Tensor</code> .

Result ID	Result	
resultForBackward	Collection of data needed for the backward fully-connected layer.	
	Element ID	Element
	auxData	Pointer to the tensor of size $n_1 \times \dots \times n_k \times \dots \times n_p$ that stores the input data for the forward fully-connected layer. This input can be an object of any class derived from <code>Tensor</code> .
	auxWeights	Pointer to the tensor of size $n_1 \times \dots \times n_{k-1} \times m \times n_{k+1} \times \dots \times n_p$ that stores a set of weights. This input can be an object of any class derived from <code>Tensor</code> .
	auxBiases	Pointer to the tensor of size m that stores a set of biases. This input can be an object of any class derived from <code>Tensor</code> .
	auxMask	Pointer to the tensor of size $n_1 \times \dots \times n_{k-1} \times m \times n_{k+1} \times \dots \times n_p$ that holds 1 for the corresponding weights used in computations and 0 for the weights not used in computations. If no mask is provided, the library uses all the weights.

Examples

Refer to the following examples in your Intel DAAL directory:

C++: `./examples/cpp/source/neural_networks/fullyconnected_layer_batch.cpp`

Java*: `./examples/java/source/com/intel/daal/examples/neural_networks/FullyconnectedLayerBatch.java`

Python*: `./examples/python/source/neural_networks/fullyconnected_layer_batch.py`

Fully-connected Backward Layer

The forward fully-connected layer computes values

$$y_i(x_1, \dots, x_n) = \sum_{k=1}^n s_{ki} w_{ki} x_k + b_i$$

for n input arguments x_k , weights w_{ki} , weights mask s_{ki} , and biases b_i , where $k \in \{1, \dots, n\}$, $i \in \{1, \dots, m\}$, and m is the number of layer outputs. For more details, see [Forward Fully-connected Layer](#).

The backward fully-connected layer computes the following values:

$$z_k = \sum_{j=1}^m g_j \cdot s_{kj} \cdot w_{kj},$$

$$\frac{\partial E}{\partial w_{kj}} = g_j \cdot s_{kj} \cdot x_k,$$

$$\frac{\partial E}{\partial b_j} = g_j,$$

where

$$k = \{1, \dots, n\}, j = \{1, \dots, m\},$$

E is the objective function used at the training stage, and g_j is the input gradient computed on the preceding layer.

Problem Statement

Given:

- p -dimensional tensor X of size $n_1 \times \dots \times n_k \dots \times n_p$
- p -dimensional tensor W of size $n_1 \times \dots \times n_{k-1} \times m \times n_{k+1} \dots \times n_p$
- p -dimensional tensor S of size $n_1 \times \dots \times n_{k-1} \times m \times n_{k+1} \dots \times n_p$
- 1-dimensional tensor B of size m
- 2-dimensional tensor G of size $n_k \times m$

The problem is to compute:

- The p -dimensional tensor Z of size $n_1 \times \dots \times n_k \dots \times n_p$ such that:

$$z_{i_1 \dots i_k \dots i_p} = \sum_{j=1}^m g_{i_k j} \cdot s_{i_1 \dots i_{k-1} j i_{k+1} \dots i_p} \cdot w_{i_1 \dots i_{k-1} j i_{k+1} \dots i_p}$$

- Values:

$$\frac{\partial E}{\partial w_{i_1 \dots j \dots i_p}} = \frac{1}{n_k} \sum_{i_k=1}^{n_k} g_{i_k j} \cdot s_{i_1 \dots j \dots i_p} \cdot x_{i_1 \dots i_k \dots i_p},$$

$$\frac{\partial E}{\partial b_j} = \frac{1}{n_k} \sum_{i_k=1}^{n_k} g_{i_k j}.$$

In the above formulas:

$$i_k = \{1, \dots, n_k\},$$

$$j = \{1, \dots, m\}.$$

Batch Processing

Layer Input

The backward fully-connected layer accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
<code>inputGradient</code>	Pointer to the tensor of size $n_k \times m$ that stores the input gradient computed on the preceding layer. This input can be an object of any class derived from <code>Tensor</code> .
<code>inputFromForward</code>	Collection of data needed for the backward fully-connected layer. This collection can contain objects of any class derived from <code>Tensor</code> .
Element ID	Element

Input ID	Input
	<p><code>auxData</code></p> <p>Pointer to the tensor of size $n_1 \times \dots \times n_k \times \dots \times n_p$ that stores the input data from the forward fully-connected layer. This input can be an object of any class derived from <code>Tensor</code>.</p>
	<p><code>auxWeights</code></p> <p>Pointer to the tensor of size $n_1 \times \dots \times n_{k-1} \times m \times n_{k+1} \times \dots \times n_p$ that stores a set of weights. This input can be an object of any class derived from <code>Tensor</code>.</p>
	<p><code>auxBiases</code></p> <p>Pointer to the tensor of size m that stores a set of biases. This input can be an object of any class derived from <code>Tensor</code>.</p>
	<p><code>auxMask</code></p> <p>Pointer to the tensor of size $n_1 \times \dots \times n_{k-1} \times m \times n_{k+1} \times \dots \times n_p$ that holds 1 for the corresponding weights used in computations and 0 for the weights not used in computations. If no mask is provided, the library uses all the weights.</p>

Layer Parameters

For common parameters of neural network layers, see [Common Parameters](#).

In addition to the common parameters, the backward fully-connected layer has the following parameters:

Parameter	Default Value	Description
<code>algorithmFPType</code>	<code>float</code>	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<code>method</code>	<code>defaultDense</code>	Performance-oriented computation method, the only method supported by the layer.
<code>nOutputs</code>	Not applicable	Number of layer outputs m . Required to initialize the algorithm.
<code>dimension</code>	0	Dimension k for which the backward propagation step of the fully-connected layer is performed.
<code>propagateGradient</code>	<code>false</code>	Flag that specifies whether the backward layer propagates the gradient.

Layer Output

The backward fully-connected layer calculates the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
gradient	Pointer to the tensor of size $n_1 \times \dots \times n_k \times \dots \times n_p$ that stores the result of the backward fully-connected layer. This input can be an object of any class derived from <code>Tensor</code> .
weightDerivatives	Pointer to the tensor of size $n_1 \times \dots \times n_{k-1} \times m \times n_{k+1} \times \dots \times n_p$ that stores result $\partial E / \partial w_{i_1 \dots j \dots i_p}$ of the backward fully-connected layer, where $j = \{1, \dots, m\}$. This input can be an object of any class derived from <code>Tensor</code> .
biasDerivatives	Pointer to the tensor of size m that stores result $\partial E / \partial b_j$ of the backward fully-connected layer, where $j = \{1, \dots, m\}$. This input can be an object of any class derived from <code>Tensor</code> .

Examples

Refer to the following examples in your Intel DAAL directory:

C++: `./examples/cpp/source/neural_networks/fullyconnected_layer_batch.cpp`

Java*: `./examples/java/source/com/intel/daal/examples/neural_networks/
FullyconnectedLayerBatch.java`

Python*: `./examples/python/source/neural_networks/fullyconnected_layer_batch.py`

Absolute Value (Abs) Forward Layer

The forward abs layer computes the absolute value of input argument x .

Problem Statement

Given a p -dimensional tensor X of size $n_1 \times n_2 \times \dots \times n_p$, the problem is to compute a p -dimensional tensor $Y = (y_{i_1 \dots i_p})$ of size $n_1 \times n_2 \times \dots \times n_p$, where:

$$y_{i_1 \dots i_p} = \text{abs}(x_{i_1 \dots i_p})$$

Batch Processing

Layer Input

The forward abs layer accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
data	Pointer to the tensor of size $n_1 \times n_2 \times \dots \times n_p$ that stores the input data for the forward abs layer. This input can be an object of any class derived from <code>Tensor</code> .

Layer Parameters

For common parameters of neural network layers, see [Common Parameters](#).

In addition to the common parameters, the forward abs layer has the following parameters:

Parameter	Default Value	Description
<code>algorithmFPTType</code>	<code>float</code>	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<code>method</code>	<code>defaultDense</code>	Performance-oriented computation method, the only method supported by the layer.

Layer Output

The forward abs layer calculates the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result				
<code>value</code>	Pointer to the tensor of size $n_1 \times n_2 \times \dots \times n_p$ that stores the result of the forward abs layer. This input can be an object of any class derived from <code>Tensor</code> .				
<code>resultForBackward</code>	Collection of data needed for the backward abs layer.				
	<table> <tr> <th>Element ID</th><th>Element</th></tr> <tr> <td><code>auxData</code></td><td>Pointer to the tensor of size $n_1 \times n_2 \times \dots \times n_p$ that stores the input data for the forward abs layer. This input can be an object of any class derived from <code>Tensor</code>.</td></tr> </table>	Element ID	Element	<code>auxData</code>	Pointer to the tensor of size $n_1 \times n_2 \times \dots \times n_p$ that stores the input data for the forward abs layer. This input can be an object of any class derived from <code>Tensor</code> .
Element ID	Element				
<code>auxData</code>	Pointer to the tensor of size $n_1 \times n_2 \times \dots \times n_p$ that stores the input data for the forward abs layer. This input can be an object of any class derived from <code>Tensor</code> .				

Examples

Refer to the following examples in your Intel DAAL directory:

C++: `./examples/cpp/source/neural_networks/abs_layer_batch.cpp`

Java*: `./examples/java/source/com/intel/daal/examples/neural_networks/AbsLayerBatch.java`

Python*: `./examples/python/source/neural_networks/abs_layer_batch.py`

Absolute Value (Abs) Backward Layer

The absolute value (abs) activation layer applies the transform $f(x) = \text{abs}(x)$ to the input data. The backward abs layer computes the value $z = y * f'(x)$, where y is the input gradient computed on the prior layer and $f'(x) = \{1 \text{ if } x > 0, 0 \text{ if } x = 0, -1 \text{ if } x < 0\}$.

Problem Statement

Given p -dimensional tensors X and Y of size $n_1 \times n_2 \times \dots \times n_p$, the problem is to compute a p -dimensional tensor $Z = (z_{i_1 \dots i_p})$ of size $n_1 \times n_2 \times \dots \times n_p$, where:

$$z_{i_1 \dots i_p} = y_{i_1 \dots i_p} * \text{sign}(x_{i_1 \dots i_p})$$

Batch Processing

Layer Input

The backward abs layer accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
<code>inputGradient</code>	Pointer to the tensor of size $n_1 \times n_2 \times \dots \times n_p$ that stores the input gradient computed on the preceding layer. This input can be an object of any class derived from <code>Tensor</code> .
<code>inputFromForward</code>	Collection of data needed for the backward abs layer.
Element ID	Element
<code>auxData</code>	Pointer to the tensor of size $n_1 \times n_2 \times \dots \times n_p$ that stores the input data for the forward abs layer. This input can be an object of any class derived from <code>Tensor</code> .

Layer Parameters

For common parameters of neural network layers, see [Common Parameters](#).

In addition to the common parameters, the backward abs layer has the following parameters:

Parameter	Default Value	Description
<code>algorithmFPType</code>	<code>float</code>	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<code>method</code>	<code>defaultDense</code>	Performance-oriented computation method, the only method supported by the layer.

Layer Output

The backward abs layer calculates the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
<code>gradient</code>	Pointer to the tensor of size $n_1 \times n_2 \times \dots \times n_p$ that stores the result of the backward abs layer. This input can be an object of any class derived from <code>Tensor</code> .

Examples

Refer to the following examples in your Intel DAAL directory:

C++: `./examples/cpp/source/neural_networks/abs_layer_batch.cpp`

Java*: `./examples/java/source/com/intel/daal/examples/neural_networks/AbsLayerBatch.java`

Python*: `./examples/python/source/neural_networks/abs_layer_batch.py`

Logistic Forward Layer

The forward logistic layer computes the function

$$f(x) = \frac{1}{1 + \exp(-x)}$$

for the input argument x .

Problem Statement

Given a p -dimensional tensor X of size $n_1 \times n_2 \times \dots \times n_p$, the problem is to compute the p -dimensional tensor $Y = (y_{i_1 \dots i_p})$ of size $n_1 \times n_2 \times \dots \times n_p$ such that:

$$y_{i_1 \dots i_p} = \frac{1}{1 + \exp\left(-x_{i_1 \dots i_p}\right)}.$$

Batch Processing

Layer Input

The forward logistic layer accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
data	Pointer to the tensor of size $n_1 \times n_2 \times \dots \times n_p$ that stores the input data for the forward logistic layer. This input can be an object of any class derived from <code>Tensor</code> .

Layer Parameters

For common parameters of neural network layers, see [Common Parameters](#).

In addition to the common parameters, the forward logistic layer has the following parameters:

Parameter	Default Value	Description
<code>algorithmFPTType</code>	<code>float</code>	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<code>method</code>	<code>defaultDense</code>	Performance-oriented computation method, the only method supported by the layer.

Layer Output

The forward logistic layer calculates the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result				
value	Pointer to the numeric tensor of size $n_1 \times n_2 \times \dots \times n_p$ that stores the result of the forward logistic layer. This input can be an object of any class derived from <code>Tensor</code> .				
resultForBackward	Collection of data needed for the backward logistic layer.				
	<table> <tr> <th>Element ID</th><th>Element</th></tr> <tr> <td>auxValue</td><td>Pointer to the tensor of size $n_1 \times n_2 \times \dots \times n_p$ that stores the result of the forward logistic layer. This input can be an object of any class derived from <code>Tensor</code>.</td></tr> </table>	Element ID	Element	auxValue	Pointer to the tensor of size $n_1 \times n_2 \times \dots \times n_p$ that stores the result of the forward logistic layer. This input can be an object of any class derived from <code>Tensor</code> .
Element ID	Element				
auxValue	Pointer to the tensor of size $n_1 \times n_2 \times \dots \times n_p$ that stores the result of the forward logistic layer. This input can be an object of any class derived from <code>Tensor</code> .				

Examples

Refer to the following examples in your Intel DAAL directory:

C++: `./examples/cpp/source/neural_networks/logistic_layer_batch.cpp`

Java*: `./examples/java/source/com/intel/daal/examples/neural_networks/LogisticLayerBatch.java`

Python*: `./examples/python/source/neural_networks/logistic_layer_batch.py`

Logistic Backward Layer

The logistic activation layer applies the transform

$$f(x) = \frac{1}{1 + \exp(-x)}$$

to the input data. The backward logistic layer computes the values $z = y * f'(x)$, where y is the input gradient computed on the preceding layer and

$$f'(x) = \frac{-\exp(-x)}{(1 + \exp(-x))^2} = f(x) * (1 - f(x)).$$

Problem Statement

Given p -dimensional tensors X and Y of size $n_1 \times n_2 \times \dots \times n_p$, the problem is to compute the p -dimensional tensor $Z = (z_{i_1 \dots i_p})$ of size $n_1 \times n_2 \times \dots \times n_p$ such that:

$$z_{i_1 \dots i_p} = y_{i_1 \dots i_p} * f(x_{i_1 \dots i_p}) * (1 - f(x_{i_1 \dots i_p})).$$

Batch Processing

Layer Input

The backward logistic layer accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input				
inputGradient	Pointer to the tensor of size $n_1 \times n_2 \times \dots \times n_p$ that stores the input gradient computed on the preceding layer. This input can be an object of any class derived from <code>Tensor</code> .				
inputFromForward	Collection of data needed for the backward logistic layer.				
	<table> <tr> <th>Element ID</th><th>Element</th></tr> <tr> <td>auxValue</td><td>Pointer to the tensor of size $n_1 \times n_2 \times \dots \times n_p$ that stores the result of the forward logistic layer. This input can be an object of any class derived from <code>Tensor</code>.</td></tr> </table>	Element ID	Element	auxValue	Pointer to the tensor of size $n_1 \times n_2 \times \dots \times n_p$ that stores the result of the forward logistic layer. This input can be an object of any class derived from <code>Tensor</code> .
Element ID	Element				
auxValue	Pointer to the tensor of size $n_1 \times n_2 \times \dots \times n_p$ that stores the result of the forward logistic layer. This input can be an object of any class derived from <code>Tensor</code> .				

Layer Parameters

For common parameters of neural network layers, see [Common Parameters](#).

In addition to the common parameters, the backward logistic layer has the following parameters:

Parameter	Default Value	Description
<code>algorithmFPTType</code>	<code>float</code>	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<code>method</code>	<code>defaultDense</code>	Performance-oriented computation method, the only method supported by the layer.

Layer Output

The backward logistic layer calculates the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
<code>gradient</code>	Pointer to the tensor of size $n_1 \times n_2 \times \dots \times n_p$ that stores the result of the backward logistic layer. This input can be an object of any class derived from <code>Tensor</code> .

Examples

Refer to the following examples in your Intel DAAL directory:

C++: `./examples/cpp/source/neural_networks/logistic_layer_batch.cpp`

Java*: `./examples/java/source/com/intel/daal/examples/neural_networks/LogisticLayerBatch.java`

Python*: `./examples/python/source/neural_networks/logistic_layer_batch.py`

Parametric Rectifier Linear Unit (pReLU) Forward Layer

The forward parametric rectifier linear unit (pReLU) layer computes the function $f(x) = \max(0, x) + w * \min(0, x)$ for the input argument x , where w is the weight of this argument [[HeZhangRenSun](#)].

Problem Statement

Given p -dimensional tensors X and W of size $n_1 \times n_2 \times \dots \times n_p$, the problem is to compute the p -dimensional tensor $Y = (y_{i_1 \dots i_p})$ of size $n_1 \times n_2 \times \dots \times n_p$ such that:

$$y_{i_1 \dots i_p} = \max(0, x_{i_1 \dots i_p}) + w_{i_1 \dots i_p} * \min(0, x_{i_1 \dots i_p}).$$

Batch Processing

Layer Input

The forward pReLU layer accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
<code>data</code>	Pointer to the tensor of size $n_1 \times n_2 \times \dots \times n_p$ that stores the input data for the forward pReLU layer. This input can be an object of any class derived from <code>Tensor</code> .

Input ID	Input
<code>weights</code>	Pointer to the tensor of size $n_k \times n_{k+1} \times \dots \times n_{k+q-1}$ that stores weights for the forward pReLU layer. This input can be an object of any class derived from <code>Tensor</code> .

Layer Parameters

For common parameters of neural network layers, see [Common Parameters](#).

In addition to the common parameters, the forward pReLU layer has the following parameters:

Parameter	Default Value	Description
<code>algorithmFPType</code>	<code>float</code>	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<code>method</code>	<code>defaultDense</code>	Performance-oriented computation method, the only method supported by the layer.
<code>dataDimension</code>	0	Starting index of data dimension of type <code>size_t</code> to apply a weight.
<code>weightsDimension</code>	1	Number of weight dimensions of type <code>size_t</code> .

Layer Output

The forward pReLU layer calculates the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result						
<code>value</code>	Pointer to the tensor of size $n_1 \times n_2 \times \dots \times n_p$ that stores the results of the forward pReLU layer. This input can be an object of any class derived from <code>Tensor</code> .						
<code>resultForBackward</code>	Collection of data needed for the backward pReLU layer. <table> <tr> <th>Element ID</th><th>Element</th></tr> <tr> <td><code>auxData</code></td><td>Pointer to the tensor of size $n_1 \times n_2 \times \dots \times n_p$ that stores the input data for the forward pReLU layer. This input can be an object of any class derived from <code>Tensor</code>.</td></tr> <tr> <td><code>auxWeights</code></td><td>Pointer to the tensor of size $n_k \times n_{k+1} \times \dots \times n_{k+q-1}$ that stores weights for the forward pReLU layer. This input can be an object of any class derived from <code>Tensor</code>.</td></tr> </table>	Element ID	Element	<code>auxData</code>	Pointer to the tensor of size $n_1 \times n_2 \times \dots \times n_p$ that stores the input data for the forward pReLU layer. This input can be an object of any class derived from <code>Tensor</code> .	<code>auxWeights</code>	Pointer to the tensor of size $n_k \times n_{k+1} \times \dots \times n_{k+q-1}$ that stores weights for the forward pReLU layer. This input can be an object of any class derived from <code>Tensor</code> .
Element ID	Element						
<code>auxData</code>	Pointer to the tensor of size $n_1 \times n_2 \times \dots \times n_p$ that stores the input data for the forward pReLU layer. This input can be an object of any class derived from <code>Tensor</code> .						
<code>auxWeights</code>	Pointer to the tensor of size $n_k \times n_{k+1} \times \dots \times n_{k+q-1}$ that stores weights for the forward pReLU layer. This input can be an object of any class derived from <code>Tensor</code> .						

Examples

Refer to the following examples in your Intel DAAL directory:

C++: `./examples/cpp/source/neural_networks/prelu_layer_batch.cpp`

Java*: `./examples/java/source/com/intel/daal/examples/neural_networks/PreLULayerBatch.java`

Python*: `./examples/python/source/neural_networks/prelu_layer_batch.py`

Parametric Rectifier Linear Unit (pReLU) Backward Layer

The parametric rectifier linear unit (pReLU) activation layer applies the transform $f(x) = \max(0, x) + w * \min(0, x)$ to the input data. The backward pReLU layer computes the values $z = y * f'(x)$, where y is the input gradient computed on the preceding layer, w is the weight of the input argument. and

$$f'(x) = \begin{cases} 1 & \text{if } x > 0, \\ 0 & \text{if } x = 0, \\ w & \text{if } x < 0. \end{cases}$$

Problem Statement

Given p -dimensional tensors X, Y , and W of size $n_1 \times n_2 \times \dots \times n_p$, the problem is to compute the p -dimensional tensor $Z = (z_{i_1 \dots i_p})$ of size $n_1 \times n_2 \times \dots \times n_p$ such that:

$$z_{i_1 \dots i_p} = \begin{cases} y_{i_1 \dots i_p} & \text{if } x_{i_1 \dots i_p} > 0, \\ 0 & \text{if } x_{i_1 \dots i_p} = 0, \\ y_{i_1 \dots i_p} * w_{i_1 \dots i_p} & \text{if } x_{i_1 \dots i_p} < 0. \end{cases}$$

Batch Processing

Layer Input

The backward pReLU layer accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
<code>inputGradient</code>	Pointer to the tensor of size $n_1 \times n_2 \times \dots \times n_p$ that stores the input gradient computed on the preceding layer. This input can be an object of any class derived from <code>Tensor</code> .
<code>inputFromForward</code>	Collection of data needed for the backward pReLU layer. This collection can contain objects of any class derived from <code>Tensor</code> .
Element ID	Element
<code>auxData</code>	Pointer to the tensor of size $n_1 \times n_2 \times \dots \times n_p$ that stores the input data for the forward pReLU layer. This input can be an object of any class derived from <code>Tensor</code> .
<code>auxWeights</code>	Pointer to the tensor of size $n_k \times n_{k+1} \times \dots \times n_{k+q-1}$ that stores weights for the forward pReLU layer. This input can be an object of any class derived from <code>Tensor</code> .

Layer Parameters

For common parameters of neural network layers, see [Common Parameters](#).

In addition to the common parameters, the backward pReLU layer has the following parameters:

Parameter	Default Value	Description
<code>algorithmFPType</code>	<code>float</code>	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<code>method</code>	<code>defaultDense</code>	Performance-oriented computation method, the only method supported by the layer.
<code>dataDimension</code>	<code>0</code>	Starting index of data dimension of type <code>size_t</code> to apply a weight.
<code>weightsDimension</code>	<code>1</code>	Number of weight dimensions of type <code>size_t</code> .
<code>propagateGradient</code>	<code>false</code>	Flag that specifies whether the backward layer propagates the gradient.

Layer Output

The backward pReLU layer calculates the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
<code>gradient</code>	Pointer to the tensor of size $n_1 \times n_2 \times \dots \times n_p$ that stores the result of the backward pReLU layer. This input can be an object of any class derived from <code>Tensor</code> .
<code>weightDerivatives</code>	Pointer to the tensor of size $n_k \times n_{k+1} \times \dots \times n_{k+q-1}$ that stores result $\partial E / \partial w_{i_k \dots i_{k+q-1}}$ of the backward pReLU layer. This input can be an object of any class derived from <code>Tensor</code> .

Examples

Refer to the following examples in your Intel DAAL directory:

C++: `./examples/cpp/source/neural_networks/prelu_layer_batch.cpp`

Java*: `./examples/java/source/com/intel/daal/examples/neural_networks/PreLULayerBatch.java`

Python*: `./examples/python/source/neural_networks/prelu_layer_batch.py`

Rectifier Linear Unit (ReLU) Forward Layer

The forward ReLU layer computes $f(x) = \max(0, x)$ for input argument x .

Problem Statement

Given a p -dimensional tensor X of size $n_1 \times n_2 \times \dots \times n_p$, the problem is to compute a p -dimensional tensor $Y = (y_{i_1 \dots i_p})$ of size $n_1 \times n_2 \times \dots \times n_p$, where:

$$y_{i_1 \dots i_p} = \max(0, x_{i_1 \dots i_p})$$

Batch Processing

Layer Input

The forward ReLU layer accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
<code>data</code>	Pointer to the tensor of size $n_1 \times n_2 \times \dots \times n_p$ that stores the input data for the forward ReLU layer. This input can be an object of any class derived from <code>Tensor</code> .

Layer Parameters

For common parameters of neural network layers, see [Common Parameters](#).

In addition to the common parameters, the forward ReLU layer has the following parameters:

Parameter	Default Value	Description
<code>algorithmFPType</code>	<code>float</code>	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<code>method</code>	<code>defaultDense</code>	Performance-oriented computation method, the only method supported by the layer.

Layer Output

The forward ReLU layer calculates the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result				
<code>value</code>	Pointer to the tensor of size $n_1 \times n_2 \times \dots \times n_p$ that stores the result of the forward ReLU layer. This input can be an object of any class derived from <code>Tensor</code> .				
<code>resultForBackward</code>	Collection of data needed for the backward ReLU layer.				
	<table> <tr> <th>Element ID</th><th>Element</th></tr> <tr> <td><code>auxData</code></td><td>Pointer to the tensor of size $n_1 \times n_2 \times \dots \times n_p$ that stores the input data for the forward ReLU layer. This input can be an object of any class derived from <code>Tensor</code>.</td></tr> </table>	Element ID	Element	<code>auxData</code>	Pointer to the tensor of size $n_1 \times n_2 \times \dots \times n_p$ that stores the input data for the forward ReLU layer. This input can be an object of any class derived from <code>Tensor</code> .
Element ID	Element				
<code>auxData</code>	Pointer to the tensor of size $n_1 \times n_2 \times \dots \times n_p$ that stores the input data for the forward ReLU layer. This input can be an object of any class derived from <code>Tensor</code> .				

Examples

Refer to the following examples in your Intel DAAL directory:

C++: `./examples/cpp/source/neural_networks/relu_layer_batch.cpp`

Java*: `./examples/java/source/com/intel/daal/examples/neural_networks/ReLULayerBatch.java`

Python*: `./examples/python/source/neural_networks/relu_layer_batch.py`

Rectifier Linear Unit (ReLU) Backward Layer

The rectifier linear unit (ReLU) activation layer applies the transform $f(x) = \max(0, x)$ to the input data. The backward ReLU layer computes the value $z = y * f'(x)$, where y is the input gradient computed on the prior layer and $f'(x) = \{1 \text{ if } x > 0, 0 \text{ if } x \leq 0\}$.

Problem Statement

Given p -dimensional tensors X and Y of size $n_1 \times n_2 \times \dots \times n_p$, the problem is to compute a p -dimensional tensor $Z = (z_{i_1 \dots i_p})$ of size $n_1 \times n_2 \times \dots \times n_p$, where:

$$z_{i_1 \dots i_p} = \{y_{i_1 \dots i_p} \text{ if } x_{i_1 \dots i_p} > 0, 0 \text{ if } x_{i_1 \dots i_p} \leq 0\}$$

Batch Processing

Layer Input

The backward ReLU layer accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
<code>inputGradient</code>	Pointer to the tensor of size $n_1 \times n_2 \times \dots \times n_p$ that stores the input gradient computed on the preceding layer. This input can be an object of any class derived from <code>Tensor</code> .
<code>inputFromForward</code>	Collection of data needed for the backward ReLU layer.
Element ID	Element
<code>auxData</code>	Pointer to the tensor of size $n_1 \times n_2 \times \dots \times n_p$ that stores the input data for the forward ReLU layer. This input can be an object of any class derived from <code>Tensor</code> .

Layer Parameters

For common parameters of neural network layers, see [Common Parameters](#).

In addition to the common parameters, the backward ReLU layer has the following parameters:

Parameter	Default Value	Description
<code>algorithmFPType</code>	<code>float</code>	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<code>method</code>	<code>defaultDense</code>	Performance-oriented computation method, the only method supported by the layer.

Layer Output

The backward ReLU layer calculates the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
gradient	Pointer to the tensor of size $n_1 \times n_2 \times \dots \times n_p$ that stores the result of the backward ReLU layer. This input can be an object of any class derived from <code>Tensor</code> .

Examples

Refer to the following examples in your Intel DAAL directory:

C++: `./examples/cpp/source/neural_networks/relu_layer_batch.cpp`

Java*: `./examples/java/source/com/intel/daal/examples/neural_networks/ReLULayerBatch.java`

Python*: `./examples/python/source/neural_networks/relu_layer_batch.py`

Smooth Rectifier Linear Unit (SmoothReLU) Forward Layer

The forward smooth rectifier linear unit (SmoothReLU) layer computes the function $f(x) = \log(1 + \exp(x))$ for the input argument x .

Problem Statement

Given a p -dimensional tensor X of size $n_1 \times n_2 \times \dots \times n_p$, the problem is to compute the p -dimensional tensor $Y = (y_{i_1 \dots i_p})$ of size $n_1 \times n_2 \times \dots \times n_p$ such that:

$$y_{i_1 \dots i_p} = \log(1 + \exp(x_{i_1 \dots i_p})).$$

Batch Processing

Layer Input

The forward SmoothReLU layer accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
data	Pointer to the tensor of size $n_1 \times n_2 \times \dots \times n_p$ that stores the input data for the forward SmoothReLU layer. This input can be an object of any class derived from <code>Tensor</code> .

Layer Parameters

For common parameters of neural network layers, see [Common Parameters](#).

In addition to the common parameters, the forward SmoothReLU layer has the following parameters:

Parameter	Default Value	Description
<code>algorithmFPTYPE</code>	<code>float</code>	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<code>method</code>	<code>defaultDense</code>	Performance-oriented computation method, the only method supported by the layer.

Layer Output

The forward SmoothReLU layer calculates the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result				
value	Pointer to the tensor of size $n_1 \times n_2 \times \dots \times n_p$ that stores the result of the forward SmoothReLU layer. This input can be an object of any class derived from <code>Tensor</code> .				
resultForBackward	Collection of data needed for the backward SmoothReLU layer.				
	<table> <tr> <th>Element ID</th><th>Element</th></tr> <tr> <td>auxData</td><td>Pointer to the tensor of size $n_1 \times n_2 \times \dots \times n_p$ that stores the input data for the forward SmoothReLU layer. This input can be an object of any class derived from <code>Tensor</code>.</td></tr> </table>	Element ID	Element	auxData	Pointer to the tensor of size $n_1 \times n_2 \times \dots \times n_p$ that stores the input data for the forward SmoothReLU layer. This input can be an object of any class derived from <code>Tensor</code> .
Element ID	Element				
auxData	Pointer to the tensor of size $n_1 \times n_2 \times \dots \times n_p$ that stores the input data for the forward SmoothReLU layer. This input can be an object of any class derived from <code>Tensor</code> .				

Examples

Refer to the following examples in your Intel DAAL directory:

C++: `./examples/cpp/source/neural_networks/smoothrelu_layer_batch.cpp`

Java*: `./examples/java/source/com/intel/daal/examples/neural_networks/SmoothReLULayerBatch.java`

Python*: `./examples/python/source/neural_networks/smoothrelu_layer_batch.py`

Smooth Rectifier Linear Unit (SmoothReLU) Backward Layer

The smooth rectifier linear unit (SmoothReLU) activation layer applies the transform $f(x) = \log(1 + \exp(x))$ to the input data. The backward SmoothReLU layer computes the values $z = y * f'(x)$, where y is the input gradient computed on the preceding layer and

$$f'(x) = \frac{1}{1 + \exp(-x)}.$$

Problem Statement

Given p -dimensional tensors X and Y of size $n_1 \times n_2 \times \dots \times n_p$, the problem is to compute the p -dimensional tensor $Z = (z_{i_1 \dots i_p})$ of size $n_1 \times n_2 \times \dots \times n_p$ such that:

$$z_{i_1 \dots i_p} = \frac{y_{i_1 \dots i_p}}{1 + \exp(-x_{i_1 \dots i_p})}.$$

Batch Processing

Layer Input

The backward SmoothReLU layer accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input				
inputGradient	Pointer to the tensor of size $n_1 \times n_2 \times \dots \times n_p$ that stores the input gradient computed on the preceding layer. This input can be an object of any class derived from <code>Tensor</code> .				
inputFromForward	Collection of data needed for the backward SmoothReLU layer.				
	<table> <tr> <th>Element ID</th><th>Element</th></tr> <tr> <td>auxData</td><td>Pointer to the tensor of size $n_1 \times n_2 \times \dots \times n_p$ that stores the input data for the forward SmoothReLU layer. This input can be an object of any class derived from <code>Tensor</code>.</td></tr> </table>	Element ID	Element	auxData	Pointer to the tensor of size $n_1 \times n_2 \times \dots \times n_p$ that stores the input data for the forward SmoothReLU layer. This input can be an object of any class derived from <code>Tensor</code> .
Element ID	Element				
auxData	Pointer to the tensor of size $n_1 \times n_2 \times \dots \times n_p$ that stores the input data for the forward SmoothReLU layer. This input can be an object of any class derived from <code>Tensor</code> .				

Layer Parameters

For common parameters of neural network layers, see [Common Parameters](#).

In addition to the common parameters, the backward SmoothReLU layer has the following parameters:

Parameter	Default Value	Description
<code>algorithmFPType</code>	<code>float</code>	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<code>method</code>	<code>defaultDense</code>	Performance-oriented computation method, the only method supported by the layer.

Layer Output

The backward SmoothReLU layer calculates the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
gradient	Pointer to the tensor of size $n_1 \times n_2 \times \dots \times n_p$ that stores the result of the backward SmoothReLU layer. This input can be an object of any class derived from <code>Tensor</code> .

Examples

Refer to the following examples in your Intel DAAL directory:

C++: `./examples/cpp/source/neural_networks/smoothrelu_layer_batch.cpp`

Java*: `./examples/java/source/com/intel/daal/examples/neural_networks/SmoothReLULayerBatch.java`

Python*: `./examples/python/source/neural_networks/smoothrelu_layer_batch.py`

Hyperbolic Tangent Forward Layer

The forward hyperbolic tangent layer computes the function

$$\tanh(x) = \frac{\sinh(x)}{\cosh(x)}$$

for the input argument x .

Problem Statement

Given a p -dimensional tensor X of size $n_1 \times n_2 \times \dots \times n_p$, the problem is to compute the p -dimensional tensor $Y = (y_{i_1 \dots i_p})$ of size $n_1 \times n_2 \times \dots \times n_p$ such that:

$$y_{i_1 \dots i_p} = \frac{\sinh(x_{i_1 \dots i_p})}{\cosh(x_{i_1 \dots i_p})}.$$

Batch Processing

Layer Input

The forward hyperbolic tangent layer accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
<code>data</code>	Pointer to the tensor of size $n_1 \times n_2 \times \dots \times n_p$ that stores the input data for the forward hyperbolic tangent layer. This input can be an object of any class derived from <code>Tensor</code> .

Layer Parameters

For common parameters of neural network layers, see [Common Parameters](#).

In addition to the common parameters, the forward hyperbolic tangent layer has the following parameters:

Parameter	Default Value	Description
<code>algorithmFPType</code>	<code>float</code>	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<code>method</code>	<code>defaultDense</code>	Performance-oriented computation method, the only method supported by the layer.

Layer Output

The forward hyperbolic tangent layer calculates the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result				
<code>value</code>	Pointer to the tensor of size $n_1 \times n_2 \times \dots \times n_p$ that stores the result of the forward hyperbolic tangent layer. This input can be an object of any class derived from <code>Tensor</code> .				
<code>resultForBackward</code>	Collection of data needed for the backward hyperbolic tangent layer.				
	<table> <tr> <th>Element ID</th><th>Element</th></tr> <tr> <td><code>auxValue</code></td><td>Pointer to the tensor of size $n_1 \times n_2 \times \dots \times n_p$ that stores the result of the forward hyperbolic tangent layer. This input can be an object of any class derived from <code>Tensor</code>.</td></tr> </table>	Element ID	Element	<code>auxValue</code>	Pointer to the tensor of size $n_1 \times n_2 \times \dots \times n_p$ that stores the result of the forward hyperbolic tangent layer. This input can be an object of any class derived from <code>Tensor</code> .
Element ID	Element				
<code>auxValue</code>	Pointer to the tensor of size $n_1 \times n_2 \times \dots \times n_p$ that stores the result of the forward hyperbolic tangent layer. This input can be an object of any class derived from <code>Tensor</code> .				

Examples

Refer to the following examples in your Intel DAAL directory:

C++: `./examples/cpp/source/neural_networks/tanh_layer_batch.cpp`

Java*: `./examples/java/source/com/intel/daal/examples/neural_networks/TanhLayerBatch.java`

Python*: `./examples/python/source/neural_networks/tanh_layer_batch.py`

Hyperbolic Tangent Backward Layer

The hyperbolic tangent activation layer applies the transform

$$f(x) = \frac{\sinh(x)}{\cosh(x)}$$

to the input data. The backward hyperbolic tangent layer computes the values $z = y * f'(x)$, where y is the input gradient computed on the preceding layer and

$$f'(x) = 1 - \left(\frac{\sinh(x)}{\cosh(x)} \right)^2 = 1 - f^2(x).$$

Problem Statement

Given p -dimensional tensors X and Y of size $n_1 \times n_2 \times \dots \times n_p$, the problem is to compute the p -dimensional tensor $Z = (z_{i_1 \dots i_p})$ of size $n_1 \times n_2 \times \dots \times n_p$ such that:

$$z_{i_1 \dots i_p} = y_{i_1 \dots i_p} * \left(1 - f^2(x_{i_1 \dots i_p}) \right).$$

Batch Processing

Layer Input

The backward hyperbolic tangent layer accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input				
inputGradient	Pointer to the tensor of size $n_1 \times n_2 \times \dots \times n_p$ that stores the input gradient computed on the preceding layer. This input can be an object of any class derived from <code>Tensor</code> .				
inputFromForward	Collection of data needed for the backward hyperbolic tangent layer.				
	<table> <tr> <th>Element ID</th><th>Element</th></tr> <tr> <td>auxValue</td><td>Pointer to the tensor of size $n_1 \times n_2 \times \dots \times n_p$ that stores the result of the forward hyperbolic tangent layer. This input can be an object of any class derived from <code>Tensor</code>.</td></tr> </table>	Element ID	Element	auxValue	Pointer to the tensor of size $n_1 \times n_2 \times \dots \times n_p$ that stores the result of the forward hyperbolic tangent layer. This input can be an object of any class derived from <code>Tensor</code> .
Element ID	Element				
auxValue	Pointer to the tensor of size $n_1 \times n_2 \times \dots \times n_p$ that stores the result of the forward hyperbolic tangent layer. This input can be an object of any class derived from <code>Tensor</code> .				

Layer Parameters

For common parameters of neural network layers, see [Common Parameters](#).

In addition to the common parameters, the backward hyperbolic tangent layer has the following parameters:

Parameter	Default Value	Description
<code>algorithmFPTType</code>	<code>float</code>	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<code>method</code>	<code>defaultDense</code>	Performance-oriented computation method, the only method supported by the layer.

Layer Output

The backward hyperbolic tangent layer calculates the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
<code>gradient</code>	Pointer to the tensor of size $n_1 \times n_2 \times \dots \times n_p$ that stores the result of the backward hyperbolic tangent layer. This input can be an object of any class derived from <code>Tensor</code> .

Examples

Refer to the following examples in your Intel DAAL directory:

C++: `./examples/cpp/source/neural_networks/tanh_layer_batch.cpp`

Java*: `./examples/java/source/com/intel/daal/examples/neural_networks/TanhLayerBatch.java`

Python*: `./examples/python/source/neural_networks/tanh_layer_batch.py`

Batch Normalization Forward Layer

The forward batch normalization layer [Ioffe2015] normalizes $x_{i_1 \dots i_p}$ from the input $X \in R^{n_1 \times n_2 \times \dots \times n_p}$ for the dimension $k \in \{1, \dots, p\}$ and then scales and shifts the result of the normalization using the provided weights and biases as follows:

$$y = \omega \frac{x - \mu^{(k)}}{\sigma^{(k)}} + \beta,$$

where the following characteristics are computed for the input X :

- means

$$\mu^{(k)} = E[x]^{(k)}$$

- standard deviation

$$\sigma^{(k)} = \sqrt{s^{2(k)} + \varepsilon}$$

with variance

$$s^{2(k)} = Var[x]^{(k)}$$

- a constant ε to improve the numerical stability

The weights and biases are learned, as well as the rest model parameters.

Problem Statement

Given a p -dimensional tensor $X \in R^{n_1 \times n_2 \times \dots \times n_p}$, the problem is to compute the p -dimensional tensor $Y \in R^{n_1 \times n_2 \times \dots \times n_p}$:

$$y_{j_1 \dots j_{k-1} j_{k+1} \dots j_p} = y \left(x_{j_1 \dots j_{k-1} j_{k+1} \dots j_p} \right) = \omega_j^{(k)} \frac{x_{j_1 \dots j_{k-1} j_{k+1} \dots j_p} - \mu_j^{(k)}}{\sigma_j^{(k)}} + \beta_j^{(k)},$$

$k \in \{1, \dots, p\}, j = 1, \dots, n_k,$

where:

- $m_k = n_1 \cdot \dots \cdot n_{k-1} \cdot n_{k+1} \cdot \dots \cdot n_p$
- mean

$$\mu^{(k)} \in \mathbb{R}^{n_k},$$

$$\mu_j^{(k)} = \frac{1}{m_k} \sum_{j_1=1}^{n_1} \dots \sum_{j_{k-1}=1}^{n_{k-1}} \sum_{j_{k+1}=1}^{n_{k+1}} \dots \sum_{j_p=1}^{n_p} x_{j_1 \dots j_{k-1} j j_{k+1} \dots j_p}$$

- variance

$$s^{2(k)} \in \mathbb{R}^{n_k},$$

$$s^{2(k)} = \frac{1}{m_k - 1} \sum_{j_1=1}^{n_1} \dots \sum_{j_{k-1}=1}^{n_{k-1}} \sum_{j_{k+1}=1}^{n_{k+1}} \dots \sum_{j_p=1}^{n_p} \left(x_{j_1 \dots j_{k-1} j j_{k+1} \dots j_p} - \mu_j^{(k)} \right)^2$$

- standard deviation

$$\sigma^{(k)} = \sqrt{s^{2(k)} + \varepsilon}$$

- weights

$$\omega^{(k)} \in \mathbb{R}^{n_k}$$

- biases

$$\beta^{(k)} \in \mathbb{R}^{n_k}$$

At the model training stage, along with the normalizing, the layer computes the population mean and variance using the exponential moving average with smoothing factor $\alpha \in [0,1]$ applied to the mini-batch means and variances.

Batch Processing

Layer Input

The forward batch normalization layer accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
data	Tensor of size $n_1 \times n_2 \times \dots \times n_p$ that stores the input data for the forward batch normalization layer. This input can be an object of any class derived from <code>Tensor</code> .
weights	One-dimensional tensor of size n_k that stores weights for scaling $\omega^{(k)}$. This input can be an object of any class derived from <code>Tensor</code> .
biases	One-dimensional tensor of size n_k that stores biases for shifting the scaled data $\beta^{(k)}$. This input can be an object of any class derived from <code>Tensor</code> .

Input ID	Input
populationMean	One-dimensional tensor of size n_k that stores population mean μ computed in the previous stage. This input can be an object of any class derived from <code>Tensor</code> .
populationVariance	One-dimensional tensor of size n_k that stores population variance s^2 computed in the previous stage. This input can be an object of any class derived from <code>Tensor</code> .

Layer Parameters

For common parameters of neural network layers, see [Common Parameters](#).

In addition to the common parameters, the forward batch normalization layer has the following parameters:

Parameter	Default Value	Description
<i>algorithmFPType</i>	float	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<i>method</i>	defaultDense	Performance-oriented computation method, the only method supported by the layer.
<i>alpha</i>	0.01	Smoothing factor of the exponential moving average used to compute the population mean and variance.
<i>epsilon</i>	0.00001	Constant added to the mini-batch variance for numerical stability.
<i>dimension</i>	0	Index of dimension k for which normalization is performed.

Layer Output

The forward batch normalization layer calculates the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result				
value	Tensor of size $n_1 \times n_2 \times \dots \times n_p$ that stores the result of the forward batch normalization layer. This input can be an object of any class derived from <code>Tensor</code> .				
resultForBackward	Collection of data needed for the backward batch normalization layer.				
	<table> <tr> <th>Element ID</th><th>Element</th></tr> <tr> <td>auxData</td><td>Tensor of size $n_1 \times n_2 \times \dots \times n_p$ that stores the input data for the forward batch normalization layer. This input can be an object of any class derived from <code>Tensor</code>.</td></tr> </table>	Element ID	Element	auxData	Tensor of size $n_1 \times n_2 \times \dots \times n_p$ that stores the input data for the forward batch normalization layer. This input can be an object of any class derived from <code>Tensor</code> .
Element ID	Element				
auxData	Tensor of size $n_1 \times n_2 \times \dots \times n_p$ that stores the input data for the forward batch normalization layer. This input can be an object of any class derived from <code>Tensor</code> .				

Result ID	Result
	<div>auxWeights</div> <div>One-dimensional tensor of size n_k that stores weights for scaling $\omega^{(k)}$. This input can be an object of any class derived from <code>Tensor</code>.</div>
	<div>auxMean</div> <div>One-dimensional tensor of size n_k that stores mini-batch mean μ^k. This input can be an object of any class derived from <code>Tensor</code>.</div>
	<div>auxStandardDeviation</div> <div>One-dimensional tensor of size n_k that stores mini-batch standard deviation $\sigma^{(k)}$. This input can be an object of any class derived from <code>Tensor</code>.</div>
	<div>auxPopulationMean</div> <div>One-dimensional tensor of size n_k that stores the resulting population mean μ. This input can be an object of any class derived from <code>Tensor</code>.</div>
	<div>auxPopulationVariance</div> <div>One-dimensional tensor of size n_k that stores the resulting population variance s^2. This input can be an object of any class derived from <code>Tensor</code>.</div>

Examples

Refer to the following examples in your Intel DAAL directory:

C++: `./examples/cpp/source/neural_networks/batch_normalization_layer_batch.cpp`

Java*: `./examples/java/source/com/intel/daal/examples/neural_networks/BatchNormalizationLayerBatch.java`

Python*: `./examples/python/source/neural_networks/batch_normalization_layer_batch.py`

Batch Normalization Backward Layer

The forward batch normalization layer normalizes $x_{i_1 \dots i_p}$ from the input $X \in R^{n_1 \times n_2 \times \dots \times n_p}$ for the dimension $k \in \{1, \dots, p\}$ and then scales and shifts the result of the normalization. For more details, see [Forward Batch Normalization Layer](#). The backward batch normalization layer [Ioffe2015] computes the values for the dimension $k \in \{1, \dots, p\}$:

$$\frac{\partial E}{\partial \omega_j^{(k)}} = \sum_{i=1}^{m_k} \left(g_i \frac{(x_i - \mu^{(k)})}{\sigma^{(k)}} \right),$$

$$\frac{\partial E}{\partial \beta_j^{(k)}} = \sum_{i=1}^{m_k} g_i,$$

where

- g is the gradient of the preceding layer

- E is the objective function used at the training stage.
- objective function used at the training stage.

$$m_k = n_1 \cdot \dots \cdot n_{k-1} \cdot n_{k+1} \cdot \dots \cdot n_p$$

- weights

$$\omega^{(k)} \in \mathbb{R}^{n_k}$$

- biases

$$\beta^{(k)} \in \mathbb{R}^{n_k}$$

- mean

$$\mu^{(k)} \in \mathbb{R}^{n_k},$$

$$\mu_j^{(k)} = \frac{1}{m_k} \sum_{j_1=1}^{n_1} \dots \sum_{j_{k-1}=1}^{n_{k-1}} \sum_{j_{k+1}=1}^{n_{k+1}} \dots \sum_{j_p=1}^{n_p} x_{j_1 \dots j_{k-1} j j_{k+1} \dots j_p}, j = 1, \dots, n_k$$

- variance

$$s^{2(k)} \in \mathbb{R}^{n_k},$$

$$s^{2(k)} = \frac{1}{m_k - 1} \sum_{j_1=1}^{n_1} \dots \sum_{j_{k-1}=1}^{n_{k-1}} \sum_{j_{k+1}=1}^{n_{k+1}} \dots \sum_{j_p=1}^{n_p} \left(x_{j_1 \dots j_{k-1} j j_{k+1} \dots j_p} - \mu_j^{(k)} \right)^2$$

- standard deviation

$$\sigma^{(k)} = \sqrt{s^{2(k)} + \varepsilon}$$

$$\sum_{j=1}^{m_k} g_i \stackrel{def}{=} \sum_{j_1=1}^{n_1} \dots \sum_{j_{k-1}=1}^{n_{k-1}} \sum_{j_{k+1}=1}^{n_{k+1}} \dots \sum_{j_p=1}^{n_p} g_{j_1 \dots j_{k-1} i j_{k+1} \dots j_p}$$

$$\sum_{j=1}^{m_k} \left(g_i \frac{(x_i - \mu^{(k)})}{\sigma^{(k)}} \right) \stackrel{def}{=} \sum_{j_1=1}^{n_1} \dots \sum_{j_{k-1}=1}^{n_{k-1}} \sum_{j_{k+1}=1}^{n_{k+1}} \dots \sum_{j_p=1}^{n_p} g_{j_1 \dots j_{k-1} i j_{k+1} \dots j_p} \frac{(x_{j_1 \dots j_{k-1} i j_{k+1} \dots j_p} - \mu_i^{(k)})}{\sigma_i^{(k)}}$$

Problem Statement

Given p -dimensional tensors:

- $G \in \mathbb{R}^{n_1 \times n_2 \times \dots \times n_p}$ - the gradient computed on the preceding layer
- $Y \in \mathbb{R}^{n_1 \times n_2 \times \dots \times n_p}$ - the output of the forward batch normalization layer

The problem is to compute the p -dimensional tensor $Z \in \mathbb{R}^{n_1 \times n_2 \times \dots \times n_p}$ such that:

$$j_1 \dots j_{k-1} j j_{k+1} \dots j_p = \frac{\omega_j^{(k)}}{\sigma_j^{(k)}} \left(g_{j_1 \dots j_{k-1} j j_{k+1} \dots j_p} - \frac{1}{m_k} \frac{\partial E}{\partial \beta_j^{(k)}} - \frac{1}{m_k - 1} \frac{x_{j_1 \dots j_{k-1} j j_{k+1} \dots j_p} - \mu_j^{(k)}}{\sigma_j^{(k)}} \frac{\partial E}{\partial \omega_j^{(k)}} \right)$$

for $j = 1, \dots, n_k$, where:

-

$$\frac{\partial E}{\partial \omega^{(k)}} \in \mathbb{R}^{n_k},$$

$$\frac{\partial E}{\partial \omega_j^{(k)}} = \sum_{j_1=1}^{n_1} \dots \sum_{j_{k-1}=1}^{n_{k-1}} \sum_{j_{k+1}=1}^{n_{k+1}} \dots \sum_{j_p=1}^{n_p} g_{j_1 \dots j_{k-1} j j_{k+1} \dots j_p} \frac{(x_{j_1 \dots j_{k-1} j j_{k+1} \dots j_p} - \mu_j^{(k)})}{\sigma_j^{(k)}}$$

•

$$\frac{\partial E}{\partial \beta^{(k)}} \in \mathbb{R}^{n_k},$$

$$\frac{\partial E}{\partial \beta_j^{(k)}} = \sum_{j_1=1}^{n_1} \cdots \sum_{j_{k-1}=1}^{n_{k-1}} \sum_{j_{k+1}=1}^{n_{k+1}} \cdots \sum_{j_p=1}^{n_p} g_{j_1 \cdots j_{k-1} j_{k+1} \cdots j_p}$$

Batch Processing

Layer Input

The backward batch normalization layer accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input	
inputGradient	Tensor of size $n_1 \times n_2 \times \dots \times n_p$ that stores the input gradient computed on the preceding layer. This input can be an object of any class derived from <code>Tensor</code> .	
inputFromForward	Collection of data needed for the backward batch normalization layer.	
	Element ID	Element
	auxData	Tensor of size $n_1 \times n_2 \times \dots \times n_p$ that stores the input data for the forward batch normalization layer. This input can be an object of any class derived from <code>Tensor</code> .
	auxWeights	One-dimensional tensor of size n_k that stores weights for scaling $\omega^{(k)}$ from the forward batch normalization layer. This input can be an object of any class derived from <code>Tensor</code> .
	auxMean	One-dimensional tensor of size n_k that stores the mini-batch mean computed in the forward step. This input can be an object of any class derived from <code>Tensor</code> .
	auxStandardDeviation	One-dimensional tensor of size n_k that stores the population standard deviation computed in the forward step. This input can be an object of any class derived from <code>Tensor</code> .
	auxPopulationMean	One-dimensional tensor of size n_k that stores the population mean computed in the forward step. This input can be an object of any class derived from <code>Tensor</code> .

Input ID	Input
	<div>auxPopulationVariance</div> <div>One-dimensional tensor of size n_k that stores the population variance computed in the forward step. This input can be an object of any class derived from <code>Tensor</code>.</div>

Layer Parameters

For common parameters of neural network layers, see [Common Parameters](#).

In addition to the common parameters, the backward batch normalization layer has the following parameters:

Parameter	Default Value	Description
<code>algorithmFPType</code>	<code>float</code>	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<code>method</code>	<code>defaultDense</code>	Performance-oriented computation method, the only method supported by the layer.
<code>epsilon</code>	<code>0.00001</code>	Constant added to the mini-batch variance for numerical stability.
<code>dimension</code>	<code>0</code>	Index of dimension k for which normalization is performed.
<code>propagateGradient</code>	<code>false</code>	Flag that specifies whether the backward layer propagates the gradient.

Layer Output

The backward batch normalization layer calculates the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
<code>gradient</code>	Tensor of size $n_1 \times n_2 \times \dots \times n_p$ that stores result z of the backward batch normalization layer. This input can be an object of any class derived from <code>Tensor</code> .
<code>weightsDerivatives</code>	One-dimensional tensor of size n_k that stores result $\partial E / \partial \omega^{(k)}$ of the backward batch normalization layer. This input can be an object of any class derived from <code>Tensor</code> .
<code>biasesDerivatives</code>	One-dimensional tensor of size n_k that stores result $\partial E / \partial \beta^{(k)}$ of the backward batch normalization layer. This input can be an object of any class derived from <code>Tensor</code> .

Examples

Refer to the following examples in your Intel DAAL directory:

C++: `./examples/cpp/source/neural_networks/batch_normalization_layer_batch.cpp`

Java*: `./examples/java/source/com/intel/daal/examples/neural_networks/BatchNormalizationLayerBatch.java`

Python*: `./examples/python/source/neural_networks/batch_normalization_layer_batch.py`

Local Response Normalization Forward Layer

For any $x_{i_1 \dots i_p}$ from the input tensor $X \in \mathbb{R}^{n_1 \times n_2 \times \dots \times n_p}$ and a dimension $k \in \{1, \dots, p\}$ of size n_k , the forward local response normalization layer computes the p -dimensional tensor $Y \in \mathbb{R}^{n_1 \times n_2 \times \dots \times n_p}$ such that:

$$y_{i_1 \dots i_p} = \frac{x_{i_1 \dots i_p}}{(S_k(i_k))^\beta},$$

where

- $S_k(i) = \kappa + \alpha \sum_{j \in D_k(i)} \left(x_{i_1 \dots i_{k-1} j i_{k+1} \dots i_p} \right)^2$
- $D_k(i) = \left\{ j \in \{1, \dots, n_k\} : \max\left(1, i - \frac{n}{2}\right) \leq j \leq \min\left(n_k, i + \frac{n}{2}\right) \right\}$
- $\alpha, \beta, \kappa \in \mathbb{R}$
- n is a positive integer number

See [Krizh2012] for an exact definition of local response normalization.

Problem Statement

Given a p -dimensional tensor $X \in \mathbb{R}^{n_1 \times n_2 \times \dots \times n_p}$ and scalars α, β, κ , and n , the problem is to compute the p -dimensional tensor $Y \in \mathbb{R}^{n_1 \times n_2 \times \dots \times n_p}$.

Batch Processing

Layer Input

The forward local response normalization layer accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
data	Pointer to the tensor of size $n_1 \times n_2 \times \dots \times n_p$ that stores the input data for the forward local response normalization layer. This input can be an object of any class derived from <code>Tensor</code> .

Layer Parameters

For common parameters of neural network layers, see [Common Parameters](#).

In addition to the common parameters, the forward local response normalization layer has the following parameters:

Parameter	Default Value	Description
<code>algorithmFPType</code>	float	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<code>method</code>	<code>defaultDense</code>	Computation method used by the algorithm. The only method supported by the layer so far is <code>acrossDimension</code> .
<code>dimension</code>	1	Numeric table of size 1 x 1 with the index of type <code>size_t</code> to calculate local response normalization.

Parameter	Default Value	Description
<i>kappa</i>	2	Value of layer hyper-parameter κ .
<i>alpha</i>	1.0e-04	Value of layer hyper-parameter α .
<i>beta</i>	0.75	Value of layer hyper-parameter β .
<i>nAdjust</i>	5	Value of layer hyper-parameter n .

Layer Output

The forward local response normalization layer calculates the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result						
value	Pointer to the tensor of size $n_1 \times n_2 \times \dots \times n_p$ that stores the result of the forward local response normalization layer. This input can be an object of any class derived from <code>Tensor</code> .						
resultForBackward	Collection of input data needed for the backward local response normalization layer. This collection can contain objects of any class derived from <code>Tensor</code> .						
	<table> <tr> <th>Element ID</th><th>Element</th></tr> <tr> <td>auxData</td><td>Pointer to the tensor of size $n_1 \times n_2 \times \dots \times n_p$ that stores the input data for the forward local response normalization layer. This input can be an object of any class derived from <code>Tensor</code>.</td></tr> <tr> <td>auxSmBeta</td><td>Pointer to the tensor of size $n_1 \times n_2 \times \dots \times n_p$ that stores the value of $(s_k * (i_k))^{-\beta}$. This input can be an object of any class derived from <code>Tensor</code>.</td></tr> </table>	Element ID	Element	auxData	Pointer to the tensor of size $n_1 \times n_2 \times \dots \times n_p$ that stores the input data for the forward local response normalization layer. This input can be an object of any class derived from <code>Tensor</code> .	auxSmBeta	Pointer to the tensor of size $n_1 \times n_2 \times \dots \times n_p$ that stores the value of $(s_k * (i_k))^{-\beta}$. This input can be an object of any class derived from <code>Tensor</code> .
Element ID	Element						
auxData	Pointer to the tensor of size $n_1 \times n_2 \times \dots \times n_p$ that stores the input data for the forward local response normalization layer. This input can be an object of any class derived from <code>Tensor</code> .						
auxSmBeta	Pointer to the tensor of size $n_1 \times n_2 \times \dots \times n_p$ that stores the value of $(s_k * (i_k))^{-\beta}$. This input can be an object of any class derived from <code>Tensor</code> .						

Examples

Refer to the following examples in your Intel DAAL directory:

C++: `./examples/cpp/source/neural_networks/lrn_layer_batch.cpp`

Java*: `./examples/java/source/com/intel/daal/examples/neural_networks/LRNLayerBatch.java`

Python*: `./examples/python/source/neural_networks/lrn_layer_batch.py`

Local Response Normalization Backward Layer

For a given dimension $k \in \{1, \dots, p\}$ of size n_k , the forward local response normalization layer normalizes the input tensor $X \in \mathbb{R}^{n_1 \times n_2 \times \dots \times n_p}$. For more details and notations, see [Forward Local Response Normalization Layer](#).

For a dimension $k \in \{1, \dots, p\}$ of size n_k , the backward local response normalization layer computes the value:

$$z_{i_1 \dots i_p} = g_{i_1 \dots i_p} S_k(i_k)^{-\beta} - 2\alpha\beta x_{i_1 \dots i_p} \sum_{d: i_k \in D_k(d)} g_{i_1 \dots i_{k-1} d i_{k+1} \dots i_p} S_k(d)^{-\beta-1} x_{i_1 \dots i_{k-1} d i_{k+1} \dots i_p},$$

where:

- $g_{i_1 \dots i_p}$ is the input gradient computed on the preceding layer
- $S_k(i) = \kappa + \alpha \sum_{j \in D_k(i)} \left(x_{i_1 \dots i_{k-1} j i_{k+1} \dots i_p} \right)^2$
- $D_k(i) = \left\{ j \in \{1, \dots, n_k\} : \max\left(1, i - \frac{n}{2}\right) \leq j \leq \min\left(n_k, i + \frac{n}{2}\right) \right\}$
- $\alpha, \beta, \kappa \in \mathbb{R}$
- n is a positive integer number

See [Krizh2012] for an exact definition of local response normalization.

Problem Statement

Given p -dimensional tensors:

- $X \in \mathbb{R}^{n_1 \times n_2 \times \dots \times n_p}$ of size $n_1 \times n_2 \times \dots \times n_p$
- $G \in \mathbb{R}^{n_1 \times n_2 \times \dots \times n_p}$ - the gradient computed on the preceding layer

The problem is to compute the p -dimensional tensor $Z = (z_{i_1 \dots i_p}) \in \mathbb{R}^{n_1 \times n_2 \times \dots \times n_p}$.

Batch Processing

Layer Input

The backward local response normalization layer accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
inputGradient	Pointer to the tensor of size $n_1 \times n_2 \times \dots \times n_p$ that stores the input gradient computed on the preceding layer. This input can be an object of any class derived from <code>Tensor</code> .
inputFromForward	Collection of input data needed for the backward local response normalization layer. This collection can contain objects of any class derived from <code>Tensor</code> .
Element ID	Element
auxData	Pointer to the tensor of size $n_1 \times n_2 \times \dots \times n_p$ that stores the input data for the forward local response normalization layer. This input can be an object of any class derived from <code>Tensor</code> .
auxSmBeta	Pointer to the tensor of size $n_1 \times n_2 \times \dots \times n_p$ that stores the value of $(s_k * (i_k))^{-\beta}$. This input can be an object of any class derived from <code>Tensor</code> .

Layer Parameters

For common parameters of neural network layers, see [Common Parameters](#).

In addition to the common parameters, the backward local response normalization layer has the following parameters:

Parameter	Default Value	Description
<i>algorithmFPTType</i>	float	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<i>method</i>	defaultDense	Computation method used by the algorithm. The only method supported by the layer so far is <code>acrossDimension</code> .
<i>dimension</i>	1	Numeric table of size 1 x 1 with the dimension index of type <i>size_t</i> to calculate local response normalization backpropagation.
<i>kappa</i>	2	Value of layer hyper-parameter κ .
<i>alpha</i>	1.0e-04	Value of layer hyper-parameter α .
<i>beta</i>	0.75	Value of layer hyper-parameter β .
<i>nAdjust</i>	5	Value of layer hyper-parameter n .

Layer Output

The backward local response normalization layer calculates the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
gradient	Pointer to the tensor of size $n_1 \times n_2 \times \dots \times n_p$ that stores the result of the backward local response normalization layer. This input can be an object of any class derived from <code>Tensor</code> .

Examples

Refer to the following examples in your Intel DAAL directory:

C++: `./examples/cpp/source/neural_networks/lrn_layer_batch.cpp`

Java*: `./examples/java/source/com/intel/daal/examples/neural_networks/LRNLayerBatch.java`

Python*: `./examples/python/source/neural_networks/lrn_layer_batch.py`

Local Contrast Normalization Forward Layer

Given a p -dimensional tensor $X \in R^{n_1 \times n_2 \times \dots \times n_p}$, two-dimensional tensor $K \in R^{m_1 \times m_2}$, dimensions k_1 of size m_1 and k_2 of size m_2 , and dimension f different from k_1 and k_2 , the layer computes the p -dimensional tensor $Y \in R^{n_1 \times n_2 \times \dots \times n_p}$ such that:

$$y_{i_1 \dots i_p} = \frac{v_{i_1 \dots i_p}}{m_{i_1 \dots i_p}}.$$

See [Jarrett2009] for an exact definition of local contrast normalization.

The library supports four-dimensional input tensors $X \in R^{n_1 \times n_2 \times n_3 \times n_4}$.

Problem Statement

Without loss of generality let's assume that forward local contrast normalization is applied to the last two dimensions.

The problem is to compute the tensor Y depending on whether the dimension f is set:

- Dimension f is set; let it be n_2 :

$$y_{nqij} = \frac{v_{nqij}}{m_{nij}},$$

$$v_{nqij} = x_{nqij} - \sum_{r=0}^{n_2-1} \sum_{a=-m_1/2}^{m_1/2} \sum_{b=-m_2/2}^{m_2/2} k_{ab} \cdot x_{nr(i+a)(j+b)},$$

$$m_{nij} = \max(c_n, \sigma_{nij}),$$

$$\sigma_{nij} = \left(\sum_{r=0}^{n_2-1} \sum_{a=0}^{m_1-1} \sum_{b=0}^{m_2-1} k_{ab} \cdot v_{nr\left(i-\frac{m_1}{2}+a\right)\left(j-\frac{m_2}{2}+b\right)}^2 \right)^{\frac{1}{2}},$$

$$c_n = \frac{1}{n_3 n_4} \sum_{i=0}^{n_3-1} \sum_{j=0}^{n_4-1} \sigma_{nij},$$

where elements of the weighting window are normalized by the library through dimension f to meet the condition:

$$\sum_{r=0}^{n_2} \sum_{a=0}^{m_1-1} \sum_{b=0}^{m_2-1} k_{ab} = 1.$$

- Dimension f is not set:

$$y_{nqij} = \frac{v_{nqij}}{m_{nqij}},$$

$$v_{nqij} = x_{nqij} - \sum_{a=0}^{m_1-1} \sum_{b=0}^{m_2-1} k_{ab} \cdot x_{nq\left(i-\frac{m_1}{2}+a\right)\left(j-\frac{m_2}{2}+b\right)},$$

$$m_{nqij} = \max(c_{nq}, \sigma_{nqij}),$$

$$\sigma_{nqij} = \left(\sum_{a=0}^{m_1-1} \sum_{b=0}^{m_2-1} k_{ab} \cdot v_{nq\left(i-\frac{m_1}{2}+a\right)\left(j-\frac{m_2}{2}+b\right)}^2 \right)^{\frac{1}{2}},$$

$$c_{nq} = \frac{1}{n_3 n_4} \sum_{i=0}^{n_3-1} \sum_{j=0}^{n_4-1} \sigma_{nqij},$$

where the weighting window meets the condition

$$\sum_{a=0}^{m_1-1} \sum_{b=0}^{m_2-1} k_{ab} = 1.$$

Batch Processing

Layer Input

The forward local contrast normalization layer accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
<code>data</code>	Pointer to tensor X of size $n_1 \times n_2 \times n_3 \times n_4$ that stores the input data for the forward local contrast normalization layer. This input can be an object of any class derived from <code>Tensor</code> .

Layer Parameters

For common parameters of neural network layers, see [Common Parameters](#).

In addition to the common parameters, the forward local contrast normalization layer has the following parameters:

Parameter	Default Value	Description
<code>algorithmFPType</code>	<code>float</code>	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<code>method</code>	<code>defaultDense</code>	Default computation method used by the algorithm, the only method supported by the layer.
<code>kernel</code>	<code>HomogenTensor<float></code> of size 5×5 with values 0.04	Tensor with sizes $m_1 \times m_2$ of the two-dimensional kernel. Only kernels with odd dimensions are currently supported.
<code>indices</code>	<code>indices(2,3)</code>	Data structure representing dimensions k_1 and k_2 for kernels.
<code>sumDimension</code>	<code>HomogenNumericTable<float></code> of size 1×1 with value 1	Numeric table of size 1×1 that stores dimension f . If it is <code>NULL</code> , there is no summation over this dimension.

Layer Output

The forward local contrast normalization layer calculates the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
<code>value</code>	Pointer to tensor Y of size $n_1 \times n_2 \times n_3 \times n_4$ that stores the result of the forward local contrast normalization layer. This input can be an object of any class derived from <code>Tensor</code> .

Result ID	Result										
layerData	Collection of input data needed for the backward local contrast normalization layer. This collection can contain objects of any class derived from <code>Tensor</code> .										
	<table> <tr> <th>Element ID</th><th>Element</th></tr> <tr> <td>auxInvMax</td><td> <p>Pointer to the tensor:</p> <ul style="list-style-type: none"> • $1/m_{nij}$ of size $n_1 \times n_3 \times n_4$ if <i>sumDimension</i> is not NULL • $1/m_{nqij}$ of size $n_1 \times n_2 \times n_3 \times n_4$ if <i>sumDimension</i> is NULL <p>This tensor stores the inverted max values. This input can be an object of any class derived from <code>Tensor</code>.</p> </td></tr> <tr> <td>auxCenteredData</td><td> <p>Pointer to tensor v_{nqij} of size $n_1 \times n_2 \times n_3 \times n_4$ that stores values as shown above. This input can be an object of any class derived from <code>Tensor</code>.</p> </td></tr> <tr> <td>auxSigma</td><td> <p>Pointer to tensor σ_{nij} of size $n_1 \times n_3 \times n_4$ if <i>sumDimension</i> is not NULL, or tensor σ_{nqij} of size $n_1 \times n_2 \times n_3 \times n_4$ otherwise, that stores values as shown above. This input can be an object of any class derived from <code>Tensor</code>.</p> </td></tr> <tr> <td>auxC</td><td> <p>Pointer to tensor c_n of size n_1 if <i>sumDimension</i> is not NULL, or tensor c_{nq} of size $n_1 \times n_2$ otherwise, that stores values as shown above. This input can be an object of any class derived from <code>Tensor</code>.</p> </td></tr> </table>	Element ID	Element	auxInvMax	<p>Pointer to the tensor:</p> <ul style="list-style-type: none"> • $1/m_{nij}$ of size $n_1 \times n_3 \times n_4$ if <i>sumDimension</i> is not NULL • $1/m_{nqij}$ of size $n_1 \times n_2 \times n_3 \times n_4$ if <i>sumDimension</i> is NULL <p>This tensor stores the inverted max values. This input can be an object of any class derived from <code>Tensor</code>.</p>	auxCenteredData	<p>Pointer to tensor v_{nqij} of size $n_1 \times n_2 \times n_3 \times n_4$ that stores values as shown above. This input can be an object of any class derived from <code>Tensor</code>.</p>	auxSigma	<p>Pointer to tensor σ_{nij} of size $n_1 \times n_3 \times n_4$ if <i>sumDimension</i> is not NULL, or tensor σ_{nqij} of size $n_1 \times n_2 \times n_3 \times n_4$ otherwise, that stores values as shown above. This input can be an object of any class derived from <code>Tensor</code>.</p>	auxC	<p>Pointer to tensor c_n of size n_1 if <i>sumDimension</i> is not NULL, or tensor c_{nq} of size $n_1 \times n_2$ otherwise, that stores values as shown above. This input can be an object of any class derived from <code>Tensor</code>.</p>
Element ID	Element										
auxInvMax	<p>Pointer to the tensor:</p> <ul style="list-style-type: none"> • $1/m_{nij}$ of size $n_1 \times n_3 \times n_4$ if <i>sumDimension</i> is not NULL • $1/m_{nqij}$ of size $n_1 \times n_2 \times n_3 \times n_4$ if <i>sumDimension</i> is NULL <p>This tensor stores the inverted max values. This input can be an object of any class derived from <code>Tensor</code>.</p>										
auxCenteredData	<p>Pointer to tensor v_{nqij} of size $n_1 \times n_2 \times n_3 \times n_4$ that stores values as shown above. This input can be an object of any class derived from <code>Tensor</code>.</p>										
auxSigma	<p>Pointer to tensor σ_{nij} of size $n_1 \times n_3 \times n_4$ if <i>sumDimension</i> is not NULL, or tensor σ_{nqij} of size $n_1 \times n_2 \times n_3 \times n_4$ otherwise, that stores values as shown above. This input can be an object of any class derived from <code>Tensor</code>.</p>										
auxC	<p>Pointer to tensor c_n of size n_1 if <i>sumDimension</i> is not NULL, or tensor c_{nq} of size $n_1 \times n_2$ otherwise, that stores values as shown above. This input can be an object of any class derived from <code>Tensor</code>.</p>										

Examples

Refer to the following examples in your Intel DAAL directory:

C++: `./examples/cpp/source/neural_networks/lcn_layer_dense_batch.cpp`

Java*: `./examples/java/source/com/intel/daal/examples/neural_networks/LCNLayerDenseBatch.java`

Python*: `./examples/python/source/neural_networks/lcn_layer_dense_batch.py`

Local Contrast Normalization Backward Layer

For given dimensions k_1 of size n_{k_1} , k_2 of size n_{k_2} , and f different from k_1 and k_2 , the forward local contrast normalization layer normalizes the input p -dimensional tensor $X \in R^{n_1 \times n_2 \times \dots \times n_p}$. For more details, see [Forward Local Contrast Normalization Layer](#).

The library supports four-dimensional input tensors $X \in R^{n_1 \times n_2 \times n_3 \times n_4}$.

Without loss of generality let's assume that backward local contrast normalization is applied to the last two dimensions. The backward local contrast normalization layer takes:

- Four-dimensional tensor $X \in R^{n_1 \times n_2 \times n_3 \times n_4}$
- Four-dimensional tensor $G \in R^{n_1 \times n_2 \times n_3 \times n_4}$ with the gradient computed on the preceding layer
- Two-dimensional tensor $K \in R^{m_1 \times m_2}$ that contains kernel parameters/weights of kernels, where $m_1 \leq n_3$, $m_2 \leq n_4$

The layer computes the four-dimensional value tensor $Z \in R^{n_1 \times n_2 \times n_3 \times n_4}$:

$$z_{nqij} = \frac{\partial E}{\partial x_{nqij}} = g_{nqij}^{(1)} + g_{nqij}^{(2)}.$$

Problem Statement

The computation depends on whether the dimension f is set:

- Dimension f is set; let n_2 be the sum dimension:

$$y_{nqij} = \frac{x_{nqij}^{(5)}}{x_{nij}^{(13)}},$$

$$g_{nqij}^{(5)} = \frac{\partial E}{\partial x_{nqij}^{(5)}} = \frac{g_{nqij}}{x_{nij}^{(13)}},$$

$$g_{nij}^{(13)} = \frac{\partial E}{\partial x_{nij}^{(13)}} = -\frac{1}{(x_{nqij}^{(13)})^2} \sum_{q=0}^{n_2-1} g_{nqij} \cdot x_{nqij}^{(5)},$$

$$m_{nij} = \max(x_n^{(12)}, x_{nij}^{(10)}),$$

$$g_n^{(12)} = \frac{\partial E}{\partial x_n^{(12)}} = \sum_{i=0}^{n_3-1} \sum_{j=0}^{n_4-1} g_{nij}^{(13)} (1 - q_{nij}),$$

$$x_n^{(12)} = c_n = \frac{1}{n_3 n_4} \sum_{i=0}^{n_3-1} \sum_{j=0}^{n_4-1} x_{nij}^{(11)},$$

$$g_{nij}^{(10)} = \frac{\partial E}{\partial x_{nij}^{(10)}} = q_{nij} \cdot g_{nij}^{(13)},$$

$$q_{nij} = \begin{cases} 1, & \sigma_{nij} > c_n \\ 0, & \text{otherwise} \end{cases},$$

$$x_{nij}^{(10)} = x_{nij}^{(11)} = x_{nij}^{(9)},$$

$$g_{nij}^{(11)} = \frac{\partial E}{\partial x_{nij}^{(11)}} = \frac{1}{n_3 n_4} g_n^{(12)},$$

$$x_{nij}^{(9)} = \sigma_{nij} = (x_{nij}^{(8)})^{\frac{1}{2}},$$

$$g_{nij}^{(9)} = g_{nij}^{(10)} + g_{nij}^{(11)},$$

$$g_{nij}^{(8)} = \frac{\partial E}{\partial x_{nij}^{(8)}} = \frac{1}{2} g_{nij}^{(9)} (x_{nij}^{(8)})^{-\frac{1}{2}},$$

$$\begin{aligned}
 x_{nij}^{(8)} = conv(x_{nqij}^{(7)}) &= \sum_{q=0}^{n_2-1} \sum_{a=-m_1/2}^{m_1/2} \sum_{b=-m_2/2}^{m_2/2} k_{ab} \\
 &\cdot x_{nq(i+a)(j+b)}^{(7)}, \\
 x_{nqij}^{(7)} &= \left(x_{nqij}^{(6)}\right)^2, \\
 x_{nqij}^{(6)} = x_{nqij}^{(5)} = x_{nqij}^{(4)} &= centeredData_{nqij}, \\
 x_{nqij}^{(4)} = x_{nqij}^{(2)} - x_{nqij}^{(3)}, \\
 x_{nij}^{(3)} = conv(x_{nqij}^{(1)}) &= \sum_{q=0}^{n_2-1} \sum_{a=-m_1/2}^{m_1/2} \sum_{b=-m_2/2}^{m_2/2} k_{ab} \\
 &\cdot x_{nq(i+a)(j+b)}^{(1)}, \\
 x_{nqij}^{(1)} = x_{nqij}^{(2)} = x_{nqij}, \\
 g_{nqij}^{(7)} = \frac{\partial E}{\partial x_{nqij}^{(7)}} &= dConv(g_{nij}^{(8)}), \\
 g_{nqij}^{(6)} = \frac{\partial E}{\partial x_{nqij}^{(6)}} &= 2 \cdot g_{nqij}^{(7)} \cdot x_{nqij}^{(6)}, \\
 g_{nqij}^{(4)} = g_{nqij}^{(5)} + g_{nqij}^{(6)}, \\
 g_{nij}^{(3)} = \frac{\partial E}{\partial x_{nij}^{(3)}} &= - \sum_{q=0}^{n_2-1} g_{nqij}^{(4)}, \\
 g_{nqij}^{(2)} = \frac{\partial E}{\partial x_{nqij}^{(2)}} &= g_{nqij}^{(4)}, \\
 g_{nqij}^{(1)} = \frac{\partial E}{\partial x_{nqij}^{(1)}} &= dConv(g_{nij}^{(3)}), \\
 z_{nqij} &= g_{nqij}^{(1)} + g_{nqij}^{(2)}.
 \end{aligned}$$

Consequently:

$$z_{nqij} = \frac{\partial E}{\partial x_{nqij}} = g_{nqij}^{(1)} + g_{nqij}^{(2)} = dConv\left(- \sum_{q=0}^{n_2-1} g_{nqij}^{(4)}\right) + g_{nqij}^{(4)}.$$

- Dimension f is not set:

$$\begin{aligned}
 y_{nqij} &= \frac{x_{nqij}^{(5)}}{x_{nqij}^{(13)}}, \\
 g_{nqij}^{(5)} &= \frac{\partial E}{\partial x_{nqij}^{(5)}} = \frac{g_{nqij}}{x_{nqij}^{(13)}},
 \end{aligned}$$

$$m_{nqij} = \max(x_{nq}^{(12)}, x_{nqij}^{(10)}),$$

$$x_{nq}^{(12)} = c_{nq} = \frac{1}{n_3 n_4} \sum_{i=0}^{n_3-1} \sum_{j=0}^{n_4-1} x_{nqij}^{(11)},$$

$$x_{nqij}^{(10)} = x_{nqij}^{(11)} = x_{nqij}^{(9)},$$

$$x_{nqij}^{(9)} = \sigma_{nqij} = \left(x_{nqij}^{(8)}\right)^{\frac{1}{2}},$$

$$x_{nqij}^{(8)} = conv(x_{nqij}^{(7)}) = \sum_{a=-m_1/2}^{m_1/2} \sum_{b=-m_2/2}^{m_2/2} k_{ab}$$

$$\cdot x_{nq(i+a)(j+b)}^{(7)},$$

$$x_{nqij}^{(7)} = \left(x_{nqij}^{(6)}\right)^2,$$

$$x_{nqij}^{(6)} = x_{nqij}^{(5)} = x_{nqij}^{(4)} = centeredData_{nqij},$$

$$x_{nqij}^{(4)} = x_{nqij}^{(2)} - x_{nqij}^{(3)},$$

$$g_{nqij}^{(13)} = \frac{\partial E}{\partial x_{nqij}^{(13)}} = - \frac{g_{nqij} \cdot x_{nqij}^{(5)}}{\left(x_{nqij}^{(13)}\right)^2},$$

$$g_{nq}^{(12)} = \frac{\partial E}{\partial x_{nq}^{(12)}} = \sum_{i=0}^{n_3-1} \sum_{j=0}^{n_4-1} g_{nqij}^{(13)} (1 - q_{nqij}),$$

$$g_{nqij}^{(10)} = \frac{\partial E}{\partial x_{nqij}^{(10)}} = q_{nqij} \cdot g_{nqij}^{(13)},$$

$$q_{nqij} = \begin{cases} 1, & \sigma_{nqij} > c_{nq} \\ 0, & otherwise \end{cases},$$

$$g_{nqij}^{(11)} = \frac{\partial E}{\partial x_{nqij}^{(13)}} = \frac{1}{n_3 n_4} g_{nq}^{(12)},$$

$$g_{nqij}^{(9)} = g_{nqij}^{(10)} + g_{nqij}^{(11)},$$

$$g_{nqij}^{(8)} = \frac{\partial E}{\partial x_{nqij}^{(8)}} = \frac{1}{2} g_{nqij}^{(9)} \left(x_{nqij}^{(8)}\right)^{-\frac{1}{2}},$$

$$g_{nqij}^{(7)} = \frac{\partial E}{\partial x_{nqij}^{(7)}} = dConv(g_{nqij}^{(8)}),$$

$$g_{nqij}^{(6)} = \frac{\partial E}{\partial x_{nqij}^{(6)}} = 2 \cdot g_{nqij}^{(7)} \cdot x_{nqij}^{(6)},$$

$$g_{nqij}^{(4)} = g_{nqij}^{(5)} + g_{nqij}^{(6)},$$

$$g_{nqij}^{(3)} = \frac{\partial E}{\partial x_{nqij}^{(3)}} = - g_{nqij}^{(4)},$$

$$g_{nqij}^{(2)} = \frac{\partial E}{\partial x_{nqij}^{(2)}} = g_{nqij}^{(4)},$$

$$x_{nqij}^{(3)} = \text{conv}\left(x_{nqij}^{(1)}\right) = \sum_{a=-m_1/2}^{m_1/2} \sum_{b=-m_2/2}^{m_2/2} k_{ab} \cdot x_{nq(i+a)(j+b)}^{(1)},$$

$$g_{nqij}^{(1)} = dConv\left(g_{nqij}^{(3)}\right),$$

$$z_{nqij} = g_{nqij}^{(1)} + g_{nqij}^{(2)}.$$

$$x_{nqij}^{(1)} = x_{nqij}^{(2)} = x_{nqij},$$

Consequently:

$$z_{nqij} = \frac{\partial E}{\partial x_{nqij}} = g_{nqij}^{(1)} + g_{nqij}^{(2)} = dConv\left(-g_{nqij}^{(4)}\right) + g_{nqij}^{(4)}.$$

Batch Processing

Layer Input

The backward local contrast normalization layer accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Result ID	Result
inputGradient	Pointer to tensor G of size $n_1 \times n_2 \times n_3 \times n_4$ that stores the input gradient computed on the preceding layer. This input can be an object of any class derived from <code>Tensor</code> .
inputFromForward	Collection of input data needed for the backward local contrast normalization layer. This collection can contain objects of any class derived from <code>Tensor</code> .
Element ID	Element
auxInvMax	<p>Pointer to the tensor:</p> <ul style="list-style-type: none"> $1/m_{nij}$ of size $n_1 \times n_3 \times n_4$ if <code>sumDimension</code> is not NULL $1/m_{nqij}$ of size $n_1 \times n_2 \times n_3 \times n_4$ if <code>sumDimension</code> is NULL <p>This tensor stores the inverted max values. This input can be an object of any class derived from <code>Tensor</code>.</p>

Result ID	Result
	<p><code>auxCenteredData</code></p> <p>Pointer to tensor $x^{(5)}_{nqij}$ of size $n_1 \times n_2 \times n_3 \times n_4$ that stores values as shown above. This input can be an object of any class derived from <code>Tensor</code>.</p>
	<p><code>auxSigma</code></p> <p>Pointer to tensor $x^{(9)}_{nij}$ of size $n_1 \times n_3 \times n_4$ if <code>sumDimension</code> is not <code>NULL</code>, or tensor $x^{(9)}_{nqij}$ of size $n_1 \times n_2 \times n_3 \times n_4$ otherwise, that stores values as shown above. This input can be an object of any class derived from <code>Tensor</code>.</p>
	<p><code>auxC</code></p> <p>Pointer to tensor $x^{(12)}_n$ of size n_1 if <code>sumDimension</code> is not <code>NULL</code>, or tensor $x^{(12)}_{nq}$ of size $n_1 \times n_3$ otherwise, that stores values as shown above. This input can be an object of any class derived from <code>Tensor</code>.</p>

Layer Parameters

For common parameters of neural network layers, see [Common Parameters](#).

In addition to the common parameters, the backward local contrast normalization layer has the following parameters:

Parameter	Default Value	Description
<code>algorithmFPTType</code>	<code>float</code>	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<code>method</code>	<code>defaultDense</code>	Default computation method used by the algorithm, the only method supported by the layer.
<code>kernel</code>	<code>HomogenTensor<float></code> of size 5×5 with values 0.04	Tensor with sizes $m_1 \times m_2$ of the two-dimensional kernel. Only kernels with odd dimensions are currently supported.
<code>indices</code>	<code>indices(2,3)</code>	Data structure representing dimensions k_1 and k_2 for kernels.
<code>sumDimension</code>	<code>HomogenNumericTable<float></code> of size 1×1 with value 1	Numeric table of size 1×1 that stores dimension f . If it is <code>NULL</code> , there is no summation over this dimension.
<code>sigmaDegenerateCasesThreshold</code>	1e-04	The threshold to avoid degenerate cases when calculating σ^{-1} .

Layer Output

The backward local contrast normalization layer calculates the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Input ID	Input
gradient	Pointer to tensor Z of size $n_1 \times n_2 \times n_3 \times n_4$ that stores the result of the backward local contrast normalization layer. This input can be an object of any class derived from <code>Tensor</code> .

Examples

Refer to the following examples in your Intel DAAL directory:

C++: `./examples/cpp/source/neural_networks/lcn_layer_dense_batch.cpp`

Java*: `./examples/java/source/com/intel/daal/examples/neural_networks/LCNLayerDenseBatch.java`

Python*: `./examples/python/source/neural_networks/lcn_layer_dense_batch.py`

Dropout Forward Layer

The forward dropout layer computes $\text{functiony} = B(r) * x / r$ for input argument x , where $B(r)$ is a Bernoulli random variable with parameter r .

Problem Statement

Given a p -dimensional tensor X of size $n_1 \times n_2 \times \dots \times n_p$, the problem is to compute p -dimensional tensors $M = (m_{i_1 \dots i_p})$ and $Y = (y_{i_1 \dots i_p})$ of size $n_1 \times n_2 \times \dots \times n_p$, where:

$$m_{i_1 \dots i_p} = B(r) / r$$

$$y_{i_1 \dots i_p} = m_{i_1 \dots i_p} * x_{i_1 \dots i_p}$$

Batch Processing

Layer Input

The forward dropout layer accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
data	Pointer to the tensor of size $n_1 \times n_2 \times \dots \times n_p$ that stores the input data for the forward dropout layer. This input can be an object of any class derived from <code>Tensor</code> .

Layer Parameters

For common parameters of neural network layers, see [Common Parameters](#).

In addition to the common parameters, the forward dropout layer has the following parameters:

Parameter	Default Value	Description
<code>algorithmFPType</code>	<code>float</code>	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<code>method</code>	<code>defaultDense</code>	Performance-oriented computation method, the only method supported by the layer.

Parameter	Default Value	Description
<i>retainRatio</i>	0.5	Probability that any particular element is retained.
<i>seed</i>	777	Seed for random generation of mask elements.

Layer Output

The forward dropout layer calculates the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result				
value	Pointer to the tensor of size $n_1 \times n_2 \times \dots \times n_p$ that stores the result of the forward dropout layer. This input can be an object of any class derived from <code>Tensor</code> .				
resultForBackward	Collection of data needed for the backward dropout layer. This collection can contain objects of any class derived from <code>Tensor</code> .				
	<table> <tr> <th>Element ID</th><th>Element</th></tr> <tr> <td>auxRetainMask</td><td>Pointer to tensor M of size $n_1 \times n_2 \times \dots \times n_p$ that stores Bernoulli random variable values (0 on positions that were dropped, 1 on the others) divided by the probability that any particular element is retained. This input can be an object of any class derived from <code>Tensor</code>.</td></tr> </table>	Element ID	Element	auxRetainMask	Pointer to tensor M of size $n_1 \times n_2 \times \dots \times n_p$ that stores Bernoulli random variable values (0 on positions that were dropped, 1 on the others) divided by the probability that any particular element is retained. This input can be an object of any class derived from <code>Tensor</code> .
Element ID	Element				
auxRetainMask	Pointer to tensor M of size $n_1 \times n_2 \times \dots \times n_p$ that stores Bernoulli random variable values (0 on positions that were dropped, 1 on the others) divided by the probability that any particular element is retained. This input can be an object of any class derived from <code>Tensor</code> .				

Examples

Refer to the following examples in your Intel DAAL directory:

C++: `./examples/cpp/source/neural_networks/dropout_layer_batch.cpp`

Java*: `./examples/java/source/com/intel/daal/examples/neural_networks/DropoutLayerBatch.java`

Python*: `./examples/python/source/neural_networks/dropout_layer_batch.py`

Dropout Backward Layer

The dropout activation layer applies the transform $y = B(r) * x / r$ to the input data, where $B(r)$ is a Bernoulli random variable with parameter r . For more details, see the [forward dropout layer](#). The backward dropout layer computes value $z = B(r) * g / r$.

Problem Statement

Given a p -dimensional tensor $G = g_{i_1 \dots i_p}$ and $M = m_{i_1 \dots i_p}$ of size $n_1 \times n_2 \times \dots \times n_p$, the problem is to compute a p -dimensional tensor $Z = (z_{i_1 \dots i_p})$ of size $n_1 \times n_2 \times \dots \times n_p$, where:

$$z_{i_1 \dots i_p} = m_{i_1 \dots i_p} * g_{i_1 \dots i_p}$$

Batch Processing

Layer Input

The backward dropout layer accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
<code>inputGradient</code>	Pointer to the tensor of size $n_1 \times n_2 \times \dots \times n_p$ that stores the input gradient computed on the preceding layer. This input can be an object of any class derived from <code>Tensor</code> .
<code>inputFromForward</code>	Collection of input data needed for the backward dropout layer. This collection can contain objects of any class derived from <code>Tensor</code> .
Element ID	Element
<code>auxRetainMask</code>	Pointer to tensor M of size $n_1 \times n_2 \times \dots \times n_p$ that stores Bernoulli random variable values (0 on positions that were dropped, 1 on the others) divided by the probability that any particular element is retained. This input can be an object of any class derived from <code>Tensor</code> .

Layer Parameters

For common parameters of neural network layers, see [Common Parameters](#).

In addition to the common parameters, the backward dropout layer has the following parameters:

Parameter	Default Value	Description
<code>algorithmFPType</code>	<code>float</code>	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<code>method</code>	<code>defaultDense</code>	Performance-oriented computation method, the only method supported by the layer.

Layer Output

The backward dropout layer calculates the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
<code>gradient</code>	Pointer to the tensor of size $n_1 \times n_2 \times \dots \times n_p$ that stores the result of the backward dropout layer. This input can be an object of any class derived from <code>Tensor</code> .

Examples

Refer to the following examples in your Intel DAAL directory:

C++: `./examples/cpp/source/neural_networks/dropout_layer_batch.cpp`

Java*: ./examples/java/source/com/intel/daal/examples/neural_networks/
DropoutLayerBatch.java

Python*: ./examples/python/source/neural_networks/dropout_layer_batch.py

1D Max Pooling Forward Layer

The forward one-dimensional (1D) max pooling layer is a form of non-linear downsampling of an input tensor $X \in \mathbb{R}^{n_1 \times n_2 \times \dots \times n_p}$. 1D max pooling partitions the input tensor data into 1D subtensors along the dimension k , selects an element with the maximal numeric value in each subtensor, and transforms the input tensor to the output tensor Y by replacing each subtensor with its maximum element.

Problem Statement

Given:

- p -dimensional tensor $X \in \mathbb{R}^{n_1 \times n_2 \times \dots \times n_p}$ with input data.
- Dimension k along which the kernel is applied
- Kernel size m : $m \leq n_k + 2 \cdot p$, where p is the padding

The problem is to compute the value tensor $Y = (y_{i_1 \dots i_p}) \in \mathbb{R}^{l_1 \times \dots \times l_p}$ using the downsampling technique.

The layer computes the value y_j as the maximum element in the subtensor. After the kernel is applied to the subtensor at position $\{i_1, \dots, s \cdot i_k - p, \dots, i_p\}$, the index of the maximum $T = (t_{i_1 \dots i_p})$ is stored for use by the backward 1D max pooling layer:

$$y_{i_1 \dots i_k \dots i_p} = \max_{j \in [s \cdot i_k, s \cdot i_k + m - 1]} x_{i_1 \dots j \dots i_p},$$

$$t_{i_1 \dots i_k \dots i_p} = \left(j \in \left[s \cdot i_k, s \cdot i_k + m - 1 \right] \arg \max_{i_1 \dots j \dots i_p} x_{i_1 \dots j \dots i_p} \right) - s \cdot i_k,$$

where

- $l_i = \begin{cases} n_i, & i \neq k \\ \frac{n_k + 2p - m}{s} + 1, & i = k \end{cases}$
- s is the stride

The following figure illustrates the transformation:



Batch Processing

Layer Input

The forward one-dimensional max pooling layer accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
<code>data</code>	Pointer to tensor X of size $n_1 \times \dots \times n_p$ that stores the input data for the forward one-dimensional max pooling layer. This input can be an object of any class derived from <code>Tensor</code> .

Layer Parameters

For common parameters of neural network layers, see [Common Parameters](#).

In addition to the common parameters, the forward one-dimensional max pooling layer has the following parameters:

Parameter	Default Value	Description
<code>algorithmFPTYPE</code>	<code>float</code>	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<code>method</code>	<code>defaultDense</code>	Performance-oriented computation method, the only method supported by the layer.
<code>kernelSize</code>	<code>KernelSize(2)</code>	Data structure representing the size of the one-dimensional subtensor from which the maximum element is selected.
<code>stride</code>	<code>Stride(2)</code>	Data structure representing the interval on which the subtensors for max pooling are selected.
<code>padding</code>	<code>Padding(0)</code>	Data structure representing the number of data elements to implicitly add to each side of the one-dimensional subtensor along which max pooling is performed.
<code>index</code>	<code>Index(p-1)</code>	Index k of the dimension along which max pooling is performed.

Layer Output

The forward one-dimensional max pooling layer calculates the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result				
<code>value</code>	Pointer to the tensor of size $l_1 \times \dots \times l_p$ that stores the result of the forward one-dimensional max pooling layer. This input can be an object of any class derived from <code>Tensor</code> .				
<code>resultForBackward</code>	Collection of data needed for the backward one-dimensional max pooling layer.				
	<table> <tr> <th>Element ID</th><th>Element</th></tr> <tr> <td><code>auxSelectedIndices</code></td><td>Tensor T of size $l_1 \times \dots \times l_p$ that stores indices of maximum elements.</td></tr> </table>	Element ID	Element	<code>auxSelectedIndices</code>	Tensor T of size $l_1 \times \dots \times l_p$ that stores indices of maximum elements.
Element ID	Element				
<code>auxSelectedIndices</code>	Tensor T of size $l_1 \times \dots \times l_p$ that stores indices of maximum elements.				

Result ID	Result
	auxInputDimensions
	NumericTable of size $1 \times p$ that stores the sizes of the dimensions of input data tensor X : n_1, n_2, \dots, n_p .

Examples

Refer to the following examples in your Intel DAAL directory:

C++: `./examples/cpp/source/neural_networks/maximum_pooling1d_layer_batch.cpp`

Java*: `./examples/java/source/com/intel/daal/examples/neural_networks/MaximumPooling1DLayerBatch.java`

Python*: `./examples/python/source/neural_networks/maximum_pooling1d_layer_batch.py`

1D Max Pooling Backward Layer

The forward one-dimensional (1D) max pooling layer is a form of non-linear downsampling of an input tensor $X \in \mathbb{R}^{n_1 \times n_2 \times \dots \times n_p}$. 1D max pooling partitions the input tensor data into 1D subtensors along the dimension k , selects an element with the maximal numeric value in each subtensor, and transforms the input tensor to the output tensor Y by replacing each subtensor with its maximum element. For more details, see [Forward 1D Max Pooling Layer](#).

The backward 1D max pooling layer back-propagates the input gradient $G \in \mathbb{R}^{l_1 \times \dots \times l_p}$ computed on the preceding layer. The backward layer propagates to the next layer only the elements of the gradient that correspond to the maximum values pooled from subtensors in the forward computation step.

Problem Statement

Given:

- p -dimensional tensor $G \in \mathbb{R}^{l_1 \times \dots \times l_p}$ with the gradient computed on the preceding layer
- Dimension k along which the kernel is applied
- Kernel size m : $m \leq n_k + 2 \cdot p$, where p is the padding

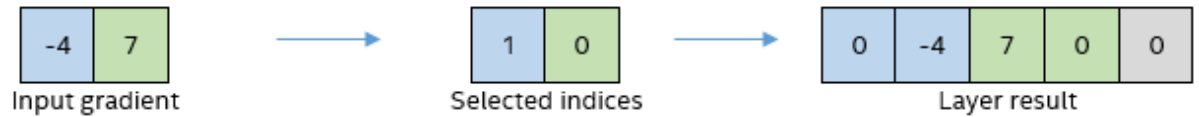
The problem is to compute the value tensor $Z = (z_{i_1 \dots i_p}) \in \mathbb{R}^{n_1 \times \dots \times n_p}$ such that:

$$z_{i_1 \dots i_p} = \frac{\partial E}{\partial x_{i_1 \dots i_p}} = \sum_{j, a \in V(i_k)} \delta_{a, t} g_{i_1 \dots j \dots i_p},$$

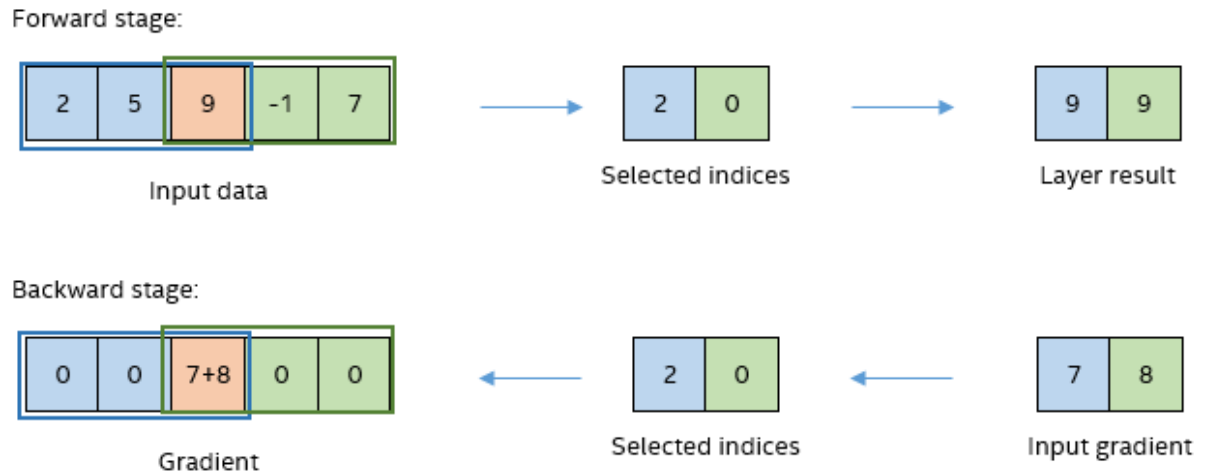
where:

- $\delta_{i, j} = \begin{cases} 1, & i = j \\ 0, & i \neq j \end{cases}$
- $t = s_{i_1 \dots j_k \dots i_p}$
- $V(i) = \left\{ \{j, a\} : \frac{j+p-a}{s} = i, \quad j \in \{1, \dots, l_k\}, \quad a \in \{1, \dots, m\} \right\}$
- $l_i = \begin{cases} n_i, & i \neq k \\ \left\lfloor \frac{n_k + 2p - m}{s} \right\rfloor + 1, & i = k \end{cases}$
- s is the stride

The following figure illustrates the transformation:



If $m > s$ and overlapping subtensors are represented with the same maximum located at the same position in the input tensor X , the gradient value z at this position is the sum of input gradients g at the respective positions, as shown in the following figure for $m = 3$ and $s = 2$:



Batch Processing

Layer Input

The backward one-dimensional max pooling layer accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input						
<code>inputGradient</code>	Pointer to tensor G of size $l_1 \times \dots \times l_p$ that stores the input gradient computed on the preceding layer. This input can be an object of any class derived from <code>Tensor</code> .						
<code>inputFromForward</code>	Collection of data needed for the backward one-dimensional max pooling layer.						
	<table> <tr> <th>Element ID</th><th>Element</th></tr> <tr> <td><code>auxSelectedIndices</code></td><td>Tensor T of size $l_1 \times \dots \times l_p$ that stores indices of maximum elements.</td></tr> <tr> <td><code>auxInputDimensions</code></td><td>NumericTable of size $1 \times p$ that stores the sizes of the dimensions of input data tensor X: n_1, n_2, \dots, n_p.</td></tr> </table>	Element ID	Element	<code>auxSelectedIndices</code>	Tensor T of size $l_1 \times \dots \times l_p$ that stores indices of maximum elements.	<code>auxInputDimensions</code>	NumericTable of size $1 \times p$ that stores the sizes of the dimensions of input data tensor X : n_1, n_2, \dots, n_p .
Element ID	Element						
<code>auxSelectedIndices</code>	Tensor T of size $l_1 \times \dots \times l_p$ that stores indices of maximum elements.						
<code>auxInputDimensions</code>	NumericTable of size $1 \times p$ that stores the sizes of the dimensions of input data tensor X : n_1, n_2, \dots, n_p .						

Layer Parameters

For common parameters of neural network layers, see [Common Parameters](#).

In addition to the common parameters, the backward one-dimensional max pooling layer has the following parameters:

Parameter	Default Value	Description
<code>algorithmFPType</code>	<code>float</code>	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<code>method</code>	<code>defaultDense</code>	Performance-oriented computation method, the only method supported by the layer.
<code>kernelSize</code>	<code>KernelSize(2)</code>	Data structure representing the size of the one-dimensional subtensor from which the maximum element is selected.
<code>stride</code>	<code>Stride(2)</code>	Data structure representing the interval on which the subtensors for max pooling are selected.
<code>padding</code>	<code>Padding(0)</code>	Data structure representing the number of data elements to implicitly add to each side of the one-dimensional subtensor along which max pooling is performed.
<code>index</code>	<code>Index(p-1)</code>	Index k of the dimension along which max pooling is performed.

Layer Output

The backward one-dimensional max pooling layer calculates the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
<code>gradient</code>	Pointer to tensor Z of size $n_1 \times \dots \times n_p$ that stores the result of the backward one-dimensional max pooling layer. This input can be an object of any class derived from <code>Tensor</code> .

Examples

Refer to the following examples in your Intel DAAL directory:

C++: `./examples/cpp/source/neural_networks/maximum_pooling1d_layer_batch.cpp`

Java*: `./examples/java/source/com/intel/daal/examples/neural_networks/MaximumPooling1DLayerBatch.java`

Python*: `./examples/python/source/neural_networks/maximum_pooling1d_layer_batch.py`

2D Max Pooling Forward Layer

The forward two-dimensional (2D) max pooling layer is a form of non-linear downsampling of an input tensor $X \in \mathbb{R}^{n_1 \times n_2 \times \dots \times n_p}$. 2D max pooling partitions the input tensor data into 2D subtensors along dimensions k_1 and k_2 , selects an element with the maximal numeric value in each subtensor, and transforms the input tensor to the output tensor Y by replacing each subtensor with its maximum element.

Problem Statement

Given:

- p -dimensional tensor $X \in R^{n_1 \times n_2 \times \dots \times n_p}$ with input data.
- Dimensions k_1 and k_2 along which the kernel is applied
- Kernel sizes m_1 and m_2 : $m_i \leq n_{k_i} + 2 \cdot p_i$, $i \in \{1, 2\}$, where p_1 and p_2 are paddings

The problem is to compute the value tensor $Y = (y_{i_1 \dots i_p}) \in R^{l_1 \times \dots \times l_p}$ using the downsampling technique.

The layer computes the value $y_{i_1 \dots i_p}$ as the maximum element in the subtensor. After the kernel is applied to the subtensor at position $\{i_1, \dots, s_1 \cdot i_{k_1} - p_1, \dots, s_2 \cdot i_{k_2} - p_2, \dots, i_p\}$, the index of the maximum $T = (t_{i_1 \dots i_p})$ is stored for use by the backward 2D max pooling layer:

$$y_{i_1 \dots i_{k_1} \dots i_{k_2} \dots i_p} = x_{i_1 \dots j_{k_1} \dots j_{k_2} \dots i_p},$$

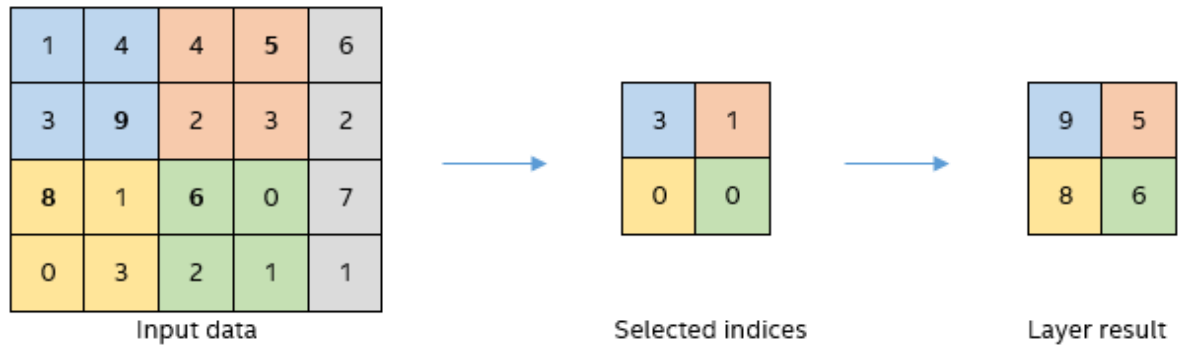
$$t_{i_1 \dots i_{k_1} \dots i_{k_2} \dots i_p} = (j_{k_1} - s_1 \cdot i_{k_1}) \cdot m_2 + (j_{k_2} - s_2 \cdot i_{k_2}),$$

$$(j_{k_1}, j_{k_2}) = \underset{(j_1, j_2) \in [s_1 \cdot i_{k_1}, s_1 \cdot i_{k_1} + m_1 - 1] \times [s_2 \cdot i_{k_2}, s_2 \cdot i_{k_2} + m_2 - 1]}{\operatorname{argmax}} x_{i_1 \dots j_1 \dots j_2 \dots i_p},$$

where

- $l_i = \begin{cases} n_i, & i \notin \{k_1, k_2\} \\ \frac{n_{k_j} + 2p_j - m_j}{s_j} + 1, & i = k_j, j \in \{1, 2\} \end{cases}$
- s_1 and s_2 are strides

The following figure illustrates the transformation.



Batch Processing

Layer Input

The forward two-dimensional max pooling layer accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
<code>data</code>	Pointer to tensor X of size $n_1 \times \dots \times n_p$ that stores the input data for the forward two-dimensional max pooling layer. This input can be an object of any class derived from <code>Tensor</code> .

Layer Parameters

For common parameters of neural network layers, see [Common Parameters](#).

In addition to the common parameters, the forward two-dimensional max pooling layer has the following parameters:

Parameter	Default Value	Description
<code>algorithmFPType</code>	<code>float</code>	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<code>method</code>	<code>defaultDense</code>	Performance-oriented computation method, the only method supported by the layer.
<code>kernelSizes</code>	<code>KernelSizes(2, 2)</code>	Data structure representing the size of the two-dimensional subtensor from which the maximum element is selected.
<code>strides</code>	<code>Strides(2, 2)</code>	Data structure representing intervals s_1, s_2 on which the subtensors for max pooling are selected.
<code>padding</code> s	<code>Paddings(0, 0)</code>	Data structure representing the number of data elements to implicitly add to each side of the two-dimensional subtensor along which max pooling is performed.
<code>indices</code>	<code>Indices(p-2, p-1)</code>	Indices k_1, k_2 of the dimensions along which max pooling is performed.

Layer Output

The forward two-dimensional max pooling layer calculates the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
<code>value</code>	Pointer to tensor Y of size $l_1 \times \dots \times l_p$ that stores the result of the forward two-dimensional max pooling layer. This input can be an object of any class derived from <code>Tensor</code> .
<code>resultForBackward</code>	Collection of data needed for the backward two-dimensional max pooling layer.
Element ID	Element

Result ID	Result
	auxSelectedIndices
	Tensor T of size $l_1 \times \dots \times l_p$ that stores indices of maximum elements.
	auxInputDimensions
	NumericTable of size $1 \times p$ that stores the sizes of the dimensions of input data tensor X : n_1, n_2, \dots, n_p .

Examples

Refer to the following examples in your Intel DAAL directory:

C++: `./examples/cpp/source/neural_networks/maximum_pooling2d_layer_batch.cpp`

Java*: `./examples/java/source/com/intel/daal/examples/neural_networks/MaximumPooling2DLayerBatch.java`

Python*: `./examples/python/source/neural_networks/maximum_pooling2d_layer_batch.py`

2D Max Pooling Backward Layer

The forward two-dimensional (2D) max pooling layer is a form of non-linear downsampling of an input tensor $X \in \mathbb{R}^{n_1 \times n_2 \times \dots \times n_p}$. 2D max pooling partitions the input tensor data into 2D subtensors along dimensions k_1 and k_2 , selects an element with the maximal numeric value in each subtensor, and transforms the input tensor to the output tensor Y by replacing each subtensor with its maximum element. For more details, see [Forward 2D Max Pooling Layer](#).

The backward 2D max pooling layer back-propagates the input gradient $G \in \mathbb{R}^{l_1 \times \dots \times l_p}$ computed on the preceding layer. The backward layer propagates to the next layer only the elements of the gradient that correspond to the maximum values pooled from subtensors in the forward computation step.

Problem Statement

Given:

- p -dimensional tensor $G \in \mathbb{R}^{l_1 \times \dots \times l_p}$ with the gradient computed on the preceding layer
- Dimensions k_1 and k_2 along which the kernel is applied
- Kernel sizes m_1 and m_2 : $m_i \leq n_{k_i} + 2 \cdot p_i$, $i \in \{1, 2\}$, where p_1 and p_2 are paddings

The problem is to compute the value tensor $Z = (z_{i_1 \dots i_p}) \in \mathbb{R}^{n_1 \times \dots \times n_p}$ such that:

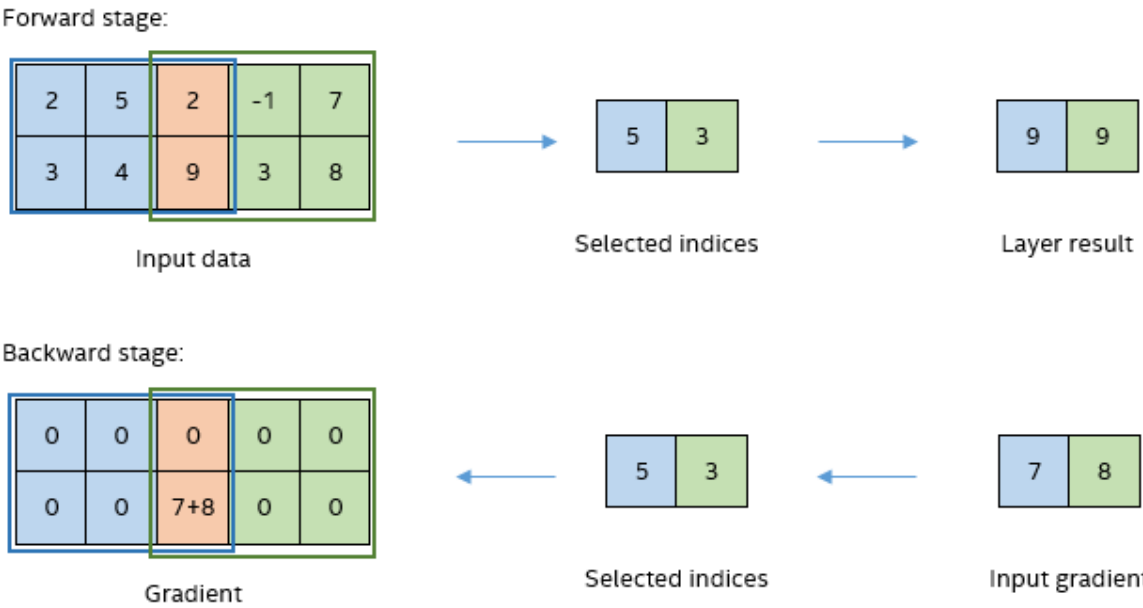
$$z_{i_1 \dots i_p} = \frac{\partial E}{\partial x_{i_1 \dots i_p}} = \sum_{j_{k_1}, a \in V_1(i_{k_1}), j_{k_2}, b \in V_2(i_{k_2})} \delta_{a \cdot m_2 + b, t} g_{i_1 \dots j_{k_1} \dots j_{k_2} \dots i_p},$$

where:

- $\delta_{i,j} = \begin{cases} 1, & i = j \\ 0, & i \neq j \end{cases}$
- $t = i_{i_1 \dots j_{k_1} \dots j_{k_2} \dots i_p}$
- $V_h(i) = \left\{ \{j, a\} : \frac{j + p_h - a}{s_h} = i, \quad j \in \{1, \dots, l_{k_h}\}, \quad a \in \{1, \dots, m_h\} \right\}, \quad h \in \{1, 2\}$

- $$l_i = \begin{cases} n_i, & i \notin \{k_1, k_2\} \\ \frac{n_{k_j} + 2p_j - m_j}{s_j} + 1, & i = k_j, j \in \{1, 2\} \end{cases}$$
- s_1 and s_2 are strides

If $m_1 > s_1$ and/or $m_2 > s_2$ and if overlapping subensors are represented with the same maximum located at the same position in the input tensor X , the gradient value z at this position is the sum of input gradients g at the respective positions, as shown in the following figure for $m_1 = 3$, $m_2 = 2$, and $s_1 = 2$:



Batch Processing

Layer Input

The backward two-dimensional max pooling layer accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input				
<code>inputGradient</code>	Pointer to tensor G of size $l_1 \times \dots \times l_p$ that stores the input gradient computed on the preceding layer. This input can be an object of any class derived from <code>Tensor</code> .				
<code>inputFromForward</code>	Collection of data needed for the backward two-dimensional max pooling layer.				
	<table> <tr> <th>Element ID</th><th>Element</th></tr> <tr> <td><code>auxSelectedIndices</code></td><td>Tensor T of size $l_1 \times \dots \times l_p$ that stores indices of maximum elements.</td></tr> </table>	Element ID	Element	<code>auxSelectedIndices</code>	Tensor T of size $l_1 \times \dots \times l_p$ that stores indices of maximum elements.
Element ID	Element				
<code>auxSelectedIndices</code>	Tensor T of size $l_1 \times \dots \times l_p$ that stores indices of maximum elements.				

Input ID	Input
	<div>auxInputDimensions</div> <div>NumericTable of size $1 \times p$ that stores the sizes of the dimensions of input data tensor X: n_1, n_2, \dots, n_p.</div>

Layer Parameters

For common parameters of neural network layers, see [Common Parameters](#).

In addition to the common parameters, the backward two-dimensional max pooling layer has the following parameters:

Parameter	Default Value	Description
<i>algorithmFPType</i>	float	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<i>method</i>	defaultDense	Performance-oriented computation method, the only method supported by the layer.
<i>kernelSizes</i>	KernelSizes(2, 2)	Data structure representing the size of the two-dimensional subtensor from which the maximum element is selected.
<i>strides</i>	Strides(2, 2)	Data structure representing intervals s_1, s_2 on which the subtensors for max pooling are selected.
<i>padding</i> s	Paddings(0, 0)	Data structure representing the number of data elements to implicitly add to each side of the two-dimensional subtensor along which max pooling is performed.
<i>indices</i>	Indices(p-2, p-1)	Indices k_1, k_2 of the dimensions along which max pooling is performed.

Layer Output

The backward two-dimensional max pooling layer calculates the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
gradient	Pointer to tensor Z of size $n_1 \times \dots \times n_p$ that stores the result of the backward two-dimensional max pooling layer. This input can be an object of any class derived from <code>Tensor</code> .

Examples

Refer to the following examples in your Intel DAAL directory:

C++: ./examples/cpp/source/neural_networks/maximum_pooling2d_layer_batch.cpp

Java*: ./examples/java/source/com/intel/daal/examples/neural_networks/
MaximumPooling2DLayerBatch.java

Python*: ./examples/python/source/neural_networks/maximum_pooling2d_layer_batch.py

3D Max Pooling Forward Layer

The forward three-dimensional (3D) max pooling layer is a form of non-linear downsampling of an input tensor $X \in \mathbb{R}^{n_1 \times n_2 \times \dots \times n_p}$. 3D max pooling partitions the input tensor data into 3D subtensors along dimensions k_1 , k_2 , and k_3 , selects an element with the maximal numeric value in each subtensor, and transforms the input tensor to the output tensor Y by replacing each subtensor with its maximum element.

Problem Statement

Given:

- p -dimensional tensor $X \in \mathbb{R}^{n_1 \times n_2 \times \dots \times n_p}$ with input data.
- Dimensions k_1 , k_2 , and k_3 along which the kernel is applied
- Kernel sizes m_1 , m_2 , and m_3 : $m_i \leq n_{k_i} + 2 \cdot p_i$, $i \in \{1, 2, 3\}$, where p_1 , p_2 and p_3 are paddings

The problem is to compute the value tensor $Y = (y_{i_1 \dots i_p}) \in \mathbb{R}^{l_1 \times \dots \times l_p}$ using the downsampling technique.

The layer computes the value $y_{i_1 \dots i_p}$ as the maximum element in the subtensor. After the kernel is applied to the subtensor at position $\{i_1, \dots, s_1 \cdot i_{k_1} - p_1, \dots, s_2 \cdot i_{k_2} - p_2, \dots, s_3 \cdot i_{k_3} - p_3, \dots, i_p\}$, the index of the maximum $T = (t_{i_1 \dots i_p})$ is stored for use by the backward 3D max pooling layer:

$$y_{i_1 \dots i_{k_1} \dots i_{k_2} \dots i_{k_3} \dots i_p} = x_{i_1 \dots j_{k_1} \dots j_{k_2} \dots j_{k_3} \dots i_p},$$

$$t_{i_1 \dots i_{k_1} \dots i_{k_2} \dots i_{k_3} \dots i_p} = (j_{k_1} - s_1 \cdot i_{k_1}) \cdot m_2 \cdot m_3 + (j_{k_2} - s_2 \cdot i_{k_2}) \cdot m_2 + (j_{k_3} - s_3 \cdot i_{k_3}),$$

$$(j_{k_1}, j_{k_2}, j_{k_3}) = \underset{(j_1, j_2, j_3) \in d_1 \times d_2 \times d_3}{\operatorname{argmax}} x_{i_1 \dots j_1 \dots j_2 \dots j_3 \dots i_p},$$

$$d_j = \left[s_j \cdot i_{k_j}, \quad s_j \cdot i_{k_j} + m_j - 1 \right], j \in \{1, 2, 3\},$$

where

- $l_i = \begin{cases} n_i, & i \notin \{k_1, k_2, k_3\} \\ \frac{n_{k_j} + 2p_j - m_j}{s_j} + 1, & i = k_j, \quad j \in \{1, 2, 3\} \end{cases}$
- s_1 , s_2 , and s_3 are strides

Batch Processing

Layer Input

The forward three-dimensional max pooling layer accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
<code>data</code>	Pointer to tensor X of size $n_1 \times \dots \times n_p$ that stores the input data for the forward three-dimensional max pooling layer. This input can be an object of any class derived from <code>Tensor</code> .

Layer Parameters

For common parameters of neural network layers, see [Common Parameters](#).

In addition to the common parameters, the forward three-dimensional max pooling layer has the following parameters:

Parameter	Default Value	Description
<code>algorithmFPType</code>	<code>float</code>	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<code>method</code>	<code>defaultDense</code>	Performance-oriented computation method, the only method supported by the layer.
<code>kernelSizes</code>	<code>KernelSizes(2, 2, 2)</code>	Data structure representing the size of the three-dimensional subtensor from which the maximum element is selected.
<code>strides</code>	<code>Strides(2, 2, 2)</code>	Data structure representing intervals s_1, s_2, s_3 on which the subtensors for max pooling are selected.
<code>padding</code>	<code>Paddings(0, 0, 0)</code>	Data structure representing the number of data elements to implicitly add to each side of the three-dimensional subtensor along which max pooling is performed.
<code>indices</code>	<code>Indices(p-3, p-2, p-1)</code>	Indices k_1, k_2, k_3 of the dimensions along which max pooling is performed.

Layer Output

The forward three-dimensional max pooling layer calculates the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
<code>value</code>	Pointer to tensor Y of size $l_1 \times \dots \times l_p$ that stores the result of the forward three-dimensional max pooling layer. This input can be an object of any class derived from <code>Tensor</code> .
<code>resultForBackward</code>	Collection of data needed for the backward three-dimensional max pooling layer.

Result ID	Result	
	Element ID	Element
	auxSelectedIndices	Tensor T of size $l_1 \times \dots \times l_p$ that stores indices of maximum elements.
	auxInputDimensions	NumericTable of size $1 \times p$ that stores the sizes of the dimensions of input data tensor X : n_1, n_2, \dots, n_p .

Examples

Refer to the following examples in your Intel DAAL directory:

C++: `./examples/cpp/source/neural_networks/maximum_pooling3d_layer_batch.cpp`

Java*: `./examples/java/source/com/intel/daal/examples/neural_networks/MaximumPooling3DLayerBatch.java`

Python*: `./examples/python/source/neural_networks/maximum_pooling3d_layer_batch.py`

3D Max Pooling Backward Layer

The forward three-dimensional (3D) max pooling layer is a form of non-linear downsampling of an input tensor $X \in \mathbb{R}^{n_1 \times n_2 \times \dots \times n_p}$. 3D max pooling partitions the input tensor data into 3D subtensors along dimensions k_1, k_2 , and k_3 , selects an element with the maximal numeric value in each subtensor, and transforms the input tensor to the output tensor by replacing each subtensor with its maximum element. For more details, see [Forward 3D Max Pooling Layer](#).

The backward 3D max pooling layer back-propagates the input gradient $G \in \mathbb{R}^{l_1 \times \dots \times l_p}$ computed on the preceding layer. The backward layer propagates to the next layer only the elements of the gradient that correspond to the maximum values pooled from subtensors in the forward computation step.

Problem Statement

Given:

- p -dimensional tensor $G \in \mathbb{R}^{l_1 \times \dots \times l_p}$ with the gradient computed on the preceding layer
- Dimensions k_1, k_2 , and k_3 along which the kernel is applied
- Kernel sizes m_1, m_2 , and m_3 : $m_i \leq n_{k_i} + 2 \cdot p_i$, $i \in \{1, 2, 3\}$, where p_1, p_2 and p_3 are paddings

The problem is to compute the value tensor $Z = (z_{i_1 \dots i_p}) \in \mathbb{R}^{n_1 \times \dots \times n_p}$ such that:

$$z_{i_1 \dots i_p} = \frac{\partial E}{\partial x_{i_1 \dots i_p}} = \sum_{j_{k_1}, a \in V_1(i_{k_1}), j_{k_2}, b \in V_2(i_{k_2}), j_{k_3}, c \in V_3(i_{k_3})} \delta_{a \cdot m_2 \cdot m_3 + b \cdot m_2 + c, t} g_{i_1 \dots j_{k_1} \dots j_{k_2} \dots j_{k_3} \dots i_p},$$

where:

- $\delta_{i,j} = \begin{cases} 1, & i = j \\ 0, & i \neq j \end{cases}$
- $t = i_1 \dots j_{k_1} \dots j_{k_2} \dots j_{k_3} \dots i_p$
- $V_h(i) = \left\{ \{j, a\} : \frac{j + p_h - a}{s_h} = i, \quad j \in \{1, \dots, l_{k_h}\}, \quad a \in \{1, \dots, m_h\} \right\}, \quad h \in \{1, 2, 3\}$

- $$l_i = \begin{cases} n_i, & i \notin \{k_1, k_2, k_3\} \\ \frac{n_{k_j} + 2p_j - m_j}{s_j} + 1, & i = k_j, \quad j \in \{1, 2, 3\} \end{cases}$$
- s_1, s_2 , and s_3 are strides

If $m_1 > s_1$, $m_2 > s_2$, or $m_3 > s_3$ (including the cases where any of these conditions are met simultaneously) and if overlapping subtensors are represented with the same maximum located at the same position in the input tensor X , the gradient value z at this position is the sum of input gradients g at the respective positions. This behavior is similar to the behavior of two-dimensional max pooling. Therefore, for an illustration see [Backward 2D Max Pooling Layer](#).

Batch Processing

Layer Input

The backward three-dimensional max pooling layer accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input	
inputGradient	Pointer to tensor G of size $l_1 \times \dots \times l_p$ that stores the input gradient computed on the preceding layer. This input can be an object of any class derived from <code>Tensor</code> .	
inputFromForward	Collection of data needed for the backward three-dimensional max pooling layer.	
	Element ID	Element
	auxSelectedIndices	Tensor T of size $l_1 \times \dots \times l_p$ that stores indices of maximum elements.
	auxInputDimensions	NumericTable of size $1 \times p$ that stores the sizes of the dimensions of input data tensor X : n_1, n_2, \dots, n_p .

Layer Parameters

For common parameters of neural network layers, see [Common Parameters](#).

In addition to the common parameters, the backward three-dimensional max pooling layer has the following parameters:

Parameter	Default Value	Description
<code>algorithmFPType</code>	<code>float</code>	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<code>method</code>	<code>defaultDense</code>	Performance-oriented computation method, the only method supported by the layer.

Parameter	Default Value	Description
<i>kernelSizes</i>	<code>KernelSizes(2, 2, 2)</code>	Data structure representing the size of the three-dimensional subtensor from which the maximum element is selected.
<i>strides</i>	<code>Strides(2, 2, 2)</code>	Data structure representing intervals s_1, s_2, s_3 on which the subtensors for max pooling are selected.
<i>padding</i> s	<code>Paddings(0, 0, 0)</code>	Data structure representing the number of data elements to implicitly add to each side of the three-dimensional subtensor along which max pooling is performed.
<i>indices</i>	<code>Indices(p-3, p-2, p-1)</code>	Indices k_1, k_2, k_3 of the dimensions along which max pooling is performed.

Layer Output

The backward three-dimensional max pooling layer calculates the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
<code>gradient</code>	Pointer to tensor Z of size $n_1 \times \dots \times n_p$ that stores the result of the backward three-dimensional max pooling layer. This input can be an object of any class derived from <code>Tensor</code> .

Examples

Refer to the following examples in your Intel DAAL directory:

C++: `./examples/cpp/source/neural_networks/maximum_pooling3d_layer_batch.cpp`

Java*: `./examples/java/source/com/intel/daal/examples/neural_networks/MaximumPooling3DLayerBatch.java`

Python*: `./examples/python/source/neural_networks/maximum_pooling3d_layer_batch.py`

1D Average Pooling Forward Layer

The forward one-dimensional (1D) average pooling layer is a form of non-linear downsampling of an input tensor $X = (x^{(1)} \dots x^{(p)})$ of size $n_1 \times n_2 \times \dots \times n_p$. 1D average pooling partitions the input tensor data into 1D subtensors by one dimension k , computes the average value of elements in each subtensor, and transforms the input tensor to the output tensor $Y = (y^{(1)} \dots y^{(p)})$ of size $m_1 \times m_2 \times \dots \times m_p$ by replacing each subtensor with one element, the average of the subtensor:

$$y_{i_1 \dots i_k \dots i_p} = \frac{1}{f_k} \sum_{j_k \in [s_k * i_k, s_k * i_k + f_k - 1]} x_{i_1 \dots j_k \dots i_p},$$

$$i_t \in \{0, \dots, n_t - 1\}, \quad t \in \{1 \dots p\} / \{k\},$$

$$i_k \in \left\{0, \dots, \left\lfloor \frac{n_k}{f_k} \right\rfloor - 1\right\},$$

Here f_k is the kernel size of the pooled subtensor for the dimension k and s_k is the stride, that is, an interval on which each subtensor is selected.

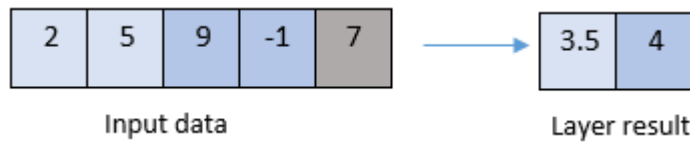
The size $m_1 \times m_2 \times \dots \times m_p$ of the output tensor of the forward 1D pooling layer Y is

$$m_t = \begin{cases} n_t, & t \in \{1, \dots, p\} \setminus \{k\} \\ \left\lfloor \frac{n_t}{f_t} \right\rfloor, & t = k \end{cases}$$

f_k cannot be greater than n_k .

Problem Statement

To perform average pooling, the problem is to compute the average for each subtensor and apply the transform to the input data:



Batch Processing

Layer Input

The forward one-dimensional average pooling layer accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
data	Pointer to the tensor of size $n_1 \times n_2 \times \dots \times n_p$ that stores the input data for the forward one-dimensional average pooling layer. This input can be an object of any class derived from <code>Tensor</code> .

Layer Parameters

For common parameters of neural network layers, see [Common Parameters](#).

In addition to the common parameters, the forward one-dimensional average pooling layer has the following parameters:

Parameter	Default Value	Description
<code>algorithmFPType</code>	<code>float</code>	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<code>method</code>	<code>defaultDense</code>	Performance-oriented computation method, the only method supported by the layer.
<code>kernelSize</code>	<code>KernelSize(2)</code>	Data structure representing the size of the one-dimensional subtensor from which the average element is computed.

Parameter	Default Value	Description
<i>stride</i>	Stride(2)	Data structure representing the intervals on which the subtensors for pooling are selected.
<i>padding</i>	Padding(0)	Data structure representing the number of data elements to implicitly add to each size of the one-dimensional subtensor on which pooling is performed.
<i>indices</i>	HomogenNumericTable(p-1)	Indices of the one dimensions on which pooling is performed, stored in HomogenNumericTable.

Layer Output

The forward one-dimensional average pooling layer calculates the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result				
value	Pointer to the tensor of size $m_1 \times m_2 \times \dots \times m_p$ that stores the result of the forward one-dimensional average pooling layer, where $m_i, i \in \{1 \dots p\}$ are defined above. This input can be an object of any class derived from <code>Tensor</code> .				
resultForBackward	Collection of data needed for the backward one-dimensional average pooling layer.				
	<table> <tr> <th>Element ID</th><th>Element</th></tr> <tr> <td>auxInputDimensions</td><td>Collection that contains the size of the dimensions of the input data tensor: n_1, n_2, \dots, n_p.</td></tr> </table>	Element ID	Element	auxInputDimensions	Collection that contains the size of the dimensions of the input data tensor: n_1, n_2, \dots, n_p .
Element ID	Element				
auxInputDimensions	Collection that contains the size of the dimensions of the input data tensor: n_1, n_2, \dots, n_p .				

Examples

Refer to the following examples in your Intel DAAL directory:

C++: `./examples/cpp/source/neural_networks/average_pooling1D_layer_batch.cpp`

Java*: `./examples/java/source/com/intel/daal/examples/neural_networks/AveragePooling1DLayerBatch.java`

Python*: `./examples/python/source/neural_networks/average_pooling1D_layer_batch.py`

1D Average Pooling Backward Layer

The forward one-dimensional (1D) average pooling layer is a form of non-linear downsampling of an input tensor $X = (x^{(1)} \dots x^{(p)})$ of size $n_1 \times n_2 \times \dots \times n_p$. For more details, see [Forward 1D Average Pooling Layer](#). The backward 1D average pooling layer back-propagates the input gradient $G = (g^{(1)} \dots g^{(p)})$ of size $m_1 \times m_2 \times \dots \times m_p$ computed on the preceding layer. The result of the backward 1D average pooling layer $Z = (z^{(1)} \dots z^{(p)})$ is the tensor of the same size $n_1 \times n_2 \times \dots \times n_p$ as the input of the forward computation. The backward layer propagates the elements of the gradient multiplied by the coefficient $1/f_k$ to the corresponding pooled subtensors of the tensor Z :

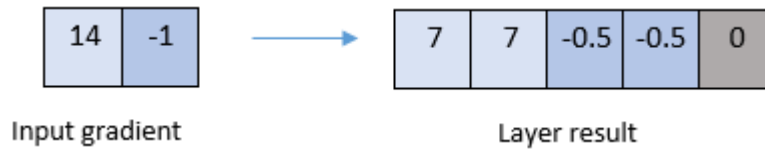
$$z_{i_1 \dots i_k \dots i_p} = \frac{1}{f_k} * g_{i_1 \dots j_k \dots i_p}, \quad j_k = \left\lfloor \frac{i_k}{f_k} \right\rfloor,$$

$$i_t \in \{0, \dots, n_t - 1\}, \quad t \in \{1 \dots p\} / \{k\},$$

$$j_k \in \left\{0, \dots, \left\lfloor \frac{n_k}{f_k} \right\rfloor - 1\right\}, i_k \in \{0, \dots, n_k - 1\}.$$

Problem Statement

Given a p -dimensional tensor $G \in \mathbb{R}^{n_1 \times n_2 \times \dots \times n_p}$ with the gradient computed on the preceding layer, the problem is to compute the p -dimensional tensor $Z \in \mathbb{R}^{n_1 \times n_2 \times \dots \times n_p}$:



Batch Processing

Layer Input

The backward one-dimensional average pooling layer accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input				
inputGradient	Pointer to the tensor of size $m_1 \times m_2 \times \dots \times m_p$ that stores input gradient g computed on the preceding layer. This input can be an object of any class derived from <code>Tensor</code> .				
inputFromForward	Collection of data needed for the backward one-dimensional average pooling layer.				
	<table> <tr> <th>Element ID</th><th>Element</th></tr> <tr> <td>auxInputDimensions</td><td>Collection that contains the size of the dimensions of the input data tensor in the forward computation step n_1, n_2, \dots, n_p.</td></tr> </table>	Element ID	Element	auxInputDimensions	Collection that contains the size of the dimensions of the input data tensor in the forward computation step n_1, n_2, \dots, n_p .
Element ID	Element				
auxInputDimensions	Collection that contains the size of the dimensions of the input data tensor in the forward computation step n_1, n_2, \dots, n_p .				

Layer Parameters

For common parameters of neural network layers, see [Common Parameters](#).

In addition to the common parameters, the backward one-dimensional average pooling layer has the following parameters:

Parameter	Default Value	Description
<code>algorithmFPType</code>	float	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<code>method</code>	defaultDense	Performance-oriented computation method, the only method supported by the layer.
<code>kernelSize</code>	KernelSize(2)	Data structure representing the size of the one-dimensional subtensor from which the average element is computed.

Parameter	Default Value	Description
<i>stride</i>	Stride(2)	Data structure representing the intervals on which the subtensors for pooling are selected.
<i>padding</i>	Padding(0)	Data structure representing the number of data elements to implicitly add to each size of the one-dimensional subtensor on which pooling is performed.
<i>indices</i>	HomogenNumericTable(p-1)	Indices of the one dimensions on which pooling is performed, stored in HomogenNumericTable.

Layer Output

The backward one-dimensional average pooling layer calculates the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
gradient	Pointer to the tensor of size $n_1 \times n_2 \times \dots \times n_p$ that stores the result of the backward one-dimensional average pooling layer. This input can be an object of any class derived from <code>Tensor</code> .

Examples

Refer to the following examples in your Intel DAAL directory:

C++: `./examples/cpp/source/neural_networks/average_pooling1D_layer_batch.cpp`

Java*: `./examples/java/source/com/intel/daal/examples/neural_networks/AveragePooling1DLayerBatch.java`

Python*: `./examples/python/source/neural_networks/average_pooling1D_layer_batch.py`

2D Average Pooling Forward Layer

The forward two-dimensional (2D) average pooling layer is a form of non-linear downsampling of an input tensor $X = (x^{(1)} \dots x^{(p)})$ of size $n_1 \times n_2 \times \dots \times n_p$. 2D average pooling partitions the input tensor data into 2D subtensors by two dimensions k_1 and k_2 , computes the average value of elements in each subtensor, and transforms the input tensor to the output tensor $Y = (y^{(1)} \dots y^{(p)})$ of size $m_1 \times m_2 \times \dots \times m_p$ by replacing each subtensor with one element, the average of the subtensor:

$$y_{i_1 \dots i_{k_1} \dots i_{k_2} \dots i_p} = \frac{1}{f_{k_1} * f_{k_2}} \sum_{\left(j_{k_1}, j_{k_2} \right) \in \left[s_{k_1} * i_{k_1}, s_{k_1} * i_{k_1} + f_{k_1} - 1 \right] \times \left[s_{k_2} * i_{k_2}, s_{k_2} * i_{k_2} + f_{k_2} - 1 \right]} x_{i_1 \dots j_{k_1} \dots j_{k_2} \dots i_p},$$

$$i_k \in \{0, \dots, n_k - 1\}, \quad k \in \{1 \dots p\} \quad / \quad \{k_1, k_2\},$$

$$i_k \in \left\{ 0, \dots, \left\lfloor \frac{n_k}{f_k} \right\rfloor - 1 \right\}, \quad k \in \{k_1, k_2\}.$$

Here f_k is the kernel size of the pooled subtensor for the dimension k and s_k is the stride, that is, an interval on which each subtensor is selected.

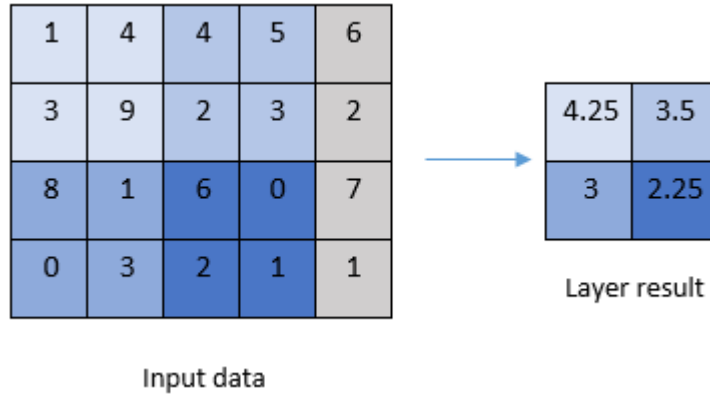
The size $m_1 \times m_2 \times \dots \times m_p$ of the output tensor of the forward 2D pooling layer Y is

$$m_k = \begin{cases} n_k, & k \in \{1, \dots, p\} / \{k_1, k_2\} \\ \left\lfloor \frac{n_k}{f_k} \right\rfloor, & k \in \{k_1, k_2\} \end{cases}$$

For $k \in \{k_1, k_2\}$, f_k cannot be greater than n_k .

Problem Statement

To perform average pooling, the problem is to compute the average for each subtensor and apply the transform to the input data:



Batch Processing

Layer Input

The forward two-dimensional average pooling layer accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
data	Pointer to the tensor of size $n_1 \times n_2 \times \dots \times n_p$ that stores the input data for the forward two-dimensional average pooling layer. This input can be an object of any class derived from <code>Tensor</code> .

Layer Parameters

For common parameters of neural network layers, see [Common Parameters](#).

In addition to the common parameters, the forward two-dimensional average pooling layer has the following parameters:

Parameter	Default Value	Description
<code>algorithmFPType</code>	<code>float</code>	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<code>method</code>	<code>defaultDense</code>	Performance-oriented computation method, the only method supported by the layer.
<code>kernelSizes</code>	<code>KernelSizes(2, 2)</code>	Data structure representing the size of the two-dimensional subtensor from which the average element is computed.

Parameter	Default Value	Description
<i>strides</i>	Strides(2, 2)	Data structure representing the intervals on which the subtensors for pooling are selected.
<i>padding</i> s	Paddings(0, 0)	Data structure representing the number of data elements to implicitly add to each size of the two-dimensional subtensor on which pooling is performed.
<i>indices</i>	HomogenNumericTable(p-2, p-1)	Indices of the two dimensions on which pooling is performed, stored in HomogenNumericTable.

Layer Output

The forward two-dimensional average pooling layer calculates the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result				
value	Pointer to the tensor of size $m_1 \times m_2 \times \dots \times m_p$ that stores the result of the forward two-dimensional average pooling layer, where $m_i, i \in \{1 \dots p\}$ are defined above. This input can be an object of any class derived from <code>Tensor</code> .				
resultForBackward	Collection of data needed for the backward two-dimensional average pooling layer.				
	<table> <tr> <th>Element ID</th><th>Element</th></tr> <tr> <td>auxInputDimensions</td><td>Collection that contains the size of the dimensions of the input data tensor: n_1, n_2, \dots, n_p.</td></tr> </table>	Element ID	Element	auxInputDimensions	Collection that contains the size of the dimensions of the input data tensor: n_1, n_2, \dots, n_p .
Element ID	Element				
auxInputDimensions	Collection that contains the size of the dimensions of the input data tensor: n_1, n_2, \dots, n_p .				

Examples

Refer to the following examples in your Intel DAAL directory:

C++: `./examples/cpp/source/neural_networks/average_pooling2d_layer_batch.cpp`

Java*: `./examples/java/source/com/intel/daal/examples/neural_networks/AveragePooling2DLayerBatch.java`

Python*: `./examples/python/source/neural_networks/average_pooling2d_layer_batch.py`

2D Average Pooling Backward Layer

The forward two-dimensional (2D) average pooling layer is a form of non-linear downsampling of an input tensor $X = (x^{(1)} \dots x^{(p)})$ of size $n_1 \times n_2 \times \dots \times n_p$. For more details, see [Forward 2D Average Pooling Layer](#). The backward 2D average pooling layer back-propagates the input gradient $G = (g^{(1)} \dots g^{(p)})$ of size $m_1 \times m_2 \times \dots \times m_p$ computed on the preceding layer. The result of the backward 2D average pooling layer $Z = (z^{(1)} \dots z^{(p)})$ is the tensor of the same size $n_1 \times n_2 \times \dots \times n_p$ as the input of the forward computation. The backward layer propagates the elements of the gradient multiplied by the coefficient $1/(f_{k_1} * f_{k_2})$ to the corresponding pooled subtensors of the tensor Z :

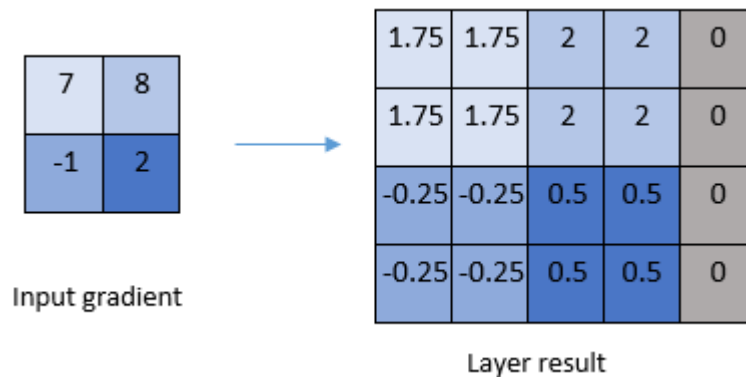
$$z_{i_1 \dots i_{k_1} \dots i_{k_2} \dots i_p} = \frac{1}{f_{k_1} * f_{k_2}} * g_{i_1 \dots j_{k_1} \dots j_{k_2} \dots i_p}, j_{k_l} = \left\lfloor \frac{i_{k_l}}{f_{k_l}} \right\rfloor, l \in \{1, 2\},$$

$$i_k \in \{0, \dots, n_k - 1\}, \quad k \in \{1 \dots p\} / \{k_1, \quad k_2\},$$

$$j_k \in \left\{0, \quad \dots, \quad \left\lfloor \frac{n_k}{f_k} \right\rfloor - 1\right\}, \quad i_k \in \{0, \quad \dots, n_k - 1\}, \quad k \in \{k_1, \quad k_2\}.$$

Problem Statement

Given a p -dimensional tensor $G \in R^{n_1 \times n_2 \times \dots \times n_p}$ with the gradient computed on the preceding layer, the problem is to compute the p -dimensional tensor $Z \in R^{n_1 \times n_2 \times \dots \times n_p}$ with the result:



Batch Processing

Layer Input

The backward two-dimensional average pooling layer accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input				
inputGradient	Pointer to the tensor of size $m_1 \times m_2 \times \dots \times m_p$ that stores the input gradient g computed on the preceding layer. This input can be an object of any class derived from <code>Tensor</code> .				
inputFromForward	Collection of data needed for the backward two-dimensional average pooling layer.				
	<table> <tr> <th>Element ID</th><th>Element</th></tr> <tr> <td>auxInputDimensions</td><td>Collection that contains the size of the dimensions of the input data tensor in the forward computation step: n_1, n_2, \dots, n_p.</td></tr> </table>	Element ID	Element	auxInputDimensions	Collection that contains the size of the dimensions of the input data tensor in the forward computation step: n_1, n_2, \dots, n_p .
Element ID	Element				
auxInputDimensions	Collection that contains the size of the dimensions of the input data tensor in the forward computation step: n_1, n_2, \dots, n_p .				

Layer Parameters

For common parameters of neural network layers, see [Common Parameters](#).

In addition to the common parameters, the backward two-dimensional average pooling layer has the following parameters:

Parameter	Default Value	Description
<i>algorithmFPTYPE</i>	float	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<i>method</i>	defaultDense	Performance-oriented computation method, the only method supported by the layer.
<i>kernelSizes</i>	KernelSizes(2, 2)	Data structure representing the size of the two-dimensional subtensor from which the average element is computed.
<i>strides</i>	Strides(2, 2)	Data structure representing the intervals on which the subtensors for pooling are selected.
<i>padding</i> s	Paddings(0, 0)	Data structure representing the number of data elements to implicitly add to each size of the two-dimensional subtensor on which pooling is performed.
<i>indices</i>	HomogenNumericTable(p-2, p-1)	Indices of the two dimensions on which pooling is performed, stored in <code>HomogenNumericTable</code> .

Layer Output

The backward two-dimensional average pooling layer calculates the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
gradient	Pointer to the tensor of size $n_1 \times n_2 \times \dots \times n_p$ that stores the result of the backward two-dimensional average pooling layer. This input can be an object of any class derived from <code>Tensor</code> .

Examples

Refer to the following examples in your Intel DAAL directory:

C++: `./examples/cpp/source/neural_networks/average_pooling2d_layer_batch.cpp`

Java*: `./examples/java/source/com/intel/daal/examples/neural_networks/AveragePooling2DLayerBatch.java`

Python*: `./examples/python/source/neural_networks/average_pooling2d_layer_batch.py`

3D Average Pooling Forward Layer

The forward three-dimensional (3D) average pooling layer is a form of non-linear downsampling of an input tensor $X = (x^{(1)} \dots x^{(p)})$ of size $n_1 \times n_2 \times \dots \times n_p$. 3D average pooling partitions the input tensor data into 3D subtensors by three dimensions $k_1 k_2$, and k_3 , computes the average value of elements in each subtensor, and transforms the input tensor to the output tensor $Y = (y^{(1)} \dots y^{(p)})$ of size $m_1 \times m_2 \times \dots \times m_p$ by replacing each subtensor with one element, the average of the subtensor:

$$y_{i_1 \dots i_{k_1} \dots i_{k_2} \dots i_{k_3} \dots i_p} = \frac{1}{f_{k_1} * f_{k_2} * f_{k_3}} \sum_{\left(j_{k_1}, j_{k_2}, j_{k_3} \right) \in d_{k_1} \times d_{k_2} \times d_{k_3}} x_{i_1 \dots j_{k_1} \dots j_{k_2} \dots j_{k_3} \dots i_p},$$

$$d_{k_c} = \left[s_{k_c} * l_{k_c}, s_{k_c} * l_{k_c} + f_{k_c} - 1 \right], c \in \{1, 2, 3\},$$

$$i_k \in \{0, \dots, n_k - 1\}, k \in \{1 \dots p\} / \{k_1, k_2, k_3\},$$

$$i_k \in \left\{ 0, \dots, \left\lfloor \frac{n_k}{f_k} \right\rfloor - 1 \right\}, k \in \{k_1, k_2, k_3\}.$$

Here f_k is the kernel size of the pooled subtensor for the dimension k and s_k is the stride, that is, an interval on which each subtensor is selected.

The size $m_1 \times m_2 \times \dots \times m_p$ of the output tensor of the forward 3D pooling layer Y is

$$m_k = \begin{cases} n_k, & k \in \{1, \dots, p\} / \{k_1, k_2, k_3\} \\ \left\lfloor \frac{n_k}{f_k} \right\rfloor, & k \in \{k_1, k_2, k_3\} \end{cases}.$$

For $k \in \{k_1, k_2, k_3\}$, f_k cannot be greater than n_k .

Problem Statement

To perform average pooling, the problem is to compute the average for each subtensor and apply the transform to the input data.

Batch Processing

Layer Input

The forward three-dimensional average pooling layer accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
data	Pointer to the tensor of size $n_1 \times n_2 \times \dots \times n_p$ that stores the input data for the forward three-dimensional average pooling layer. This input can be an object of any class derived from <code>Tensor</code> .

Layer Parameters

For common parameters of neural network layers, see [Common Parameters](#).

In addition to the common parameters, the forward three-dimensional average pooling layer has the following parameters:

Parameter	Default Value	Description
<code>algorithmFPType</code>	<code>float</code>	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<code>method</code>	<code>defaultDense</code>	Performance-oriented computation method, the only method supported by the layer.

Parameter	Default Value	Description
<i>kernelSizes</i>	<code>KernelSizes(2, 2, 2)</code>	Data structure representing the size of the three-dimensional subtensor from which the average element is computed.
<i>strides</i>	<code>Strides(2, 2, 2)</code>	Data structure representing the intervals on which the subtensors for pooling are selected.
<i>padding</i> s	<code>Paddings(0, 0, 0)</code>	Data structure representing the number of data elements to implicitly add to each size of the three-dimensional subtensor on which pooling is performed.
<i>indices</i>	<code>HomogenNumericTable(p-3, p-2, p-1)</code>	Indices of the three dimensions on which pooling is performed, stored in <code>HomogenNumericTable</code> .

Layer Output

The forward three-dimensional average pooling layer calculates the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result				
value	Pointer to the tensor of size $m_1 \times m_2 \times \dots \times m_p$ that stores the result of the forward three-dimensional average pooling layer, where $m_i, i \in \{1 \dots p\}$ are defined above. This input can be an object of any class derived from <code>Tensor</code> .				
resultForBackward	Collection of data needed for the backward three-dimensional average pooling layer.				
	<table> <tr> <th>Element ID</th><th>Element</th></tr> <tr> <td>auxInputDimensions</td><td>Collection that contains the size of the dimensions of the input data tensor: n_1, n_2, \dots, n_p.</td></tr> </table>	Element ID	Element	auxInputDimensions	Collection that contains the size of the dimensions of the input data tensor: n_1, n_2, \dots, n_p .
Element ID	Element				
auxInputDimensions	Collection that contains the size of the dimensions of the input data tensor: n_1, n_2, \dots, n_p .				

Examples

Refer to the following examples in your Intel DAAL directory:

C++: `./examples/cpp/source/neural_networks/average_pooling3d_layer_batch.cpp`

Java*: `./examples/java/source/com/intel/daal/examples/neural_networks/AveragePooling3DLayerBatch.java`

Python*: `./examples/python/source/neural_networks/average_pooling3d_layer_batch.py`

3D Average Pooling Backward Layer

The forward three-dimensional (3D) average pooling layer is a form of non-linear downsampling of an input tensor $X = (x^{(1)} \dots x^{(p)})$ of size $n_1 \times n_2 \times \dots \times n_p$. For more details, see [Forward 3D Average Pooling Layer](#). The backward 3D average pooling layer back-propagates the input gradient $G = (g^{(1)} \dots g^{(p)})$ of size $m_1 \times m_2 \times \dots \times m_p$ computed on the preceding layer. The result of the backward 3D average pooling layer $Z = (z^{(1)} \dots$

$z^{(p)}$) is the tensor of the same size $n_1 \times n_2 \times \dots \times n_p$ as the input of the forward computation. The backward layer propagates the elements of the gradient multiplied by the coefficient $1/(f_{k_1} * f_{k_2} * f_{k_3})$ to the corresponding pooled subensors of the tensor Z :

$$z_{i_1 \dots i_{k_1} \dots i_{k_2} \dots i_{k_3} \dots i_p} = \frac{1}{f_{k_1} * f_{k_2} * f_{k_3}} * g_{i_1 \dots j_{k_1} \dots j_{k_2} \dots j_{k_3} \dots i_p}, \quad j_{k_l} = \left\lfloor \frac{i_{k_l}}{f_{k_l}} \right\rfloor, \quad l \in \{1, 2, 3\},$$

$$i_k \in \{0, \dots, n_k - 1\}, \quad k \in \{1 \dots p\} \setminus \{k_1, k_2, k_3\},$$

$$j_k \in \left\{0, \dots, \left\lfloor \frac{n_k}{f_k} \right\rfloor - 1\right\}, \quad i_k \in \{0, \dots, n_k - 1\}, \quad k \in \{k_1, k_2, k_3\}.$$

Problem Statement

Given a p -dimensional tensor $G \in \mathbb{R}^{n_1 \times n_2 \times \dots \times n_p}$ with the gradient computed on the preceding layer, the problem is to compute the p -dimensional tensor $Z \in \mathbb{R}^{n_1 \times n_2 \times \dots \times n_p}$ with the result.

Batch Processing

Layer Input

The backward three-dimensional average pooling layer accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input				
inputGradient	Pointer to the tensor of size $m_1 \times m_2 \times \dots \times m_p$ that stores input gradient g computed on the preceding layer. This input can be an object of any class derived from <code>Tensor</code> .				
inputFromForward	Collection of data needed for the backward three-dimensional average pooling layer.				
	<table> <tr> <th>Element ID</th><th>Element</th></tr> <tr> <td>auxInputDimensions</td><td>Collection that contains the size of the dimensions of the input data tensor in the forward computation step: n_1, n_2, \dots, n_p.</td></tr> </table>	Element ID	Element	auxInputDimensions	Collection that contains the size of the dimensions of the input data tensor in the forward computation step: n_1, n_2, \dots, n_p .
Element ID	Element				
auxInputDimensions	Collection that contains the size of the dimensions of the input data tensor in the forward computation step: n_1, n_2, \dots, n_p .				

Layer Parameters

For common parameters of neural network layers, see [Common Parameters](#).

In addition to the common parameters, the backward three-dimensional average pooling layer has the following parameters:

Parameter	Default Value	Description
algorithmFPType	float	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
method	defaultDense	Performance-oriented computation method, the only method supported by the layer.

Parameter	Default Value	Description
<i>kernelSizes</i>	<code>KernelSizes(2, 2, 2)</code>	Data structure representing the size of the three-dimensional subtensor from which the average element is computed.
<i>strides</i>	<code>Strides(2, 2, 2)</code>	Data structure representing the intervals on which the subtensors for pooling are selected.
<i>padding</i> s	<code>Paddings(0, 0, 0)</code>	Data structure representing the number of data elements to implicitly add to each size of the three-dimensional subtensor on which pooling is performed.
<i>indices</i>	<code>HomogenNumericTable(p-3, p-2, p-1)</code>	Indices of the three dimensions on which pooling is performed, stored in <code>HomogenNumericTable</code> .

Layer Output

The backward three-dimensional average pooling layer calculates the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
gradient	Pointer to the tensor of size $n_1 \times n_2 \times \dots \times n_p$ that stores the result of the backward three-dimensional average pooling layer. This input can be an object of any class derived from <code>Tensor</code> .

Examples

Refer to the following examples in your Intel DAAL directory:

C++: `./examples/cpp/source/neural_networks/average_pooling3d_layer_batch.cpp`

Java*: `./examples/java/source/com/intel/daal/examples/neural_networks/AveragePooling3DLayerBatch.java`

Python*: `./examples/python/source/neural_networks/average_pooling3d_layer_batch.py`

2D Stochastic Pooling Forward Layer

The forward two-dimensional (2D) stochastic pooling layer is a form of non-linear downsampling of an input tensor $X \in \mathbb{R}^{n_1 \times n_2 \times \dots \times n_p}$ with non-negative elements $x_{i_1 \dots i_p}$. 2D stochastic pooling partitions the input tensor data into 2D subtensors along dimensions k_1 and k_2 and applies the transformations (kernels).

At the training stage, the layer selects an element in each subtensor using sampling from a multinomial distribution. Probabilities required in the distribution are calculated by normalizing the subtensor. In the output, the selected element replaces the entire subtensor.

At the prediction stage, the layer replaces each subtensor with the weighted average of its elements using the probabilities as weights. For more details, see [\[Matthew2013\]](#).

Problem Statement

Given:

- The tensor $X \in \mathbb{R}^{n_1 \times n_2 \times \dots \times n_p}$ of input data with non-negative elements $x_{i_1 \dots i_p}$
- Dimensions k_1 and k_2 along which kernels are applied
- Kernel sizes m_1 and m_2 : $m_i \leq n_i + 2 \cdot p_i$, $i \in \{1, 2\}$, where p_1 and p_2 are paddings

The problem is to compute the value tensor $Y \in \mathbb{R}^{l_1 \times \dots \times l_p}$ using the downsampling technique, where:

$$l_i = \begin{cases} n_i, & i \notin \{k_1, k_2\} \\ \frac{n_{k_j} + 2p_j - m_j}{s_j} + 1, & i = k_j, \quad j \in \{1, 2\} \end{cases}$$

and s_1 and s_2 are strides.

To define the values of Y , let's apply the kernel to the subtensor at position $\{i_1, \dots, i_p\}$ and consider elements of the subtensor as the vector a of size $M = m_1 \cdot m_2$ with elements

$$a_j = x_{i_1 \dots \left(i_{k_1} \cdot s_1 - p_1 + j_1\right) \dots \left(i_{k_2} \cdot s_2 - p_2 + j_2\right) \dots i_p},$$

$$j \in \{0, \dots, M-1\}, \quad j_1 = \left\lfloor \frac{j}{m_2} \right\rfloor, j_2 = j - j_1 \cdot m_2.$$

The probabilities used in the multinomial distribution are defined as:

$$pr_j = \frac{a_j}{\sum_{i=0}^{M-1} a_i}.$$

Training Stage

At the training stage, the layer computes the value y as the element at the position t in the subtensor. The position t is picked as a sample from the multinomial distribution and kept for use in the backward step:

$$y_{i_1 \dots i_p} = a_t,$$

$$s_{i_1 \dots i_p} = t.$$

Prediction Stage

At the prediction stage, the layer computes the value as a weighted average of elements of the vector a that represents the subtensor:

$$y_{i_1 \dots i_p} = \sum_{j=0}^{M-1} pr_j \cdot a_j.$$

Batch Processing

Layer Input

The forward two-dimensional stochastic pooling layer accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
<code>data</code>	Pointer to tensor X of size $n_1 \times \dots \times n_p$ that stores the non-negative input data for the forward two-dimensional stochastic pooling layer. This input can be an object of any class derived from <code>Tensor</code> .
NOTE If you provide the input data tensor with negative elements, the layer algorithm returns unpredicted results.	

Layer Parameters

For common parameters of neural network layers, see [Common Parameters](#).

In addition to the common parameters, the forward two-dimensional stochastic pooling layer has the following parameters:

Parameter	Default Value	Description
<code>algorithmFPType</code>	<code>float</code>	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<code>method</code>	<code>defaultDense</code>	Performance-oriented computation method, the only method supported by the layer.
<code>kernelSizes</code>	<code>KernelSizes(2, 2)</code>	Data structure representing sizes m_1, m_2 of two-dimensional tensor K .
<code>strides</code>	<code>Strides(2, 2)</code>	Data structure representing intervals s_1, s_2 on which the subtensors for stochastic pooling are selected.
<code>padding</code>	<code>Paddings(0, 0)</code>	Data structure representing numbers p_1, p_2 of data elements to implicitly add to each side of the two-dimensional subtensor along which stochastic pooling is performed.
<code>indices</code>	<code>Indices(p-2, p-1)</code>	Indices k_1, k_2 of the dimensions along which stochastic pooling is performed.
<code>predictionStage</code>	<code>false</code>	Flag that specifies whether the layer is used for the prediction stage.

Parameter	Default Value	Description
<code>seed</code>	777	Seed for multinomial random number generator.

Layer Output

The forward two-dimensional stochastic pooling layer calculates the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result						
<code>value</code>	Pointer to tensor Y of size $l_1 \times \dots \times l_p$ that stores the result of the forward two-dimensional stochastic pooling layer. This input can be an object of any class derived from <code>Tensor</code> .						
<code>resultForBackward</code>	Collection of data needed for the backward two-dimensional stochastic pooling layer. <table> <tr> <th>Element ID</th><th>Element</th></tr> <tr> <td><code>auxSelectedIndices</code></td><td>Tensor S of size $l_1 \times \dots \times l_p$ that stores positions of selected elements.</td></tr> <tr> <td><code>auxInputDimensions</code></td><td><code>NumericTable</code> of size $1 \times p$ that stores the sizes of the dimensions of input data tensor X: n_1, n_2, \dots, n_p.</td></tr> </table>	Element ID	Element	<code>auxSelectedIndices</code>	Tensor S of size $l_1 \times \dots \times l_p$ that stores positions of selected elements.	<code>auxInputDimensions</code>	<code>NumericTable</code> of size $1 \times p$ that stores the sizes of the dimensions of input data tensor X : n_1, n_2, \dots, n_p .
Element ID	Element						
<code>auxSelectedIndices</code>	Tensor S of size $l_1 \times \dots \times l_p$ that stores positions of selected elements.						
<code>auxInputDimensions</code>	<code>NumericTable</code> of size $1 \times p$ that stores the sizes of the dimensions of input data tensor X : n_1, n_2, \dots, n_p .						

Examples

Refer to the following examples in your Intel DAAL directory:

C++: `./examples/cpp/source/neural_networks/stochastic_pooling2d_layer_batch.cpp`

Java*: `./examples/java/source/com/intel/daal/examples/neural_networks/StochasticPooling2DLayerBatch.java`

Python*: `./examples/python/source/neural_networks/stochastic_pooling2d_layer_batch.py`

2D Stochastic Pooling Backward Layer

At the training stage, the forward two-dimensional (2D) stochastic pooling layer is a form of non-linear downsampling of an input tensor $X \in R^{n_1 \times n_2 \times \dots \times n_p}$ with non-negative elements x_{i_1, \dots, i_p} . The layer partitions the input tensor data into 2D subtensors along dimensions k_1 and k_2 and selects an element in each subtensor using sampling from a multinomial distribution. Probabilities required in the distribution are calculated by normalizing the subtensor. In the output, the selected element replaces the entire subtensor. For more details, see [Forward 2D Stochastic Pooling Layer](#).

The backward 2D stochastic pooling layer computes the derivatives of the objective function E as the sum of input gradients that correspond to the elements pooled from subtensors in the forward step.

Problem Statement

Given:

- The tensor $G \in R^{l_1 \times \dots \times l_p}$ with the input gradient
- Dimensions k_1 and k_2 along which kernels are applied
- Kernel sizes m_1 and m_2 :

$$m_i \leq n_i + 2 \cdot p_i, i \in \{1, 2\},$$

where p_1 and p_2 are paddings

The problem is to compute the value tensor $Z \in \mathbb{R}^{n_1 \times \dots \times n_p}$ as follows:

$$z_{i_1 \dots i_p} = \frac{\partial E}{\partial x_{i_1 \dots i_p}} = \sum_{j_{k_1}, a \in V_1(i_{k_1})} \sum_{j_{k_2}, b \in V_2(i_{k_2})} \delta_{(a \cdot m_2 + b), t} g_{i_1 \dots j_{k_1} \dots j_{k_2} \dots i_p},$$

where:

- $\delta_{i,j} = \begin{cases} 1, & i = j \\ 0, & i \neq j \end{cases}$
- $t = s_{i_1 \dots j_{k_1} \dots j_{k_2} \dots i_p}$
- $V_h(i) = \left\{ \{j, a\} : \frac{j + p_h - a}{s_h} = i, \quad j \in \{1, \dots, l_{k_h}\}, \quad a \in \{1, \dots, m_h\} \right\}, \quad h \in \{1, 2\}$
- $l_i = \begin{cases} n_i, & i \notin \{k_1, k_2\} \\ \frac{n_{k_j} + 2p_j - m_j}{s_j} + 1, & i = k_j, \quad j \in \{1, 2\} \end{cases}$
- s_1 and s_2 are strides

Batch Processing

Layer Input

The backward two-dimensional stochastic pooling layer accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input						
inputGradient	Pointer to tensor G of size $l_1 \times \dots \times l_p$ that stores the input gradient computed on the preceding layer. This input can be an object of any class derived from <code>Tensor</code> .						
inputFromForward	Collection of data passed from the forward two-dimensional stochastic pooling layer.						
	<table> <tr> <th>Element ID</th><th>Element</th></tr> <tr> <td>auxSelectedIndices</td><td>Tensor S of size $l_1 \times \dots \times l_p$ that stores positions of selected elements.</td></tr> <tr> <td>auxInputDimensions</td><td>NumericTable of size $1 \times p$ that stores the sizes of the dimensions of input data tensor X: n_1, n_2, \dots, n_p.</td></tr> </table>	Element ID	Element	auxSelectedIndices	Tensor S of size $l_1 \times \dots \times l_p$ that stores positions of selected elements.	auxInputDimensions	NumericTable of size $1 \times p$ that stores the sizes of the dimensions of input data tensor X : n_1, n_2, \dots, n_p .
Element ID	Element						
auxSelectedIndices	Tensor S of size $l_1 \times \dots \times l_p$ that stores positions of selected elements.						
auxInputDimensions	NumericTable of size $1 \times p$ that stores the sizes of the dimensions of input data tensor X : n_1, n_2, \dots, n_p .						

Layer Parameters

For common parameters of neural network layers, see [Common Parameters](#).

In addition to the common parameters, the backward two-dimensional stochastic pooling layer has the following parameters:

Parameter	Default Value	Description
<i>algorithmFPTYPE</i>	float	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<i>method</i>	defaultDense	Performance-oriented computation method, the only method supported by the layer.
<i>kernelSizes</i>	KernelSizes(2, 2)	Data structure representing sizes m_1, m_2 of two-dimensional tensor K .
<i>strides</i>	Strides(2, 2)	Data structure representing intervals s_1, s_2 on which the subtensors for stochastic pooling are selected.
<i>padding</i> s	Padding(0, 0)	Data structure representing numbers p_1, p_2 of data elements to implicitly add to each side of the two-dimensional subtensor along which stochastic pooling is performed.
<i>indices</i>	Indices(p-2, p-1)	Indices k_1, k_2 of the dimensions along which stochastic pooling is performed.
<i>predictionStage</i>	false	Flag that specifies whether the layer is used for the prediction stage.
<i>seed</i>	777	Seed for multinomial random number generator.

Layer Output

The backward two-dimensional stochastic pooling layer calculates the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
gradient	Pointer to tensor Z of size $n_1 \times \dots \times n_p$ that stores the result of the backward two-dimensional stochastic pooling layer. This input can be an object of any class derived from <code>Tensor</code> .

Examples

Refer to the following examples in your Intel DAAL directory:

C++: `./examples/cpp/source/neural_networks/stochastic_pooling2d_layer_batch.cpp`

Java*: `./examples/java/source/com/intel/daal/examples/neural_networks/
StochasticPooling2DLayerBatch.java`

Python*: `./examples/python/source/neural_networks/stochastic_pooling2d_layer_batch.py`

2D Spatial Pyramid Pooling Forward Layer

The forward two-dimensional (2D) spatial pyramid pooling layer with pyramid height $L \in \mathbb{N}$ is a form of non-linear downsampling of an input tensor X . The library supports four-dimensional input tensors $X \in \mathbb{R}^{n_1 \times n_2 \times n_3 \times n_4}$. 2D spatial pyramid pooling partitions the input tensor data into $(2^l)^2$ subtensors/bins, $l \in \{0, \dots, L-1\}$, along dimensions k_1 and k_2 and computes the result in each subtensor. The computation is done according to the selected pooling strategy: maximum, average, or stochastic. The spatial pyramid pooling layer applies the pooling L times with different kernel sizes, strides, and paddings.

Problem Statement

The library provides several spatial pyramid pooling layers:

- Spatial pyramid maximum pooling
- Spatial pyramid average pooling
- Spatial pyramid stochastic pooling

The following description applies to each of these layers.

Let $X \in \mathbb{R}^{n_1 \times n_2 \times n_3 \times n_4}$ be the tensor of input data and k_1 and k_2 be the dimensions along which kernels are applied. Without loss of generality k_1 and k_2 are the last dimensions of the tensor X . For each level $l \in \{0, \dots, L-1\}$ and number of bins $b = 2^l$, the layer applies 2D pooling with parameters:

- Kernel sizes

$$m_i = \left\lceil \frac{n_{k_i}}{b} \right\rceil$$

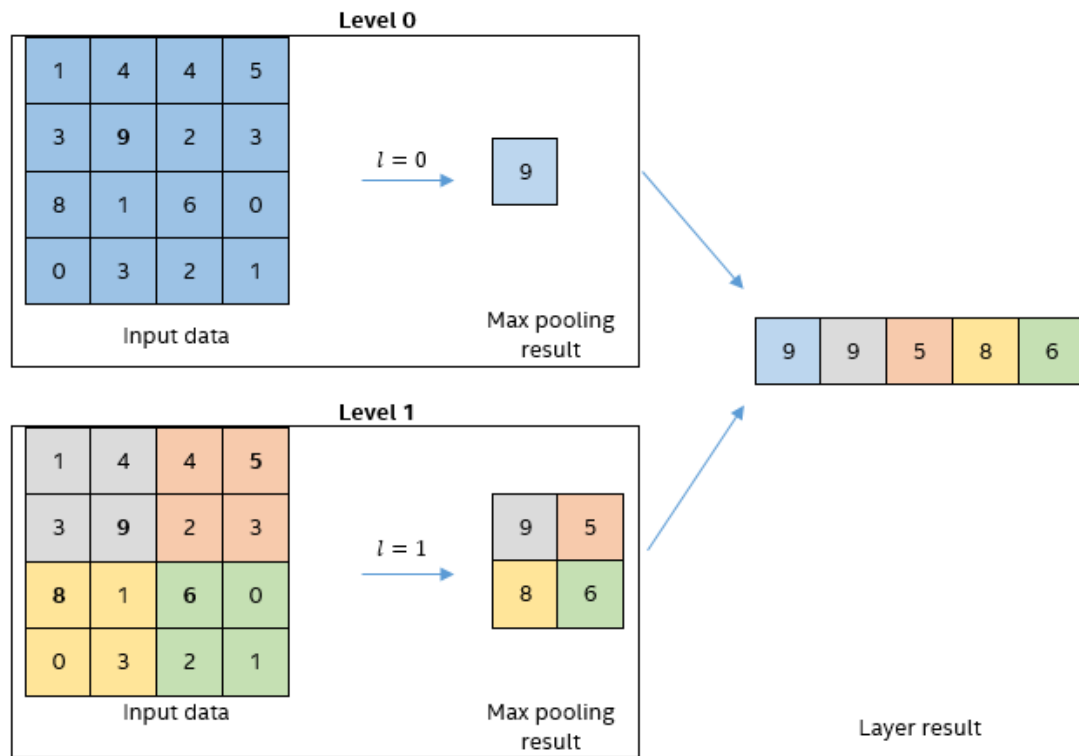
- Strides $s_i = m_i$
- Paddings

$$p_i = \left\lfloor \frac{m_i \cdot b - n_{k_i} + 1}{2} \right\rfloor, i \in \{1, 2\}$$

In the layout flattened along the dimension n' , the layer result is represented as a two-dimensional tensor Y

$\in \mathbb{R}^{n_1 \times n'}$, where $n' = n_2 \cdot \left(2^0 + (2^1)^2 + \dots + (2^{L-1})^2\right) = \frac{n_2}{3}(4^L - 1)$.

The following figure illustrates the behavior of the spatial pyramid maximum pooling forward layer with pyramid height $L = 2$:



Batch Processing

Layer Input

The forward two-dimensional spatial pyramid pooling layer accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
data	Pointer to tensor X of size $n_1 \times n_2 \times n_3 \times n_4$ that stores the input data for the forward two-dimensional spatial pyramid pooling layer. This input can be an object of any class derived from <code>Tensor</code> .

Layer Parameters

For common parameters of neural network layers, see [Common Parameters](#).

In addition to the common parameters, the forward two-dimensional spatial pyramid pooling layer has the following parameters:

Algorithm	Parameter	Default Value	Description
any	<i>algorithmFPType</i>	float	The floating-point type that the algorithm uses for intermediate computations. Can be float or double.
any	<i>method</i>	defaultDense	Performance-oriented computation method, the only method supported by the layer.

Algorithm	Parameter	Default Value	Description
any	<i>pyramidHeight</i>	Not applicable	Value L of the pyramid height.
any	<i>indices</i>	Indices(p-2, p-1)	Indices k_1, k_2 of the dimensions along which spatial pyramid pooling is performed.
spatial pyramid stochastic pooling	<i>predictionStage</i>	false	Flag that specifies whether the layer is used at the prediction stage.
spatial pyramid stochastic pooling	<i>seed</i>	777	Seed for multinomial random number generator.

Layer Output

The forward two-dimensional spatial pyramid pooling layer calculates the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result									
value	Pointer to tensor Y of size $n_1 \times n'$ that stores the result of the forward two-dimensional spatial pyramid pooling layer. This input can be an object of any class derived from <code>Tensor</code> .									
resultForBackward	Collection of data needed for the backward two-dimensional spatial pyramid pooling layer. <table><tr><th>Algorithm</th><th>Element ID</th><th>Element</th></tr><tr><td>any</td><td>auxInputDimensions</td><td>NumericTable of size $1 \times p$ that stores the sizes of the dimensions of input data tensor X: n_1, n_2, \dots, n_p.</td></tr><tr><td>spatial pyramid max pooling, spatial pyramid stochastic pooling</td><td>auxSelectedIndices</td><td>Tensor T of size $n_1 \times n'$ that stores indices of maximum/stochastic elements.</td></tr></table>	Algorithm	Element ID	Element	any	auxInputDimensions	NumericTable of size $1 \times p$ that stores the sizes of the dimensions of input data tensor X : n_1, n_2, \dots, n_p .	spatial pyramid max pooling, spatial pyramid stochastic pooling	auxSelectedIndices	Tensor T of size $n_1 \times n'$ that stores indices of maximum/stochastic elements.
Algorithm	Element ID	Element								
any	auxInputDimensions	NumericTable of size $1 \times p$ that stores the sizes of the dimensions of input data tensor X : n_1, n_2, \dots, n_p .								
spatial pyramid max pooling, spatial pyramid stochastic pooling	auxSelectedIndices	Tensor T of size $n_1 \times n'$ that stores indices of maximum/stochastic elements.								

Examples

Refer to the following examples in your Intel DAAL directory:

C++:

- `./examples/cpp/source/neural_networks/spat_ave_pool2d_layer_dense_batch.cpp`
- `./examples/cpp/source/neural_networks/spat_max_pool2d_layer_dense_batch.cpp`
- `./examples/cpp/source/neural_networks/spat_stoch_pool2d_layer_dense_batch.cpp`

Java*:

- `./examples/java/source/com/intel/daal/examples/neural_networks/SpatAvePool2DLayerDenseBatch.java`

- `./examples/java/source/com/intel/daal/examples/neural_networks/SpatMaxPool2DLayerDenseBatch.java`
- `./examples/java/source/com/intel/daal/examples/neural_networks/SpatStochPool2DLayerDenseBatch.java`

Python*:

- `./examples/python/source/neural_networks/spat_ave_pool2d_layer_dense_batch.py`
- `./examples/python/source/neural_networks/spat_max_pool2d_layer_dense_batch.py`
- `./examples/python/source/neural_networks/spat_stoch_pool2d_layer_dense_batch.py`

2D Spatial Pyramid Pooling Backward Layer

The forward two-dimensional (2D) spatial pyramid pooling layer with pyramid height $L \in \mathbb{N}$ is a form of non-linear downsampling of an input tensor X . For more details, see [Forward 2D Spatial Pyramid Pooling Layer](#).

The backward 2D spatial pyramid pooling layer computes the input gradient $G \in \mathbb{R}^{n_1 \times n'}$, where n' is computed on the preceding layer as explained for the forward layer. Only the elements corresponding to the values pooled from subensors in the forward computation step are propagated to the next layer.

Problem Statement

The library provides several spatial pyramid pooling layers:

- Spatial pyramid maximum pooling
- Spatial pyramid average pooling
- Spatial pyramid stochastic pooling

The following description applies to each of these layers.

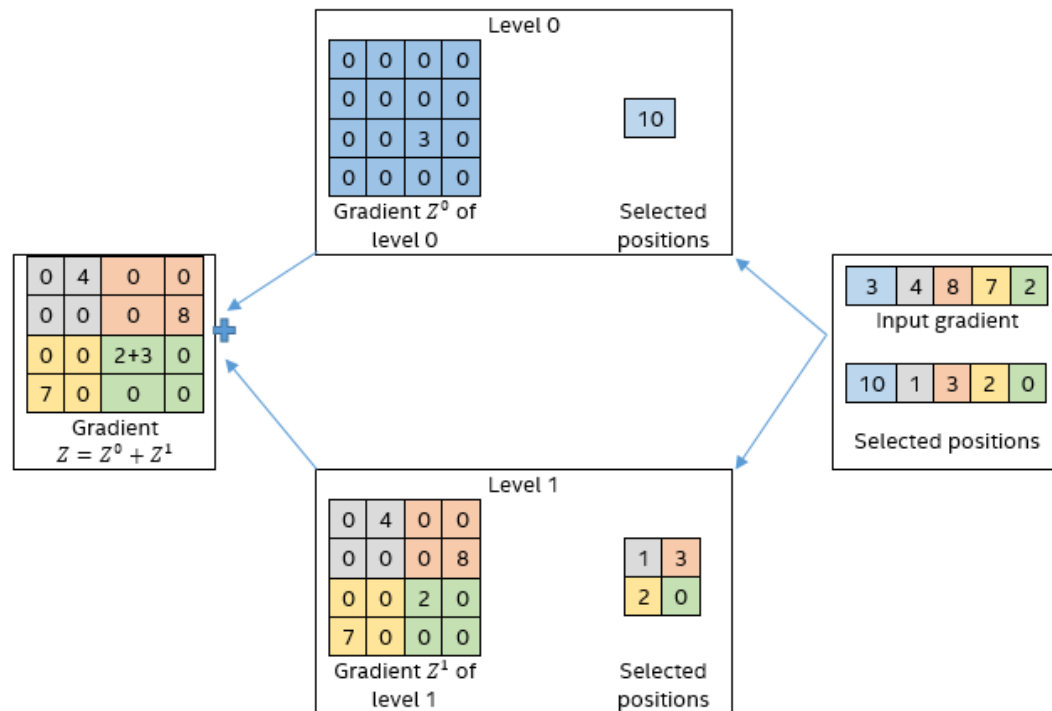
Let $G \in \mathbb{R}^{n_1 \times n'}$ be the two-dimensional tensor with the input gradient and k_1 and k_2 be the dimensions along which pooling kernels are applied. The backward 2D spatial pyramid pooling layer computes gradients for every pooling level from the respective input gradient and accumulates the pooling gradients to get the output gradient

$Z = (z_{i_1 \dots i_p}) \in \mathbb{R}^{n_1 \times n_2 \times \dots \times n_p}$ of the layer:

$$z_{i_1 \dots i_p} = \sum_{l=0}^{L-1} z_{i_1 \dots i_p}^l,$$

where $z_{i_1 \dots i_p}^l, l \in \{0, \dots, L-1\}$, is the gradient of the l -th pooling level.

The following figure illustrates the behavior of the backward spatial pyramid maximum pooling layer with pyramid height $L = 2$:



Batch Processing

Layer Input

The backward two-dimensional spatial pyramid pooling layer accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input	
inputGradient	Pointer to tensor G of size $n_1 \times n'$ that stores the input gradient computed on the preceding layer. This input can be an object of any class derived from <code>Tensor</code> .	
inputFromForward	Collection of data needed for the backward two-dimensional spatial pyramid pooling layer.	
Algorithm	Element ID	Element
any	auxInputDimensions	<code>NumericTable</code> of size $1 \times p$ that stores the sizes of the dimensions of input data tensor X : n_1, n_2, \dots, n_p .
spatial pyramid max pooling, spatial pyramid stochastic pooling	auxSelectedIndices	Tensor T of size $n_1 \times n'$ that stores indices of maximum/stochastic elements.

Layer Parameters

For common parameters of neural network layers, see [Common Parameters](#).

In addition to the common parameters, the backward two-dimensional spatial pyramid pooling layer has the following parameters:

Algorithm	Parameter	Default Value	Description
any	<i>algorithmFPType</i>	float	The floating-point type that the algorithm uses for intermediate computations. Can be float or double.
any	<i>method</i>	defaultDense	Performance-oriented computation method, the only method supported by the layer.
any	<i>indices</i>	Indices (p-2, p-1)	Indices k_1, k_2 of the dimensions along which spatial pyramid pooling is performed.
spatial pyramid stochastic pooling	<i>pyramidHeight</i>	Not applicable	Value L of the pyramid height.
spatial pyramid stochastic pooling	<i>seed</i>	777	Seed for multinomial random number generator.

Layer Output

The backward two-dimensional spatial pyramid pooling layer calculates the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
gradient	Pointer to tensor Z of size $n_1 \times n_2 \times n_3 \times n_4$ that stores the result of the backward two-dimensional spatial pyramid pooling layer. This input can be an object of any class derived from <code>Tensor</code> .

Examples

Refer to the following examples in your Intel DAAL directory:

C++:

- `./examples/cpp/source/neural_networks/spat_ave_pool2d_layer_dense_batch.cpp`
- `./examples/cpp/source/neural_networks/spat_max_pool2d_layer_dense_batch.cpp`
- `./examples/cpp/source/neural_networks/spat_stoch_pool2d_layer_dense_batch.cpp`

Java*:

- `./examples/java/source/com/intel/daal/examples/neural_networks/SpatAvePool2DLayerDenseBatch.java`
- `./examples/java/source/com/intel/daal/examples/neural_networks/SpatMaxPool2DLayerDenseBatch.java`
- `./examples/java/source/com/intel/daal/examples/neural_networks/SpatStochPool2DLayerDenseBatch.java`

Python*:

- `./examples/python/source/neural_networks/spat_ave_pool2d_layer_dense_batch.py`
- `./examples/python/source/neural_networks/spat_max_pool2d_layer_dense_batch.py`
- `./examples/python/source/neural_networks/spat_stoch_pool2d_layer_dense_batch.py`

2D Convolution Forward Layer

The forward two-dimensional (2D) convolution layer computes the tensor Y of values by applying a set of $nKernels$ 2D kernels K of size $m_3 \times m_4$ to the input tensor X . The library supports four-dimensional input tensors $X \in \mathbb{R}^{n_1 \times n_2 \times n_3 \times n_4}$. Therefore, the following formula applies:

$$y_{rij} = \sum_{c=0}^{n_2-1} \sum_{u=0}^{m_3-1} \sum_{v=0}^{m_4-1} k_{rcuv} \cdot x_{c(i+u)(j+v)} + b_r,$$

where $i+u < n_3$, $j+v < n_4$, and r is the kernel index.

Problem Statement

Without loss of generality, let's assume that convolution kernels are applied to the last two dimensions.

Given:

- Four-dimensional tensor $X \in \mathbb{R}^{n_1 \times n_2 \times n_3 \times n_4}$ with input data
- Four-dimensional tensor $K \in \mathbb{R}^{nKernels \times m_2 \times m_3 \times m_4}$ with kernel parameters/weights of kernels (convolutions)
- One-dimensional tensor $B \in \mathbb{R}^{nKernels}$ with the bias of each kernel.

For the above tensors:

- $m_2 = \frac{n_2}{nGroups}$, $m_i \leq n_i + 2 \cdot p_i$, $i \in \{3, 4\}$, and p_i is the respective padding.
- $nGroups$ is defined as follows: let's assume that n_2 is the group dimension. The input tensor is split along this dimension into $nGroups$ groups, the tensors of values and weights are split into $nGroups$ groups along the $nKernels$ dimension. $nKernels$ and n_2 must be multiples of $nGroups$. Each group of values is computed using the respective group in tensors of input data, weights, and biases.

The problem is to compute the four-dimensional tensor of values $Y \in \mathbb{R}^{n_1 \times nKernels \times l_3 \times l_4}$ such that:

$$y_{trab}^q = \sum_{c=0}^{m_2-1} \sum_{u=0}^{m_3-1} \sum_{v=0}^{m_4-1} k_{rcuv}^q x_{tc(a \cdot s_3 - p_3 + u)(b \cdot s_4 - p_4 + v)}^q + b_r^q,$$

$$y_{trab}^q = y_{t(q \cdot n_k + r)ab},$$

$$k_{rcuv}^q = k_{(q \cdot n_k + r)cuv},$$

$$x_{tcij}^q = x_{t(q \cdot m_2 + c)ij}$$

$$b_r^q = b_{q \cdot n_k + r},$$

where:

- s_3 and s_4 are strides

- $n_k = \frac{nKernels}{nGroups}$
- $t \in \{0, \dots, n_1 - 1\}$
- $r \in \{0, \dots, n_k - 1\}$
- $q \in \{0, \dots, nGroups - 1\}$
- $i \in \{0, \dots, n_3 - 1\}$
- $j \in \{0, \dots, n_4 - 1\}$
- $a \in \{0, \dots, l_3 - 1\}$
- $b \in \{0, \dots, l_4 - 1\}$
- $u \in \{0, \dots, m_3 - 1\}$
- $v \in \{0, \dots, m_4 - 1\}$
- $l_i = \left\lfloor \frac{n_i + 2p_i - m_i + s_i - 1}{s_i} \right\rfloor + 1, \quad i \in \{3, 4\}$

Batch Processing

Layer Input

The forward two-dimensional convolution layer accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
data	Pointer to tensor X of size $n_1 \times n_2 \times n_3 \times n_4$ that stores the input data for the forward two-dimensional convolution layer. This input can be an object of any class derived from <code>Tensor</code> .
weights	Pointer to tensor K of size $nKernels \times m_2 \times m_3 \times m_4$ that stores a set of kernel weights. This input can be an object of any class derived from <code>Tensor</code> .
biases	Pointer to tensor B of size $nKernels$ that stores a set of biases. This input can be an object of any class derived from <code>Tensor</code> . If you pass a null pointer, the library does not apply a bias-related transformation.

Layer Parameters

For common parameters of neural network layers, see [Common Parameters](#).

In addition to the common parameters, the forward two-dimensional convolution layer has the following parameters:

Parameter	Default Value	Description
<code>algorithmFPTType</code>	<code>float</code>	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<code>method</code>	<code>defaultDense</code>	Performance-oriented computation method, the only method supported by the layer.
<code>kernelSizes</code>	<code>KernelSizes(2, 2)</code>	Data structure representing the sizes $m_i, i \in \{3, 4\}$, of the two-dimensional kernel subtensor.

Parameter	Default Value	Description
<i>indices</i>	<code>Indices(2, 3)</code>	Data structure representing the dimensions for applying convolution kernels.
<i>strides</i>	<code>Strides(2, 2)</code>	Data structure representing the intervals $s_i, i \in \{3, 4\}$, on which the kernel should be applied to the input.
<i>padding</i> s	<code>Paddings(0, 0)</code>	Data structure representing the number of data elements $p_i, i \in \{3, 4\}$, to implicitly add to each side of the two-dimensional subtensor along which forward two-dimensional convolution is performed.
<i>nKernels</i>	n/a	Number of kernels applied to the input layer data.
<i>groupDimension</i>	1	Dimension for which grouping is applied.
<i>nGroups</i>	1	Number of groups into which the input data is split in dimension <code>groupDimension</code> .

Layer Output

The forward two-dimensional convolution layer calculates the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result						
<code>value</code>	Pointer to tensor Y of size $n_1 \times nKernels \times l_3 \times l_4$ that stores the result of the forward two-dimensional convolution layer. This result can be an object of any class derived from <code>Tensor</code> .						
<code>resultForBackward</code>	Collection of data received on the forward two-dimensional convolution layer. This collection can contain objects of any class derived from <code>Tensor</code> .						
	<table> <tr> <th>Element ID</th><th>Element</th></tr> <tr> <td><code>auxData</code></td><td>Pointer to tensor X of size $n_1 \times n_2 \times n_3 \times n_4$ that stores the input data for the forward two-dimensional convolution layer. This result can be an object of any class derived from <code>Tensor</code>.</td></tr> <tr> <td><code>auxWeights</code></td><td>Pointer to tensor K of size $nKernels \times m_2 \times m_3 \times m_4$ that stores a set of kernel weights. This result can be an object of any class derived from <code>Tensor</code>.</td></tr> </table>	Element ID	Element	<code>auxData</code>	Pointer to tensor X of size $n_1 \times n_2 \times n_3 \times n_4$ that stores the input data for the forward two-dimensional convolution layer. This result can be an object of any class derived from <code>Tensor</code> .	<code>auxWeights</code>	Pointer to tensor K of size $nKernels \times m_2 \times m_3 \times m_4$ that stores a set of kernel weights. This result can be an object of any class derived from <code>Tensor</code> .
Element ID	Element						
<code>auxData</code>	Pointer to tensor X of size $n_1 \times n_2 \times n_3 \times n_4$ that stores the input data for the forward two-dimensional convolution layer. This result can be an object of any class derived from <code>Tensor</code> .						
<code>auxWeights</code>	Pointer to tensor K of size $nKernels \times m_2 \times m_3 \times m_4$ that stores a set of kernel weights. This result can be an object of any class derived from <code>Tensor</code> .						

Examples

Refer to the following examples in your Intel DAAL directory:

C++: `./examples/cpp/source/neural_networks/convolution2d_layer_batch.cpp`

Java*: `./examples/java/source/com/intel/daal/examples/neural_networks/Convolution2DLayerBatch.java`

Python*: `./examples/python/source/neural_networks/convolution2d_layer_batch.py`

2D Convolution Backward Layer

The forward two-dimensional (2D) convolution layer applies a set of $nKernels$ 2D kernels K of size $m_3 \times m_4$ to the input tensor X . The library supports four-dimensional input tensors $X \in R^{n_1 \times n_2 \times n_3 \times n_4}$. Therefore, the following formula applies:

$$y_{.rij} = \sum_{c=0}^{n_2-1} \sum_{u=0}^{m_3-1} \sum_{v=0}^{m_4-1} k_{rcuv} x_{.c(i+u)(j+v)} + b_r,$$

where $i + u < n_3$, $j + v < n_4$, and r is the kernel index.

For more details, see [Forward 2D Convolution Layer](#).

The backward 2D convolution layer computes the derivatives of the objective function E :

$$\frac{\partial E}{\partial k_{rcuv}} = \frac{1}{n_1} \sum_{i=0}^{n_3-m_3-1} \sum_{j=0}^{n_4-m_4-1} g_{.rij} x_{.c(i+u)(j+v)},$$

$$\frac{\partial E}{\partial b_r} = \frac{1}{n_1} \sum_{i=0}^{n_3-1} \sum_{j=0}^{n_4-1} g_{.rij},$$

$$z_{.cij} = \frac{\partial E}{\partial x_{.cij}} = \sum_{n=0}^{nKernels-1} \sum_{u=0}^{m_3-1} \sum_{v=0}^{m_4-1} g_{.r(i-u)(j-v)} k_{rcuv}, \quad \text{where } g_{.rij} \equiv 0, \quad i < 0, \quad j < 0.$$

Problem Statement

Without loss of generality, let's assume that convolution kernels are applied to the last two dimensions.

Given:

- Four-dimensional tensor $G \in R^{n_1 \times nKernels \times l_3 \times l_4}$ with the gradient from the preceding layer
- Four-dimensional tensor $K \in R^{nKernels \times m_2 \times m_3 \times m_4}$ with kernel parameters/weights of kernels (convolutions)

For the above tensors:

- $m_2 = \frac{n_2}{nGroups}$, $m_i \leq n_i + 2 \cdot p_i$, $i \in \{3, 4\}$, and p_i is the respective padding.
- $nGroups$ is defined as follows: let's assume that n_2 is the group dimension in the input tensor for the forward 2D convolution layer. The output gradient tensor is split along this dimension into $nGroups$ groups, and the input gradient tensor and weights tensor are split into $nGroups$ groups along the $nKernels$ dimension. $nKernels$ and n_2 must be multiples of $nGroups$.

The problem is to compute:

- Four-dimensional tensor $Z \in R^{n_1 \times n_2 \times n_3 \times n_4}$ such that:

$$z_{tcij}^q = \sum_{r=0}^{n_k-1} \sum_{a \in U_3(i)} \sum_{b \in U_4(j)} g_{trab}^q k_{rc(i-a \cdot s_3+p_3)(j-b \cdot s_4+p_4)}^q,$$

$$z_{tcij}^q = z_{t(q \cdot m_2 + c)ij}$$

$$g_{trab}^q = g_{t(q \cdot n_k + r)ab}$$

$$k_{rcuv}^q = k_{(q \cdot n_k + r)cuv}$$

- Values:

$$\frac{\partial E}{\partial k_{rcuv}^q} = \frac{1}{n_1} \sum_{t=0}^{n_1-1} \sum_{a=0}^{l_3-1} \sum_{b=0}^{l_4-1} g_{trab}^q x_{tc}^q (a \cdot s_3 - p_3 + u)(b \cdot s_4 - p_4 + v),$$

$$\frac{\partial E}{\partial b_r^q} = \frac{1}{n_1} \sum_{t=0}^{n_1-1} \sum_{a=0}^{l_3-1} \sum_{b=0}^{l_4-1} g_{trab}^q,$$

$$x_{tcij}^q = x_{t(q \cdot m_2 + c)ij}$$

$$b_r^q = b_{q \cdot n_k + r}$$

In the above formulas:

- s_3 and s_4 are strides

-

$$n_k = \frac{nKernels}{nGroups}$$

- $t \in \{0, \dots, n_1 - 1\}$

- $r \in \{0, \dots, n_k - 1\}$

- $q \in \{0, \dots, nGroups - 1\}$

- $i \in \{0, \dots, n_3 - 1\}$

- $j \in \{0, \dots, n_4 - 1\}$

- $a \in \{0, \dots, l_3 - 1\}$

- $b \in \{0, \dots, l_4 - 1\}$

-

$$U_i(a) = \left\{ j: \left\lfloor \frac{a - m_i + p_i}{s_i} \right\rfloor + 1 \leq j \leq \left\lfloor \frac{a + p_i}{s_i} \right\rfloor \right\}, i \in \{3, 4\}$$

- $u \in \{0, \dots, m_3 - 1\}$

- $v \in \{0, \dots, m_4 - 1\}$

-

$$l_i = \left\lfloor \frac{n_i + 2p_i - m_i + s_i - 1}{s_i} \right\rfloor + 1, \quad i \in \{3, 4\}$$

Batch Processing

Layer Input

The backward two-dimensional convolution layer accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
<code>inputGradient</code>	Pointer to tensor G of size $n_1 \times nKernels \times l_3 \times l_4$ that stores the input gradient computed on the preceding layer. This input can be an object of any class derived from <code>Tensor</code> .
<code>inputFromForward</code>	Collection of input data needed for the backward two-dimensional convolution layer. This collection can contain objects of any class derived from <code>Tensor</code> .
Element ID	Element
<code>auxData</code>	Pointer to tensor X of size $n_1 \times n_2 \times n_3 \times n_4$ that stores the input data for the forward two-dimensional convolution layer. This input can be an object of any class derived from <code>Tensor</code> .
<code>auxWeights</code>	Pointer to tensor K of size $nKernels \times m_2 \times m_3 \times m_4$ that stores a set of kernel weights. This input can be an object of any class derived from <code>Tensor</code> .

Layer Parameters

For common parameters of neural network layers, see [Common Parameters](#).

In addition to the common parameters, the backward two-dimensional convolution layer has the following parameters:

Parameter	Default Value	Description
<code>algorithmFPType</code>	<code>float</code>	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<code>method</code>	<code>defaultDense</code>	Performance-oriented computation method, the only method supported by the layer.
<code>kernelSizes</code>	<code>KernelSizes(2, 2)</code>	Data structure representing the sizes $m_i, i \in \{3, 4\}$, of the two-dimensional kernel subtensor.
<code>indices</code>	<code>Indices(2, 3)</code>	Data structure representing the dimensions for applying convolution kernels.
<code>strides</code>	<code>Strides(2, 2)</code>	Data structure representing the intervals $s_i, i \in \{3, 4\}$, on which the kernel should be applied to the input.
<code>padding</code>	<code>Paddings(0, 0)</code>	Data structure representing the number of data elements $p_i, i \in \{3, 4\}$, to implicitly add to each side of the two-dimensional subtensor along which forward two-dimensional convolution is performed.
<code>nKernels</code>	n/a	Number of kernels applied to the input layer data.
<code>groupDimension</code>	1	Dimension for which grouping is applied.

Parameter	Default Value	Description
<i>nGroups</i>	1	Number of groups into which the input data is split in dimension <code>groupDimension</code> .
<i>propagateGradient</i>	false	Flag that specifies whether the backward layer propagates the gradient.

Layer Output

The backward two-dimensional convolution layer calculates the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
<code>gradient</code>	Pointer to tensor Z of size $n_1 \times n_2 \times n_3 \times n_4$ that stores the result of the backward two-dimensional convolution layer. This result can be an object of any class derived from <code>Tensor</code> .
<code>weightDerivatives</code>	Pointer to the tensor of size $nKernels \times m_2 \times m_3 \times m_4$ that stores result $\partial E / \partial k_{rcuv}$ of the backward two-dimensional convolution layer, where $r = \{0, \dots, nKernels - 1\}$, $c = \{0, \dots, m_2 - 1\}$, $u = \{0, \dots, m_3 - 1\}$, $v = \{0, \dots, m_4 - 1\}$. This result can be an object of any class derived from <code>Tensor</code> .
<code>biasDerivatives</code>	Pointer to the tensor of size $nKernels$ that stores result $\partial E / \partial b_r$ of the backward two-dimensional convolution layer, where $r = \{0, \dots, nKernels - 1\}$. This result can be an object of any class derived from <code>Tensor</code> .

Examples

Refer to the following examples in your Intel DAAL directory:

C++: `./examples/cpp/source/neural_networks/convolution2d_layer_batch.cpp`

Java*: `./examples/java/source/com/intel/daal/examples/neural_networks/Convolution2DLayerBatch.java`

Python*: `./examples/python/source/neural_networks/convolution2d_layer_batch.py`

2D Transposed Convolution Forward Layer

The forward two-dimensional (2D) transposed convolution layer computes the tensor Y by applying a set of $nKernels$ 2D kernels K of size $m_3 \times m_4$ to the input tensor X .

Problem Statement

The problem is to compute the four-dimensional tensor of values $Y \in \mathbb{R}^{n_1 \times nKernels \times n_3 \times n_4}$ such that:

$$y_{tcij}^q = \sum_{r=0}^{n_k-1} \sum_{a \in U_3(i)} \sum_{b \in U_4(j)} x_{trab}^q k_{rc(i-a \cdot s_3 + p_3)(j-b \cdot s_4 + p_4)}^q + b_c^q,$$

where $b_c^q = b_{q \cdot n_c + c}$.

For the notations in this formula, refer to [2D Convolution Backward Layer](#).

The computation flow in the forward 2D transposed convolution layer is identical to the computation of the gradient in the 2D convolution backward layer, except the following notation changes:

2D Convolution Backward Layer	2D Transposed Convolution Forward Layer
Input gradient tensor G	Input tensor X
Gradient tensor Z	Result tensor Y
$nKernels$	l_2
n_2	$nKernels$

Batch Processing

Layer Input

The forward two-dimensional transposed convolution layer accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
<code>data</code>	Pointer to tensor X of size $n_1 \times l_2 \times l_3 \times l_4$ that stores the input data of the forward two-dimensional transposed convolution layer. This input can be an object of any class derived from <code>Tensor</code> .
<code>weights</code>	Pointer to the tensor K of size $l_2 \times nKernels/nGroups \times m_3 \times m_4$ that stores a set of kernel weights. This input can be an object of any class derived from <code>Tensor</code> .
<code>biases</code>	Pointer to the tensor B of size $nKernels$ that stores a set of biases. This input can be an object of any class derived from <code>Tensor</code> . If you pass a null pointer, the library does not apply a bias-related transformation.

Layer Parameters

For common parameters of neural network layers, see [Common Parameters](#).

In addition to the common parameters, the forward two-dimensional transposed convolution layer has the following parameters:

Parameter	Default Value	Description
<code>algorithmFPType</code>	<code>float</code>	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<code>method</code>	<code>defaultDense</code>	Performance-oriented computation method, the only method supported by the layer.
<code>kernelSizes</code>	<code>KernelSizes(2, 2)</code>	Data structure representing the sizes $m_i, i \in \{3, 4\}$, of the two-dimensional kernel subtensor.
<code>indices</code>	<code>Indices(2,3)</code>	Data structure representing the dimensions for applying transposed convolution kernels.
<code>strides</code>	<code>Strides(2, 2)</code>	Data structure representing the intervals $s_i, i \in \{3, 4\}$, on which the kernel should be applied to the input.

Parameter	Default Value	Description
<i>padding</i> s	<code>padding</code> s(0, 0)	Data structure representing the number of data elements p_i , $i \in \{3, 4\}$, to implicitly add to each side of the two-dimensional subtensor along which forward two-dimensional transposed convolution is performed.
<i>value</i> Size	<code>Value</code> Size(0, 0)	Data structure representing the dimension sizes n_i , $i \in \{3, 4\}$, of the value tensor Y . If this value contains (0, 0), the size of value dimensions are computed automatically.
<i>n</i> Kernels	n/a	Number of kernels applied to the layer input data.
<i>group</i> Dimension	1	Dimension to which grouping is applied.
<i>n</i> Groups	1	Number of groups into which the input data is split in dimension <i>group</i> Dimension.

Layer Output

The forward two-dimensional transposed convolution layer calculates the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result						
value	Pointer to tensor Y of size $n_1 \times nKernels \times n_3 \times n_4$ that stores the result of the forward two-dimensional transposed convolution layer. This result can be an object of any class derived from <code>Tensor</code> .						
resutForBackward	Collection of data obtained on the forward two-dimensional transposed convolution layer. This collection can contain objects of any class derived from <code>Tensor</code> .						
	<table> <tr> <th>Element ID</th><th>Element</th></tr> <tr> <td>auxData</td><td>Pointer to tensor X of size $n_1 \times l_2 \times l_3 \times l_4$ that stores the input data for the forward two-dimensional transposed convolution layer. This result can be an object of any class derived from <code>Tensor</code>.</td></tr> <tr> <td>auxWeights</td><td>Pointer to tensor K of size $l_2 \times nKernels/nGroups \times m_3 \times m_4$ that stores a set of kernel weights. This result can be an object of any class derived from <code>Tensor</code>.</td></tr> </table>	Element ID	Element	auxData	Pointer to tensor X of size $n_1 \times l_2 \times l_3 \times l_4$ that stores the input data for the forward two-dimensional transposed convolution layer. This result can be an object of any class derived from <code>Tensor</code> .	auxWeights	Pointer to tensor K of size $l_2 \times nKernels/nGroups \times m_3 \times m_4$ that stores a set of kernel weights. This result can be an object of any class derived from <code>Tensor</code> .
Element ID	Element						
auxData	Pointer to tensor X of size $n_1 \times l_2 \times l_3 \times l_4$ that stores the input data for the forward two-dimensional transposed convolution layer. This result can be an object of any class derived from <code>Tensor</code> .						
auxWeights	Pointer to tensor K of size $l_2 \times nKernels/nGroups \times m_3 \times m_4$ that stores a set of kernel weights. This result can be an object of any class derived from <code>Tensor</code> .						

2D Transposed Convolution Backward Layer

The forward two-dimensional (2D) transposed convolution layer computes the tensor Y by applying a set of *nKernels* 2D kernels K of size $m_3 \times m_4$ to the input tensor X . For more details, refer to [2D Transposed Convolution Forward Layer](#).

The backward 2D transposed convolution layer computes the derivatives of the objective function E .

Problem Statement

The problem is to compute:

- The four-dimensional tensor of values $Z \in \mathbb{R}^{n_1 \times l_2 \times l_3 \times l_4}$ such that:

$$z_{trab}^q = \sum_{c=0}^{m_2-1} \sum_{u=0}^{m_3-1} \sum_{v=0}^{m_4-1} k_{rcuv}^q g_{tc}^q(a \cdot s_3 - p_3 + u)(b \cdot s_4 - p_4 + v)$$

- Values:

$$\frac{\partial E}{\partial k_{rcuv}^q} = \frac{1}{n_1} \sum_{t=0}^{n_1-1} \sum_{(i,a) \in V_3(u)} \sum_{(j,b) \in V_4(v)} x_{trab}^q g_{tcij}^q$$

$$\frac{\partial E}{\partial b_c^q} = \frac{1}{n_1} \sum_{t=0}^{n_1-1} \sum_{i=0}^{l_3-1} \sum_{j=0}^{l_4-1} g_{tcij}^q$$

where:

- $V_j(u) = \{(i, a) \in \{0, \dots, n_j - 1\} \times \{0, \dots, l_j - 1\} : a \cdot s_j - p_j + u = i\}, \quad j \in \{3, 4\}$
- $b_c^q = b_{q \cdot n_c + c}$

For the notations in this formula, refer to [2D Convolution Forward Layer](#).

The computation flow in the backward 2D transposed convolution layer is identical to the computation of the gradient in the 2D convolution forward layer, except the following notation changes:

2D Convolution Forward Layer	2D Transposed Convolution Backward Layer
Input tensor X	Input gradient tensor G
Values tensor Y	Gradient tensor Z
$nKernels$	l_2
n_2	$nKernels$

Batch Processing

Layer Input

The backward two-dimensional transposed convolution layer accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
inputGradient	Pointer to tensor G of size $n_1 \times nKernels \times n_3 \times n_4$ that stores the input gradient computed on the preceding layer. This input can be an object of any class derived from <code>Tensor</code> .

Input ID	Input	
inputFromForward	Collection of input data for the backward two-dimensional transposed convolution layer. This collection can contain objects of any class derived from <code>Tensor</code> .	
	Element ID	Element
	auxData	Pointer to tensor X of size $n_1 \times l_2 \times l_3 \times l_4$ that stores the input data for the forward two-dimensional transposed convolution layer. This input can be an object of any class derived from <code>Tensor</code> .
	auxWeights	Pointer to the tensor K of size $l_2 \times nKernels/nGroups \times m_3 \times m_4$ that stores a set of kernel weights. This input can be an object of any class derived from <code>Tensor</code> .

Layer Parameters

For common parameters of neural network layers, see [Common Parameters](#).

In addition to the common parameters, the backward two-dimensional transposed convolution layer has the following parameters:

Parameter	Default Value	Description
<i>algorithmFPType</i>	float	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<i>method</i>	defaultDense	Performance-oriented computation method, the only method supported by the layer.
<i>kernelSizes</i>	KernelSizes(2, 2)	Data structure representing the sizes $m_i, i \in \{3, 4\}$, of the two-dimensional kernel subtensor.
<i>indices</i>	Indices(2,3)	Data structure representing the dimensions for applying transposed convolution kernels.
<i>strides</i>	Strides(2, 2)	Data structure representing the intervals $s_i, i \in \{3, 4\}$, on which the kernel should be applied to the input.
<i>padding</i> s	Paddings(0, 0)	Data structure representing the number of data elements $p_i, i \in \{3, 4\}$, to implicitly add to each side of the two-dimensional subtensor along which backward two-dimensional transposed convolution is performed.
<i>nKernels</i>	n/a	Number of kernels applied to the layer input data.
<i>groupDimension</i>	1	Dimension to which grouping is applied.

Parameter	Default Value	Description
<i>nGroups</i>	1	Number of groups into which the input data is split in dimension <i>groupDimension</i> .
<i>propagateGradient</i>	false	Flag that specifies whether the forward layer propagates the gradient.

Layer Output

The backward two-dimensional transposed convolution layer calculates the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
gradient	Pointer to tensor Z of size $n_1 \times l_2 \times l_3 \times l_4$ that stores the computed gradient. This result can be an object of any class derived from <code>Tensor</code> .
weightDerivatives	Pointer to the tensor of size $l_2 \times nKernels/nGroups \times m_3 \times m_4$ that stores the derivatives $\partial E / \partial k_{rcuv}$, where $r = \{0, \dots, l_2\}$, $c = \{0, \dots, nKernels/nGroups\}$, $u = \{0, \dots, m_3 - 1\}$, and $v = \{0, \dots, m_4 - 1\}$. This result can be an object of any class derived from <code>Tensor</code> .
biasDerivatives	Pointer to tensor of size $nKernels$ that stores the derivatives $\partial E / \partial b_r$, where $r = \{0, \dots, nKernels - 1\}$. This result can be an object of any class derived from <code>Tensor</code> .

2D Locally-Connected Forward Layer

The forward two-dimensional (2D) locally-connected layer computes the value tensor Y by applying a set of $nKernels$ 2D kernels K of size $m_1 \times m_2$ to the input argument x . The library supports four-dimensional input tensors $X \in \mathbb{R}^{n_1 \times n_2 \times n_3 \times n_4}$. Therefore, the following formula applies:

$$y_{,rij} = \sum_{a=0}^{m_1-1} \sum_{b=0}^{m_2-1} k_{rij,ab} \cdot x_{,i+a,j+b} + b_{rij},$$

where $i + a < n_1$, $j + b < n_2$, and r is the kernel index.

A set of kernels is specific to the selected dimensions of the input argument x .

See [\[GregorLecun2010\]](#) for additional details of the two-dimensional locally-connected layer.

Problem Statement

Without loss of generality let's assume that convolution kernels are applied to the last two dimensions.

Given:

- Four-dimensional tensor $X \in \mathbb{R}^{n_1 \times n_2 \times n_3 \times n_4}$ with input data
- Six-dimensional tensor $K \in \mathbb{R}^{nKernels \times l_3 \times l_4 \times m_2 \times m_3 \times m_4}$ with kernel parameters/weights
- Three-dimensional tensor $B \in \mathbb{R}^{nKernels \times l_3 \times l_4}$ with the bias of each kernel.

For the above tensors:

- $m_2 = \frac{n_2}{nGroups}$, $m_i \leq n_i + 2 \cdot p_i$, $i \in \{3, 4\}$, and p_i is the respective padding.

- $nGroups$ is defined as follows: let's assume that n_2 is the group dimension. The input tensor is split along this dimension into $nGroups$ groups, the tensors of values and weights are split into $nGroups$ groups along the $nKernels$ dimension. $nKernels$ and n_2 must be multiples of $nGroups$. Each group of values is computed using the respective group in tensors of input data, weights, and biases.

The problem is to compute the four-dimensional tensor of values $Y \in \mathbb{R}^{n_1 \times nKernels \times l_3 \times l_4}$ such that:

$$y_{trij}^q = \sum_{c=0}^{n_c-1} \sum_{a=0}^{m_3-1} \sum_{b=0}^{m_4-1} k_{rijcab}^q x_{tc(i \cdot s_3 - p_3 + a)(j \cdot s_4 - p_4 + b)}^q + b_{rij}^q,$$

$$y_{trij}^q = y_{t(q \cdot n_k + r)ij},$$

$$k_{rijcab}^q = k_{(q \cdot n_k + r)ijcab},$$

$$x_{tcij}^q = x_{t(q \cdot n_c + c)ij},$$

$$b_r^q = b_{q \cdot n_k + r},$$

where:

- s_3 and s_4 are strides
-

$$n_c = \frac{n_2}{nGroups}, \quad n_k = \frac{nKernels}{nGroups}$$

- $t \in \{0, \dots, n_1 - 1\}$
- $r \in \{0, \dots, n_k - 1\}$
- $q \in \{0, \dots, nGroups - 1\}$
- $i \in \{0, \dots, l_3 - 1\}$
- $j \in \{0, \dots, l_4 - 1\}$
- $l_i = \frac{n_i + 2p_i - m_i}{s_i} + 1, \quad i \in \{3, 4\}$

Batch Processing

Layer Input

The forward two-dimensional locally-connected layer accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
data	Pointer to tensor X of size $n_1 \times n_2 \times n_3 \times n_4$ that stores the input data for the forward two-dimensional locally-connected layer. This input can be an object of any class derived from <code>Tensor</code> .

Input ID	Input
<code>weights</code>	Pointer to tensor K of size $nKernels \times l_3 \times l_4 \times m_2 \times m_3 \times m_4$ that stores a set of kernel weights. This input can be an object of any class derived from <code>Tensor</code> .
<code>biases</code>	Pointer to tensor B of size $nKernels \times l_3 \times l_4$ that stores a set of biases. This input can be an object of any class derived from <code>Tensor</code> . If you pass a null pointer, the library does not apply a bias-related transformation.

Layer Parameters

For common parameters of neural network layers, see [Common Parameters](#).

In addition to the common parameters, the forward two-dimensional locally-connected layer has the following parameters:

Parameter	Default Value	Description
<code>algorithmFPType</code>	<code>float</code>	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<code>method</code>	<code>defaultDense</code>	Performance-oriented computation method, the only method supported by the layer.
<code>kernelSizes</code>	<code>KernelSizes(2, 2)</code>	Data structure representing the sizes $m_i, i \in \{3, 4\}$ of the two-dimensional kernel subtensor.
<code>indices</code>	<code>Indices(2, 3)</code>	Data structure representing the dimensions for applying kernels of the forward locally-connected layer.
<code>strides</code>	<code>Strides(2, 2)</code>	Data structure representing the intervals $s_i, i \in \{3, 4\}$ on which the kernel should be applied to the input.
<code>padding</code>	<code>Paddings(0, 0)</code>	Data structure representing the number of data elements $p_i, i \in \{3, 4\}$ to implicitly add to each side of the two-dimensional subtensor to which the kernel are applied. Only symmetric padding is currently supported.
<code>nKernels</code>	<code>n/a</code>	Number of kernels applied to the input layer data.
<code>groupDimension</code>	<code>1</code>	Dimension n_2 for which grouping is applied. Only <code>groupDimension = 1</code> is currently supported.
<code>nGroups</code>	<code>1</code>	Number of groups into which the input data is split in dimension <code>groupDimension</code> .

Layer Output

The forward two-dimensional locally-connected layer calculates the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
value	Pointer to tensor Y of size $n_1 \times nKernels \times l_3 \times l_4$ that stores the result of the forward two-dimensional locally-connected layer. This input can be an object of any class derived from <code>Tensor</code> .
layerData	Collection of data received on the forward two-dimensional locally-connected layer. This collection can contain objects of any class derived from <code>Tensor</code> .
Element ID	Element
auxData	Pointer to tensor X of size $n_1 \times n_2 \times n_3 \times n_4$ that stores the input data for the forward two-dimensional locally-connected layer. This input can be an object of any class derived from <code>Tensor</code> .
auxWeights	Pointer to tensor K of size $nKernels \times l_3 \times l_4 \times m_2 \times m_3 \times m_4$ that stores a set of kernel weights. This input can be an object of any class derived from <code>Tensor</code> .

Examples

Refer to the following examples in your Intel DAAL directory:

C++: `./examples/cpp/source/neural_networks/locallycon2d_layer_dense_batch.cpp`

Java*: `./examples/java/source/com/intel/daal/examples/neural_networks/Locallycon2DLayerDenseBatch.java`

Python*: `./examples/python/source/neural_networks/locallycon2d_layer_dense_batch.py`

2D Locally-connected Backward Layer

The forward two-dimensional (2D) locally-connected layer computes the value tensor Y by applying a set of $nKernels$ 2D kernels K of size $m_1 \times m_2$ to the input argument x . The library supports four-dimensional input tensors $X \in R^{n_1 \times n_2 \times n_3 \times n_4}$. Therefore, the following formula applies:

$$y_{rij} = \sum_{a=0}^{m_1-1} \sum_{b=0}^{m_2-1} k_{rij \cdot ab} x_{i+a, j+b} + b_{rij},$$

where $i + a < n_1$, $j + b < n_2$, and r is the kernel index.

A set of kernels is specific to the selected dimensions of the input argument x .

For more details, see [Forward 2D Locally-connected Layer](#).

The backward 2D locally-connected layer computes the derivatives of the objective function E with respect to the input argument, weights, and biases.

Problem Statement

Without loss of generality, let's assume that convolution kernels are applied to the last two dimensions.

Given:

- Four-dimensional tensor $G \in \mathbb{R}^{n_1 \times nKernels \times l_3 \times l_4}$ with the gradient from the preceding layer
- Four-dimensional tensor $X \in \mathbb{R}^{n_1 \times n_2 \times n_3 \times n_4}$ with input data of the forward layer
- Six-dimensional tensor $K \in \mathbb{R}^{nKernels \times l_3 \times l_4 \times m_2 \times m_3 \times m_4}$ with kernel parameters/weights
- Three-dimensional tensor $B \in \mathbb{R}^{nKernels \times l_3 \times l_4}$ with the bias of each kernel.

For the above tensors:

- $m_2 = \frac{n_2}{nGroups}$, $m_i \leq n_i + 2 \cdot p_i$, $i \in \{3, 4\}$, and p_i is the respective padding.
- $nGroups$ is defined as follows: let's assume that n_2 is the group dimension in the input tensor for the forward 2D locally-connected layer. The output gradient tensor is split along this dimension into $nGroups$ groups, and the input gradient tensor and weights tensor are split into $nGroups$ groups along the $nKernels$ dimension. $nKernels$ and n_2 must be multiples of $nGroups$.

The problem is to compute:

- Four-dimensional tensor $Z \in \mathbb{R}^{n_1 \times n_2 \times n_3 \times n_4}$ such that:

$$z_{tcij}^q = \sum_{r=0}^{n_k-1} \sum_{a \in U_3(i)} \sum_{b \in U_4(j)} g_{trab}^q k_{rabc}^q (i - a \cdot s_3 + p_3)(j - b \cdot s_4 + p_4),$$

$$z_{tcij}^q = z_{t(q \cdot n_c + c)ij},$$

$$g_{trij}^q = g_{t(q \cdot n_k + r)ij},$$

$$k_{rijcab}^q = k_{(q \cdot n_k + r)ijcab}$$

- Values:

$$\frac{\partial E}{\partial k_{rijcuv}^q} = \frac{1}{n_1} \sum_{t=0}^{n_1-1} g_{trij}^q x_{tc(i \cdot s_3 - p_3 + u)(j \cdot s_4 - p_4 + v)}^q,$$

$$\frac{\partial E}{\partial b_{rij}} = \frac{1}{n_1} \sum_{t=0}^{n_1-1} g_{trij}$$

In the above formulas:

- s_3 and s_4 are strides

•

$$n_c = \frac{n_2}{nGroups}, \quad n_k = \frac{nKernels}{nGroups}$$

- $x_{tcij}^q = x_{t(q \cdot n_c + c)ij}$

•

$$U_i(a) = \left\{ 0 \leq j < l_i : \left\lfloor \frac{a - m_i + p_i}{s_i} \right\rfloor + 1 \leq j \leq \left\lfloor \frac{a + p_i}{s_i} \right\rfloor \right\}$$

- $t \in \{0, \dots, n_1 - 1\}$
- $c \in \{0, \dots, n_2 - 1\}$
- $i \in \{0, \dots, n_3 - 1\}$
- $j \in \{0, \dots, n_4 - 1\}$
- $r \in \{0, \dots, n_k - 1\}$
- $q \in \{0, \dots, nGroups - 1\}$
- $u \in \{0, \dots, m_3 - 1\}$
- $v \in \{0, \dots, m_4 - 1\}$
- $l_i = \frac{n_i + 2p_i - m_i}{s_i} + 1, \quad i \in \{3, 4\}$

Batch Processing

Layer Input

The backward two-dimensional locally-connected layer accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
inputGradient	Pointer to tensor G of size $n_1 \times nKernels \times l_3 \times l_4$ that stores the input gradient computed on the preceding layer. This input can be an object of any class derived from <code>Tensor</code> .
inputFromForward	Collection of input data needed for the backward two-dimensional locally-connected layer. This collection can contain objects of any class derived from <code>Tensor</code> .
Element ID	Element
auxData	Pointer to tensor X of size $n_1 \times n_2 \times n_3 \times n_4$ that stores the input data for the forward two-dimensional locally-connected layer. This input can be an object of any class derived from <code>Tensor</code> .
auxWeights	Pointer to tensor K of size $nKernels \times l_3 \times l_4 \times m_2 \times m_3 \times m_4$ that stores a set of kernel weights. This input can be an object of any class derived from <code>Tensor</code> .

Layer Parameters

For common parameters of neural network layers, see [Common Parameters](#).

In addition to the common parameters, the backward two-dimensional locally-connected layer has the following parameters:

Parameter	Default Value	Description
<i>algorithmFPTYPE</i>	float	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<i>method</i>	defaultDense	Performance-oriented computation method, the only method supported by the layer.
<i>kernelSizes</i>	KernelSizes(2, 2)	Data structure representing the sizes m_i , $i \in \{3, 4\}$ of the two-dimensional kernel subtensor.
<i>indices</i>	Indices(2, 3)	Data structure representing the dimensions for applying kernels of the backward locally-connected layer.
<i>strides</i>	Strides(2, 2)	Data structure representing the intervals s_i , $i \in \{3, 4\}$ on which the kernel should be applied to the input.
<i>padding</i> s	Paddings(0, 0)	Data structure representing the number of data elements p_i , $i \in \{3, 4\}$ to implicitly add to each side of the two-dimensional subtensor to which the kernel are applied. Only symmetric padding is currently supported.
<i>nKernels</i>	n/a	Number of kernels applied to the input layer data.
<i>groupDimension</i>	1	Dimension n_2 for which grouping is applied. Only <i>groupDimension</i> = 1 is currently supported.
<i>nGroups</i>	1	Number of groups into which the input data is split in dimension <i>groupDimension</i> .
<i>propagateGradient</i>	false	Flag that specifies whether the backward layer propagates the gradient.

Layer Output

The backward two-dimensional locally-connected layer calculates the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
gradient	Pointer to tensor Z of size $n_1 \times n_2 \times n_3 \times n_4$ that stores the result of the backward two-dimensional locally-connected layer. This input can be an object of any class derived from <code>Tensor</code> .
weightDerivatives	Pointer to the tensor of size $nKernels \times l_3 \times l_4 \times m_2 \times m_3 \times m_4$ that stores result $\partial E / \partial k_{rijcuv}$ of the backward two-dimensional locally-connected layer. This input can be an object of any class derived from <code>Tensor</code> .
biasDerivatives	Pointer to the tensor of size $nKernels \times l_3 \times l_4$ that stores result $\partial E / \partial b_{rij}$ of the backward two-dimensional locally-connected layer. This input can be an object of any class derived from <code>Tensor</code> .

Examples

Refer to the following examples in your Intel DAAL directory:

C++: `./examples/cpp/source/neural_networks/locallycon2d_layer_dense_batch.cpp`

Java*: `./examples/java/source/com/intel/daal/examples/neural_networks/`
`Locallycon2DLayerDenseBatch.java`

Python*: `./examples/python/source/neural_networks/locallycon2d_layer_dense_batch.py`

Reshape Forward Layer

The forward reshape layer generates a tensor from input argument X of size $n_1 \times n_2 \times \dots \times n_p$ with modified size of dimensions $m_1 \times m_2 \times \dots \times m_q$ without modifying data and its order.

Problem Statement

Given the dimension sizes of the output tensor $m_1 \times m_2 \times \dots \times m_q$, such that

$$\prod_{i=1}^p n_i = \prod_{i=1}^q m_i.$$

NOTE

There are two reserved values of m_i :

- 0 means $m_i = n_i$, valid only when $i \leq p$
- -1 means m_i is calculated to make the product of all m_i equal to the product of all n_i , valid only if used once. Corresponding m_i calculated as follows:

$$m_i = \frac{\prod_{j=1}^p n_j}{\prod_{j=1 \dots i-1 \ i+1 \dots q} m_j}.$$

The q -dimensional result tensor Y of size $m_1 \times \dots \times m_q$ contains the same data from X in unmodified order.

Batch Processing

Layer Input

The forward reshape layer accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
data	Pointer to tensor of size $n_1 \times n_2 \times \dots \times n_p$ that stores the input data for the forward reshape layer. This input can be an object of any class derived from <code>Tensor</code> .

Layer Parameters

For common parameters of neural network layers, see [Common Parameters](#).

In addition to the common parameters, the forward reshape layer has the following parameters:

Parameter	Default Value	Description
<i>reshapeDimensions</i>	Not applicable	Collection of dimension sizes: m_1, m_2, \dots, m_q for the output Tensor.

Layer Output

The forward reshape layer calculates the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result	
value	Pointer to tensor of size $m_1 \times m_2 \times \dots \times m_q$ that stores the result of the forward reshape layer. This input can be an object of any class derived from <code>Tensor</code> .	
layerData	Collection of data needed for the backward reshape layer.	
	Element ID	Element
	auxInputDimensions	Collection of integers that stores the dimension sizes of the input tensor in the forward computation step: n_1, n_2, \dots, n_p .

Examples

Refer to the following examples in your Intel DAAL directory:

C++: `./examples/cpp/source/neural_networks/reshape_layer_batch.cpp`

Java*: `./examples/java/source/com/intel/daal/examples/neural_networks/ReshapeLayerBatch.java`

Python*: `./examples/python/source/neural_networks/reshape_layer_batch.py`

Reshape Backward Layer

The forward reshape layer generates a tensor from input argument X of size $n_1 \times n_2 \times \dots \times n_p$ with modified size of dimensions $m_1 \times m_2 \times \dots \times m_q$ without modifying data and its order. For more details, see [Forward Reshape Layer](#). The backward reshape layer generates output tensor Z of size $n_1 \times n_2 \times \dots \times n_p$ from input tensor G of dimensions $m_1 \times m_2 \times \dots \times m_q$.

Problem Statement

Given the dimension sizes of the output tensor $n_1 \times n_2 \times \dots \times n_p$, such that

$$\prod_{i=1}^p n_i = \prod_{i=1}^q m_i.$$

The p -dimensional result tensor Z of size $n_1 \times n_2 \times \dots \times n_p$ contains the same data from G in unmodified order.

Batch Processing

Layer Input

The backward reshape layer accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
inputGradient	Pointer to tensor of size $m_1 \times \dots \times m_q$ that stores the input gradient G computed on the preceding layer. This input can be an object of any class derived from <code>Tensor</code> .
inputFromForward	Collection of input data needed for the backward reshape layer.
Element ID	Element
auxInputDimensions	Collection of integers that stores the dimension sizes of the input tensors in the forward computation step: n_1, \dots, n_p .

Layer Output

The backward reshape layer calculates the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
gradient	Pointer to tensor of size $n_1 \times n_2 \times \dots \times n_p$ that stores the result of the backward reshape layer. This input can be an object of any class derived from <code>Tensor</code> .

Examples

Refer to the following examples in your Intel DAAL directory:

C++: `./examples/cpp/source/neural_networks/reshape_layer_batch.cpp`

Java*: `./examples/java/source/com/intel/daal/examples/neural_networks/ReshapeLayerBatch.java`

Python*: `./examples/python/source/neural_networks/reshape_layer_batch.py`

Concat Forward Layer

The forward concat layer generates a tensor from input arguments X .

Problem Statement

Given the dimension k along which the concatenation to be implemented and tp -dimensional tensors $X^{(i)}$, $i=1, \dots, t$, of size $n_1 \times n_2 \times \dots \times n_{k_i} \times \dots \times n_p$, where $n_1, \dots, n_{k-1}, n_{k+1}, \dots, n_p$ are the same for all tensors, the problem is to generate

- The p -dimensional tensor $Y = (y_{i_1 \dots i_p})$ of size $n_1 \times \dots \times n_k \times \dots \times n_p$, such that:

$$y_{i_1 \dots i_p} = (x_{i_1 \dots i_{k_1} \dots i_p}^{(1)}, x_{i_1 \dots i_{k_2} \dots i_p}^{(2)}, \dots, x_{i_1 \dots i_{k_t} \dots i_p}^{(t)})$$

$$n_k = \sum_{i=1}^t n_{k_i}$$

- A collection of sizes n_{k_i} along the k -th dimension.

Batch Processing

Layer Input

The forward concat layer accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
<code>inputLayerData</code>	Collection of tensors of size $n_1 \times \dots \times n_{k_i} \times \dots \times n_p$ that stores the input data for the forward concat layer. This collection can contain objects of any class derived from <code>Tensor</code> .

Layer Parameters

For common parameters of neural network layers, see [Common Parameters](#).

In addition to the common parameters, the forward concat layer has the following parameters:

Parameter	Default Value	Description
<code>concatDimension</code>	0	Index of the dimension along which concatenation should be implemented.

Layer Output

The forward concat layer calculates the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result				
<code>value</code>	Pointer to the tensor of size $n_1 \times \dots \times n_k \times \dots \times n_p$ that stores the result of the forward concat layer. This input can be an object of any class derived from <code>Tensor</code> .				
<code>resultForBackward</code>	Collection of data needed for the backward concat layer.				
	<table> <tr> <th>Element ID</th><th>Element</th></tr> <tr> <td><code>auxInputDimensions</code></td><td>Collection of integers that stores the sizes of the input tensors along <code>concatDimension</code>: n_{k_1}, \dots, n_{k_t}.</td></tr> </table>	Element ID	Element	<code>auxInputDimensions</code>	Collection of integers that stores the sizes of the input tensors along <code>concatDimension</code> : n_{k_1}, \dots, n_{k_t} .
Element ID	Element				
<code>auxInputDimensions</code>	Collection of integers that stores the sizes of the input tensors along <code>concatDimension</code> : n_{k_1}, \dots, n_{k_t} .				

Examples

Refer to the following examples in your Intel DAAL directory:

C++: `./examples/cpp/source/neural_networks/concat_layer_batch.cpp`

Java*: `./examples/java/source/com/intel/daal/examples/neural_networks/ConcatLayerBatch.java`

Python*: `./examples/python/source/neural_networks/concat_layer_batch.py`

Concat Backward Layer

The forward concat layer generates a tensor from input arguments X . For more details, see [Forward Concat Layer](#). The backward concat layer deconcatenates a tensor from the input argument G into multiple output tensors $Z^{(j)}$.

Problem Statement

Given the dimension k along which the concatenation to be implemented, a collection $N = (n_{k_1}, \dots, n_{k_t})$ of t integer elements, and a p -dimensional tensor $G = (g_{i_1 \dots i_p})$ of size $n_1 \times n_2 \times \dots \times n_p$, the problem is to generate t p -dimensional tensors $Z^{(j)}$, $j=1, \dots, t$, of size $n_1 \times \dots \times n_{k_j} \times \dots \times n_p$ such that:

$$z_{i_1 \dots i_{k_j} \dots i_p}^{(j)} = g_{i_1 \dots i_k \dots i_p},$$

$$i_{k_j} = s_j + 1, \dots, s_{j+1},$$

$$s_j = \sum_{i=1}^{j-1} n_{k_i}.$$

Batch Processing

Layer Input

The backward concat layer accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input				
inputGradient	Pointer to the tensor of size $n_1 \times \dots \times n_p$ that stores input gradient G computed on the preceding layer. This input can be an object of any class derived from <code>Tensor</code> .				
inputFromForward	Collection of data needed for the backward concat layer.				
	<table> <tr> <th>Element ID</th><th>Element</th></tr> <tr> <td>auxInputDimensions</td><td>Collection of integers that stores the sizes along the k-th dimension of the input tensors in the forward computation step: n_{k_1}, \dots, n_{k_t}.</td></tr> </table>	Element ID	Element	auxInputDimensions	Collection of integers that stores the sizes along the k -th dimension of the input tensors in the forward computation step: n_{k_1}, \dots, n_{k_t} .
Element ID	Element				
auxInputDimensions	Collection of integers that stores the sizes along the k -th dimension of the input tensors in the forward computation step: n_{k_1}, \dots, n_{k_t} .				

Layer Parameters

For common parameters of neural network layers, see [Common Parameters](#).

In addition to the common parameters, the backward concat layer has the following parameters:

Parameter	Default Value	Description
<code>concatDimension</code>	0	Index of the dimension along which deconcatenation should be implemented.

Layer Output

The backward concat layer calculates the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
resultLayerData	Collection of tensors of size $n_1 \times \dots \times n_{k_j} \times \dots \times n_p$ that stores the result of the backward concat layer. This collection can contain objects of any class derived from <code>Tensor</code> .

NOTE

The `gradient` field stores a null pointer. All the computation results are stored in `resultLayerData`.

Examples

Refer to the following examples in your Intel DAAL directory:

C++: `./examples/cpp/source/neural_networks/concat_layer_batch.cpp`

Java*: `./examples/java/source/com/intel/daal/examples/neural_networks/ConcatLayerBatch.java`

Python*: `./examples/python/source/neural_networks/concat_layer_batch.py`

Split Forward Layer

The forward split layer copies a tensor from the input argument $X \in \mathbb{R}^{n_1 \times n_2 \times \dots \times n_p}$ to multiple output tensors.

Problem Statement

Given the number of output tensors k and a p -dimensional tensor $X = (x_{i_1 \dots i_p})$ of size $n_1 \times n_2 \times \dots \times n_p$, the problem is to generate $k p$ -dimensional tensors $Y^{(j)}$ ($j=1, \dots, k$) of size $n_1 \times n_2 \times \dots \times n_p$, such that:

$$y_{i_1 \dots i_p}^{(1)} = y_{i_1 \dots i_p}^{(2)} = \dots = y_{i_1 \dots i_p}^{(k)} = x_{i_1 \dots i_p},$$

$$i_1 = 1, \dots, n_1, \quad i_2 = 1, \dots, n_2, \quad \dots, \quad i_p = 1, \dots, n_p.$$

Batch Processing**Layer Input**

The forward split layer accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
data	Tensor of size $n_1 \times \dots \times n_p$ that stores the input data for the forward split layer. This input can be an object of any class derived from <code>Tensor</code> .

Layer Parameters

For common parameters of neural network layers, see [Common Parameters](#).

In addition to the common parameters, the forward split layer has the following parameters:

Parameter	Default Value	Description
<code>nOutputs</code>	1	Number of output tensors.

Layer Output

The forward split layer calculates the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
valueCollection	Collection of k tensors of size $n_1 \times \dots \times n_p$ that stores the result of the forward split layer. This collection can contain objects of any class derived from <code>Tensor</code> .

NOTE

The `value` result stores a null pointer while `valueCollection` stores actual computation results.

Examples

Refer to the following examples in your Intel DAAL directory:

C++: `./examples/cpp/source/neural_networks/split_layer_batch.cpp`

Java*: `./examples/java/source/com/intel/daal/examples/neural_networks/SplitLayerBatch.java`

Python*: `./examples/python/source/neural_networks/split_layer_batch.py`

Split Backward Layer

The forward split layer copies a tensor from the input argument $X \in R^{n_1 \times n_2 \times \dots \times n_p}$ to multiple output tensors. For more details, see [Forward Split Layer](#).

Problem Statement

Given kp -dimensional input gradient tensors $G^{(j)}$ of size $n_1 \times n_2 \times \dots \times n_p$, where $j=1, \dots, k$, the backward split layer computes the value:

$$z_{i_1 \dots i_p} = \sum_{j=1}^k g_{i_1 \dots i_p}^{(j)}, \quad i_1 = 1, \dots, n_1, \quad i_2 = 1, \dots, n_2, \quad \dots, \quad i_p = 1, \dots, n_p.$$

Batch Processing**Layer Input**

The backward split layer accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
inputGradientCollection	Collection of k tensors of size $n_1 \times \dots \times n_p$ that stores the result of the forward split layer. This collection can contain objects of any class derived from <code>Tensor</code> .

NOTE

`inputGradient` stores a null pointer while `inputGradientCollection` stores all input tensors.

Layer Parameters

For common parameters of neural network layers, see [Common Parameters](#).

In addition to the common parameters, the backward split layer has the following parameters:

Parameter	Default Value	Description
<i>nInputs</i>	1	Number of input tensors.

Layer Output

The backward split layer calculates the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
gradient	Tensor of size $n_1 \times \dots \times n_p$ that stores result z of the backward split layer. This result can be an object of any class derived from <code>Tensor</code> .

Examples

Refer to the following examples in your Intel DAAL directory:

C++: `./examples/cpp/source/neural_networks/split_layer_batch.cpp`

Java*: `./examples/java/source/com/intel/daal/examples/neural_networks/`
`SplitLayerBatch.java`

Python*: `./examples/python/source/neural_networks/split_layer_batch.py`

Softmax Forward Layer

For any $x_{i_1 \dots i_p}$ from $X \in R^{n_1 \times \dots \times n_p}$ and dimension k of size n_k , the forward softmax layer for computes the function defined as

$$y_{i_1 \dots i_p} = \frac{e^{x_{i_1 \dots i_p}}}{\sum_{i_k=1}^{n_k} e^{x_{i_1 \dots i_k-1 i_k i_k+1 \dots i_p}}}, \quad k \in \{1, \dots, p\}.$$

The softmax function is known as the normalized exponential (see [Bishop2006] for exact definitions of softmax).

Problem Statement

Given a p -dimensional tensor X of size $n_1 \times n_2 \times \dots \times n_p$, the problem is to compute the p -dimensional tensor $Y = (y_{i_1 \dots i_p})$ of size $n_1 \times n_2 \times \dots \times n_p$ such that:

$$y_{i_1 \dots i_p} = \frac{e^{x_{i_1 \dots i_p}}}{\sum_{i_k=1}^{n_k} e^{x_{i_1 \dots i_k-1 i_k i_k+1 \dots i_p}}}.$$

The library supports the numerically stable version of the softmax function:

$$y_{i_1 \dots i_p} = \frac{e^{(x_{i_1 \dots i_p} - \max_{i_k \in \{1 \dots n_k\}} x_{i_1 \dots i_k-1 i_k i_k+1 \dots i_p})}}{\sum_{i_k=1}^{n_k} e^{(x_{i_1 \dots i_k-1 i_k i_k+1 \dots i_p} - \max_{i_k \in \{1 \dots n_k\}} x_{i_1 \dots i_k-1 i_k i_k+1 \dots i_p})}},$$

where

$$\max_{i_k \in \{1 \dots n_k\}} x_{i_1 \dots i_k-1 i_k i_k+1 \dots i_p} = \max_{i_k \in \{1 \dots n_k\}} x_{i_1 \dots i_k-1 i_k i_k+1 \dots i_p}.$$

Batch Processing

Layer Input

The forward softmax layer accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
data	Pointer to the tensor of size $n_1 \times n_2 \times \dots \times n_p$ that stores the input data for the forward softmax layer. This input can be an object of any class derived from <code>Tensor</code> .

Layer Parameters

For common parameters of neural network layers, see [Common Parameters](#).

In addition to the common parameters, the forward softmax layer has the following parameters:

Parameter	Default Value	Description
<code>algorithmFPType</code>	float	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<code>method</code>	defaultDense	Performance-oriented computation method, the only method supported by the layer.
<code>dimension</code>	1	Index of the dimension of type <code>size_t</code> to calculate softmax.

Layer Output

The forward softmax layer calculates the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
value	Pointer to the tensor of size $n_1 \times n_2 \times \dots \times n_p$ that stores the result of the forward softmax layer. This input can be an object of any class derived from <code>Tensor</code> .

Examples

Refer to the following examples in your Intel DAAL directory:

C++: `./examples/cpp/source/neural_networks/softmax_layer_batch.cpp`

Java*: `./examples/java/source/com/intel/daal/examples/neural_networks/SoftmaxLayerBatch.java`

Python*: `./examples/python/source/neural_networks/softmax_layer_batch.py`

Softmax Backward Layer

For any $x_{i_1 \dots i_p}$ from $X \in R^{n_1 \times \dots \times n_p}$ and for dimension k of size n_k , the softmax activation layer applies the transform defined as

$$y\left(x_{i_1 \dots i_p}\right) = y_{i_1 \dots i_p} = \frac{e^{x_{i_1 \dots i_p}}}{\sum_{i_k=1}^{n_k} e^{x_{i_1 \dots i_{k-1} i_k i_{k+1} \dots i_p}}}, \quad k \in \{1, \dots, p\}.$$

The softmax function is known as the normalized exponential (see [Bishop2006] for exact definitions of softmax).

The backward softmax layer for dimension k of size n_k computes the value:

$$z_{i_1 \dots i_p} = y_{i_1 \dots i_p} \left(g_{i_1 \dots i_p} - \sum_{i_k=1}^{n_k} g_{i_1 \dots i_{k-1} i_{k+1} \dots i_p} y_{i_1 \dots i_{k-1} i_{k+1} \dots i_p} \right),$$

where $g_{i_1 \dots i_p}$ is the input gradient computed on the preceding layer.

Problem Statement

Given p -dimensional tensors of size $n_1 \times n_2 \times \dots \times n_p$:

- $G = (g_{i_1 \dots i_p})$ with the gradient computed on the preceding layer
- $Y = (y_{i_1 \dots i_p})$ with the output of the forward softmax layer

The problem is to compute the p -dimensional tensor $Z = (z_{i_1 \dots i_p})$ of size $n_1 \times n_2 \times \dots \times n_p$ such that:

$$z_{i_1 \dots i_p} = y_{i_1 \dots i_p} \left(g_{i_1 \dots i_p} - \sum_{i_k=1}^{n_k} g_{i_1 \dots i_{k-1} i_{k+1} \dots i_p} y_{i_1 \dots i_{k-1} i_{k+1} \dots i_p} \right).$$

Batch Processing

Layer Input

The backward softmax layer accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
inputGradient	Pointer to tensor G of size $n_1 \times \dots \times n_p$ that stores the input gradient computed on the preceding layer. This input can be an object of any class derived from <code>Tensor</code> .
inputFromForward	Collection of input data needed for the backward softmax layer. The collection may contain objects of any class derived from <code>Tensor</code> .
Element ID	Element
auxValue	Pointer to tensor Y of size $n_1 \times n_2 \times \dots \times n_p$ that stores the result of the forward softmax layer. This input can be an object of any class derived from <code>Tensor</code> .

Layer Parameters

For common parameters of neural network layers, see [Common Parameters](#).

In addition to the common parameters, the backward softmax layer has the following parameters:

Parameter	Default Value	Description
<code>algorithmFPType</code>	<code>float</code>	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .

Parameter	Default Value	Description
<i>method</i>	defaultDense	Performance-oriented computation method, the only method supported by the layer.
<i>dimension</i>	1	Index of the dimension of type <i>size_t</i> to calculate softmax backpropagation.

Layer Output

The backward softmax layer calculates the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
gradient	Pointer to tensor Z of size $n_1 \times n_2 \times \dots \times n_p$ that stores the result of the backward softmax layer. This input can be an object of any class derived from <code>Tensor</code> .

Examples

Refer to the following examples in your Intel DAAL directory:

C++: `./examples/cpp/source/neural_networks/softmax_layer_batch.cpp`

Java*: `./examples/java/source/com/intel/daal/examples/neural_networks/SoftmaxLayerBatch.java`

Python*: `./examples/python/source/neural_networks/softmax_layer_batch.py`

Loss Forward Layer

The loss function of a neural network is the sum of loss functions for each data sample. The loss function measures and penalizes the difference between the output of the neural network and ground truth. Because the loss layer evaluates the quality of the model being trained, it must be the last layer in the neural network topology.

The loss layer takes the ground truth T and output values X of the preceding layer as input and computes the value y of the cost function E :

$$y = E(X, T) = \frac{1}{n} \sum_i^n \text{loss}(x_i, t_i).$$

Batch Processing

Layer Input

The forward loss layer accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
data	Pointer to the tensor that stores the input data for the forward loss layer. This input can be an object of any class derived from <code>Tensor</code> .

Input ID	Input
groundTruth	Pointer to the tensor that stores the ground truth data for the forward loss layer. This input can be an object of any class derived from <code>Tensor</code> .

Layer Output

The forward loss layer calculates the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
value	Pointer to the one-dimensional tensor of size 1 that stores the result of the forward loss layer. This input can be an object of any class derived from <code>Tensor</code> .
resultForBackward	Collection of data needed for the backward loss layer.

Loss Backward Layer

Taking the ground truth T and output values X of the preceding layer as input, the forward loss layer computes the value y of the cost function E . For more details, see [Forward Loss Layer](#). The backward loss layer computes the derivative of the cost function E with respect to the input X :

$$z_i = \frac{\partial E}{\partial x_i} = \frac{1}{n} \frac{\partial \text{loss}(x_i, t_i)}{\partial x_i}, \quad i = 1, \dots, n.$$

Batch Processing

Layer Input

The backward loss layer accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
inputGradient	Pointer to the tensor that stores the input gradient computed on the preceding layer. This input can be an object of any class derived from <code>Tensor</code> .
inputFromForward	Collection of data needed for the backward loss layer.

NOTE

The `inputGradient` field stores a null pointer because there is no input gradient on the last layer.

Layer Output

The backward loss layer calculates the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
gradient	Pointer to the tensor that stores the result of the backward loss layer. This input can be an object of any class derived from <code>Tensor</code> .

Loss Softmax Cross-entropy Forward Layer

The loss softmax cross-entropy layer implements an interface of the loss layer.

For an input tensor $X \in \mathbb{R}^{n_1 \times n_2 \times \dots \times n_k \times \dots \times n_p}$, selected dimension k of size n_k , and ground truth tensor $T \in \mathbb{R}^{n_1 \times n_2 \times \dots \times 1 \times \dots \times n_p}$, the layer computes a one-dimensional tensor with the cross-entropy value:

$$y = - \frac{1}{n_1 * \dots * n_{k-1} * n_{k+1} * \dots * n_p} \sum_m \log s_{m t_m},$$

where $s_{m t_m}$ defined below is the probability that the sample m corresponds to the ground truth t_m .

Problem Statement

Given:

- The p -dimensional tensor $X = (x_{j_1 \dots j_k \dots j_p}) \in \mathbb{R}^{n_1 \times n_2 \times \dots \times n_k \times \dots \times n_p}$ with input data
- The p -dimensional tensor $T = (t_{j_1 \dots j_k \dots j_p}) \in \mathbb{R}^{n_1 \times n_2 \times \dots \times 1 \times \dots \times n_p}$ that contains the values of ground truth,
where $t_{j_1 \dots j_p} \equiv t_{j_1 \dots j_{k-1} 1 j_{k+1} \dots j_p} \in \{0, \dots, n_k - 1\}$.

The problem is to compute a one-dimensional tensor $Y \in \mathbb{R}^1$ such that:

$$y = - \frac{1}{n_1 * \dots * n_{k-1} * n_{k+1} * \dots * n_p} \sum_{j_1=1}^{n_1} \dots \sum_{j_{k-1}=1}^{n_{k-1}} \sum_{j_{k+1}=1}^{n_{k+1}} \dots \sum_{j_p=1}^{n_p} \log s_{j_1 \dots j_{k-1} 1 j_{k+1} \dots j_p}.$$

The library uses the numerically stable formula for computing the probability value $s_{j_1 \dots j_{k-1} 1 j_{k+1} \dots j_p}$:

$$s_{j_1 \dots j_{k-1} 1 j_{k+1} \dots j_p} = \frac{e^{x_{j_1 \dots j_{k-1} 1 j_{k+1} \dots j_p} - \max_{j_1 \dots j_{k-1} 1 j_{k+1} \dots j_p} x_{j_1 \dots j_{k-1} 1 j_{k+1} \dots j_p}}}{\sum_{i_k=1}^{n_k} e^{x_{j_1 \dots j_{k-1} i_k j_{k+1} \dots j_p} - \max_{j_1 \dots j_{k-1} 1 j_{k+1} \dots j_p} x_{j_1 \dots j_{k-1} 1 j_{k+1} \dots j_p}}},$$

where

$$\max_{j_1 \dots j_{k-1} 1 j_{k+1} \dots j_p} x_{j_1 \dots j_{k-1} 1 j_{k+1} \dots j_p} = \max_{i_k \in \{1 \dots n_k\}} x_{j_1 \dots j_{k-1} i_k j_{k+1} \dots j_p}.$$

See Also

[Loss Forward Layer](#)

[Batch Processing](#)

Layer Input

The forward loss softmax cross-entropy layer accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
<code>data</code>	Pointer to the tensor X of size $n_1 \times n_2 \times \dots \times n_k \times \dots \times n_p$ that stores the input data for the forward loss softmax cross-entropy layer. This input can be an object of any class derived from <code>Tensor</code> .
<code>groundTruth</code>	Pointer to the tensor of size $n_1 \times n_2 \times \dots \times 1 \times \dots \times n_p$ that stores the ground truth data for the forward loss softmax cross-entropy layer. This input can be an object of any class derived from <code>Tensor</code> .

Layer Parameters

For common parameters of neural network layers, see [Common Parameters](#).

In addition to the common parameters, the forward loss softmax cross-entropy layer has the following parameters:

Parameter	Default Value	Description
<code>algorithmFPType</code>	<code>float</code>	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<code>method</code>	<code>defaultDense</code>	Performance-oriented computation method, the only method supported by the layer.
<code>accuracyThresold</code>	<code>0.0001</code>	The value needed to avoid degenerate cases in computing the logarithm.
<code>dimension</code>	<code>1</code>	The dimension index to calculate softmax cross-entropy.

Layer Output

The forward loss softmax cross-entropy layer calculates the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result						
<code>value</code>	Pointer to tensor Y of size 1 that stores the result of the forward loss softmax cross-entropy layer. This input can be an object of any class derived from <code>Tensor</code> .						
<code>resultForBackward</code>	Collection of data needed for the backward loss softmax cross-entropy layer. The collection may contain objects of any class derived from <code>Tensor</code> . <table> <tr> <th>Element ID</th><th>Element</th></tr> <tr> <td><code>auxProbabilities</code></td><td>Pointer to tensor S of size $n_1 \times n_2 \times \dots \times n_k \times \dots \times n_p$ that stores probabilities for the forward loss softmax cross-entropy layer. This input can be an object of any class derived from <code>Tensor</code>.</td></tr> <tr> <td><code>auxGroundTruth</code></td><td>Pointer to tensor T of size $n_1 \times n_2 \times \dots \times 1 \times \dots \times n_p$ that stores the ground truth data for the forward</td></tr> </table>	Element ID	Element	<code>auxProbabilities</code>	Pointer to tensor S of size $n_1 \times n_2 \times \dots \times n_k \times \dots \times n_p$ that stores probabilities for the forward loss softmax cross-entropy layer. This input can be an object of any class derived from <code>Tensor</code> .	<code>auxGroundTruth</code>	Pointer to tensor T of size $n_1 \times n_2 \times \dots \times 1 \times \dots \times n_p$ that stores the ground truth data for the forward
Element ID	Element						
<code>auxProbabilities</code>	Pointer to tensor S of size $n_1 \times n_2 \times \dots \times n_k \times \dots \times n_p$ that stores probabilities for the forward loss softmax cross-entropy layer. This input can be an object of any class derived from <code>Tensor</code> .						
<code>auxGroundTruth</code>	Pointer to tensor T of size $n_1 \times n_2 \times \dots \times 1 \times \dots \times n_p$ that stores the ground truth data for the forward						

Result ID	Result
	loss softmax cross-entropy layer. This input can be an object of any class derived from <code>Tensor</code> .

Examples

Refer to the following examples in your Intel DAAL directory:

C++: `./examples/cpp/source/neural_networks/loss_softmax_entr_layer_dense_batch.cpp`

Java*: `./examples/java/source/com/intel/daal/examples/neural_networks/LossSoftmaxCrossEntropyLayerBatch.java`

Python*: `./examples/python/source/neural_networks/loss_softmax_entr_layer_dense_batch.py`

Loss Softmax Cross-entropy Backward Layer

For an input tensor $X \in R^{n_1 \times n_2 \times \dots \times n_k \times \dots \times n_p}$, selected dimension k of size n_k , and ground truth tensor $T \in R^{n_1 \times n_2 \times \dots \times 1 \times \dots \times n_p}$, the forward loss softmax cross-entropy layer computes a one-dimensional tensor with the cross-entropy value. For more details, see [Forward Loss Softmax Cross-entropy Layer](#).

The backward loss softmax cross-entropy layer computes gradient values $z_m = s_m - \delta_m$, where s_m are probabilities computed on the forward layer and δ_m are indicator functions computed using t_m , the ground truth values computed on the preceding layer.

Problem Statement

Given:

- The p -dimensional tensor $T = (t_{j_1 \dots j_k \dots j_p}) \in R^{n_1 \times n_2 \times \dots \times 1 \times \dots \times n_p}$ that contains ground truths, where

$$t_{j_1 \dots j_p} \equiv t_{j_1 \dots j_{k-1} 1 j_{k+1} \dots j_p} \in \{0, \dots, n_k - 1\}.$$
- The p -dimensional tensor $S = (s_{j_1 \dots j_k \dots j_p}) \in R^{n_1 \times n_2 \times \dots \times n_k \times \dots \times n_p}$ with the probabilities that the sample $j_1 \dots j_p$ corresponds to the ground truth $t_{j_1 \dots j_p}$.

The problem is to compute a one-dimensional tensor $Z \in R^{n_1 \times n_2 \times \dots \times n_k \times \dots \times n_p}$ such that:

$$z_{j_1 \dots j_{k-1} i j_{k+1} \dots j_p} = s_{j_1 \dots j_{k-1} i j_{k+1} \dots j_p} - \delta_{j_1 \dots j_{k-1} i j_{k+1} \dots j_p},$$

where

$$\delta_{j_1 \dots j_{k-1} i j_{k+1} \dots j_p} = \begin{cases} 1, & i = t_{j_1 j_2 \dots j_p} \\ 0, & \text{otherwise} \end{cases}$$

Batch Processing

Layer Input

The backward loss softmax cross-entropy layer accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
<code>inputGradient</code>	Pointer to the tensor that stores the input gradient computed on the preceding layer. This input can be an object of any class derived from <code>Tensor</code> .
<code>inputFromForward</code>	Collection of data needed for the backward loss softmax cross-entropy layer. The collection may contain objects of any class derived from <code>Tensor</code> .
Element ID	Element
<code>auxProbabilities</code>	Pointer to the tensor S of size $n_1 \times n_2 \times \dots \times n_k \times \dots \times n_p$ that stores probabilities for the forward loss softmax cross-entropy layer. This input can be an object of any class derived from <code>Tensor</code> .
<code>auxGroundTruth</code>	Pointer to the tensor T of size $n_1 \times n_2 \times \dots \times 1 \times \dots \times n_p$ that stores the ground truth data for the forward loss softmax cross-entropy layer. This input can be an object of any class derived from <code>Tensor</code> .

NOTE

The layer does not use `inputGradient` because there is no input gradient on the last layer.

Layer Parameters

For common parameters of neural network layers, see [Common Parameters](#).

In addition to the common parameters, the backward loss softmax cross-entropy layer has the following parameters:

Parameter	Default Value	Description
<code>algorithmFPType</code>	<code>float</code>	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<code>method</code>	<code>defaultDense</code>	Performance-oriented computation method, the only method supported by the layer.
<code>dimension</code>	<code>1</code>	The dimension index to calculate softmax cross-entropy.

Layer Output

The backward loss softmax cross-entropy layer calculates the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
gradient	Pointer to the tensor Z of size $n_1 \times n_2 \times \dots \times n_k \times \dots \times n_p$ that stores the result of the backward loss softmax cross-entropy layer. This input can be an object of any class derived from <code>Tensor</code> .

Examples

Refer to the following examples in your Intel DAAL directory:

C++: `./examples/cpp/source/neural_networks/loss_softmax_entr_layer_dense_batch.cpp`

Java*: `./examples/java/source/com/intel/daal/examples/neural_networks/LossSoftmaxCrossEntropyLayerBatch.java`

Python*: `./examples/python/source/neural_networks/loss_softmax_entr_layer_dense_batch.py`

Loss Logistic Cross-entropy Forward Layer

The loss logistic cross-entropy layer implements an interface of the loss layer.

For a p -dimensional input tensor $\in R^{n_1 \times 1 \times \dots \times 1}$ with batch dimension of size n_1 and p -dimensional $T \in R^{n_1 \times 1 \times \dots \times 1}$ (or one-dimensional tensor $T \in R^{n_1}$), the layer computes a one-dimensional tensor Y with the logistic cross-entropy value:

$$y = -\frac{1}{n_1} \sum_{i=1}^{n_1} (pr_i \cdot \log(\sigma(x_i)) + (1 - pr_i) \cdot \log(1 - \sigma(x_i))),$$

where \log is the natural logarithm, $\sigma(x)$ is the logistic function, and $pr_i \in [0,1]$ is the probability that a sample belongs to the first of two classes.

Problem Statement

Given:

- The p -dimensional input tensor $X \in R^{n_1 \times 1 \times \dots \times 1}$ with input data
- The p -dimensional $T \in R^{n_1 \times 1 \times \dots \times 1}$ or one-dimensional tensor $T \in R^{n_1}$ that contains the values of ground truth for each element of the batch

The problem is to compute a one-dimensional tensor $Y \in R^1$ such that:

$$y = -\frac{1}{n_1} \sum_{i=1}^{n_1} s_i,$$

$$s_i = pr_i \cdot \log(\sigma(x_i)) + (1 - pr_i) \cdot \log(1 - \sigma(x_i)),$$

$$\sigma(x_i) = \frac{1}{1 + e^{-x_i}}.$$

The library uses the numeric stable formula for computing the value of s_i :

$$s_i = -x_i \cdot (\delta_i - pr_i) - \log(1 + e^{-|x_i|}),$$

where

- $$\delta_i = \begin{cases} 1, & x_i > 0 \\ 0, & x_i \leq 0 \end{cases}$$
- $$i \in \{1, \dots, n_1\}$$

Batch Processing

Layer Input

The forward loss logistic cross-entropy layer accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
<code>data</code>	Pointer to tensor X of size $n_1 \times 1 \times \dots \times 1$ that stores the input data for the forward loss logistic cross-entropy layer. This input can be an object of any class derived from <code>Tensor</code> .
<code>groundTruth</code>	Pointer to the tensor T of size n_1 or $n_1 \times 1 \times \dots \times 1$ that stores the ground truth data for the forward loss logistic cross-entropy layer. This input can be an object of any class derived from <code>Tensor</code> .

Layer Parameters

For common parameters of neural network layers, see [Common Parameters](#).

In addition to the common parameters, the forward loss logistic cross-entropy layer has the following parameters:

Parameter	Default Value	Description
<code>algorithmFPType</code>	<code>float</code>	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<code>method</code>	<code>defaultDense</code>	Performance-oriented computation method, the only method supported by the layer.

Layer Output

The forward loss logistic cross-entropy layer calculates the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result				
<code>value</code>	Pointer to tensor Y of size 1 that stores the result of the forward loss logistic cross-entropy layer. This input can be an object of any class derived from <code>Tensor</code> .				
<code>resultForBackward</code>	Collection of data needed for the backward loss logistic cross-entropy layer. The collection may contain objects of any class derived from <code>Tensor</code> .				
	<table> <tr> <th>Element ID</th><th>Element</th></tr> <tr> <td><code>auxData</code></td><td>Pointer to tensor X of size $n_1 \times 1 \times \dots \times 1$ that stores probabilities for the forward loss logistic cross-</td></tr> </table>	Element ID	Element	<code>auxData</code>	Pointer to tensor X of size $n_1 \times 1 \times \dots \times 1$ that stores probabilities for the forward loss logistic cross-
Element ID	Element				
<code>auxData</code>	Pointer to tensor X of size $n_1 \times 1 \times \dots \times 1$ that stores probabilities for the forward loss logistic cross-				

Result ID	Result
	entropy layer. This input can be an object of any class derived from <code>Tensor</code> .
<code>auxGroundTruth</code>	Pointer to tensor T of size n_1 or $n_1 \times 1 \times \dots \times 1$ that stores the ground truth data for the forward loss logistic cross-entropy layer. This input can be an object of any class derived from <code>Tensor</code> .

Examples

Refer to the following examples in your Intel DAAL directory:

C++: `./examples/cpp/source/neural_networks/loss_logistic_entr_layer_dense_batch.cpp`

Java*: `./examples/java/source/com/intel/daal/examples/neural_networks/LossLogisticCrossEntropyLayerBatch.java`

Python*: `./examples/python/source/neural_networks/loss_logistic_entr_layer_dense_batch.py`

Loss Logistic Cross-entropy Backward Layer

For a p -dimensional input tensor $\in R^{n_1 \times 1 \times \dots \times 1}$ with batch dimension of size n_1 and p -dimensional $T \in R^{n_1 \times 1 \times \dots \times 1}$ (or one-dimensional tensor $T \in R^{n_1}$), the layer computes a one-dimensional tensor with the logistic cross-entropy value. For more details, see [Forward Loss Logistic Cross-entropy Layer](#).

The backward loss logistic cross-entropy layer for a given p -dimensional tensor $X \in R^{n_1 \times 1 \times \dots \times 1}$ and p -dimensional $T \in R^{n_1 \times 1 \times \dots \times 1}$ or one-dimensional tensor $T \in R^{n_1}$ computes the p -dimensional tensor $Z \in R^{n_1 \times 1 \times \dots \times 1}$ such that

$$z_i = \frac{1}{n_1} (\sigma(x_i) - pr_i),$$

where

- $pr_i \in [0, 1]$ is the probability that a sample belongs to the first of two classes
- $\sigma(x_i) = \frac{1}{1 + e^{-x_i}}$
- $i \in \{0, \dots, n_1 - 1\}$

Batch Processing

Layer Input

The backward loss logistic cross-entropy layer accepts the input described below. Pass the Input ID as a parameter to the methods that provide input for your algorithm. For more details, see [Algorithms](#).

Input ID	Input
<code>inputGradient</code>	Pointer to the tensor that stores the input gradient computed on the preceding layer. This input can be an object of any class derived from <code>Tensor</code> .

Input ID	Input						
inputFromForward	Collection of data needed for the backward loss logistic cross-entropy layer. The collection may contain objects of any class derived from <code>Tensor</code> .						
	<table> <tr> <th>Element ID</th><th>Element</th></tr> <tr> <td>auxData</td><td>Pointer to tensor X of size $n_1 \times 1 \times \dots \times 1$ that stores probabilities for the forward loss logistic cross-entropy layer. This input can be an object of any class derived from <code>Tensor</code>.</td></tr> <tr> <td>auxGroundTruth</td><td>Pointer to tensor T of size n_1 or $n_1 \times 1 \times \dots \times 1$ that stores the ground truth data for the forward loss logistic cross-entropy layer. This input can be an object of any class derived from <code>Tensor</code>.</td></tr> </table>	Element ID	Element	auxData	Pointer to tensor X of size $n_1 \times 1 \times \dots \times 1$ that stores probabilities for the forward loss logistic cross-entropy layer. This input can be an object of any class derived from <code>Tensor</code> .	auxGroundTruth	Pointer to tensor T of size n_1 or $n_1 \times 1 \times \dots \times 1$ that stores the ground truth data for the forward loss logistic cross-entropy layer. This input can be an object of any class derived from <code>Tensor</code> .
Element ID	Element						
auxData	Pointer to tensor X of size $n_1 \times 1 \times \dots \times 1$ that stores probabilities for the forward loss logistic cross-entropy layer. This input can be an object of any class derived from <code>Tensor</code> .						
auxGroundTruth	Pointer to tensor T of size n_1 or $n_1 \times 1 \times \dots \times 1$ that stores the ground truth data for the forward loss logistic cross-entropy layer. This input can be an object of any class derived from <code>Tensor</code> .						

NOTE

The `inputGradient` field stores a null pointer since there is no input gradient on the last layer.

Layer Parameters

For common parameters of neural network layers, see [Common Parameters](#).

In addition to the common parameters, the backward loss logistic cross-entropy layer has the following parameters:

Parameter	Default Value	Description
<code>algorithmFPType</code>	<code>float</code>	The floating-point type that the algorithm uses for intermediate computations. Can be <code>float</code> or <code>double</code> .
<code>method</code>	<code>defaultDense</code>	Performance-oriented computation method, the only method supported by the layer.

Layer Output

The backward loss logistic cross-entropy layer calculates the result described below. Pass the Result ID as a parameter to the methods that access the results of your algorithm. For more details, see [Algorithms](#).

Result ID	Result
<code>gradient</code>	Pointer to tensor Z of size $n_1 \times 1 \times \dots \times 1$ that stores the result of the backward loss logistic cross-entropy layer. This input can be an object of any class derived from <code>Tensor</code> .

Examples

Refer to the following examples in your Intel DAAL directory:

C++: `./examples/cpp/source/neural_networks/loss_logistic_entr_layer_dense_batch.cpp`

Java*: `./examples/java/source/com/intel/daal/examples/neural_networks/
LossLogisticCrossEntropyLayerBatch.java`

Python*: `./examples/python/source/neural_networks/
loss_logistic_entr_layer_dense_batch.py`

Services

Classes and utilities included in the Services component of the Intel® Data Analytics Acceleration Library (Intel® DAAL) are subdivided into the following groups according to their purpose:

- [Extract Version Information](#)
- [Handle Errors](#)
- [Manage Memory](#)
- [Manage the Computational Environment](#)

Extracting Version Information

The Services component provides methods that enable you to extract information about the version of Intel DAAL. You can get the following information about the installed version of the library from the fields of the `LibraryVersionInfo` structure:

Field Name	Description
<code>majorVersion</code>	Major version of the library
<code>minorVersion</code>	Minor version of the library
<code>updateVersion</code>	Update version of the library
<code>productStatus</code>	Status of the library: alpha, beta, or product
<code>build</code>	Build number
<code>name</code>	Library name
<code>processor</code>	Processor optimization

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Examples

Refer to the following examples in your Intel DAAL directory:

C++: `./examples/cpp/source/services/library_version_info.cpp`

Java*: `./examples/java/source/com/intel/daal/examples/services/LibraryVersionInfoExample.java`

Python*: `./examples/python/source/services/library_version_info.py`

Handling Errors

Intel DAAL provides classes and methods to handle exceptions or errors that can occur during library operation.

In Intel DAAL C++ interfaces, the base class for error handling is `Error`. This class contains an error message and details of the issue. For example, an `Error` object can store the number of the row in the `NumericTable` that caused the issue or a message that an SQL database generated to describe the reasons of an unsuccessful query. A single `Error` object can store the error description and an arbitrary number of details of various types: integer or double values or strings.

Most of Intel DAAL classes, such as `Algorithm` classes, contain an `ErrorCollection` class, which handles errors for the class. In addition to the functionality of the `Error` class, `ErrorCollection` creates an integrated error description and throws run-time exceptions. Some of the Intel DAAL classes, such as `NumericTable`, contain a `KernelErrorCollection` class. This class functions similarly to `ErrorCollection`, but does not throw run-time exceptions.

By default, when you add an error to the `ErrorCollection` object, `ErrorCollection` throws a run-time exception. To prevent throwing any exceptions, you can set the `DAAL_NOTHROW_EXCEPTIONS` flag. If the flag is set, use `isEmpty()` and `getErrors()` methods of the `ErrorCollection` or `KernelErrorCollection` class to check whether any errors occur. To get a concatenated error description, call the `getDescription()` method of the `KernelErrorCollection` or `ErrorCollection` class.

Intel DAAL Java* interfaces handle errors by throwing Java exceptions.

Examples

Refer to the following examples in your Intel DAAL directory:

C++:

- `./examples/cpp/source/error_handling/error_handling_nothrow.cpp`
- `./examples/cpp/source/error_handling/error_handling_throw.cpp`

Java*: `./examples/java/source/com/intel/daal/examples/error_handling/ErrorHandler.java`

Python*: `./examples/python/source/error_handling/error_handling_throw.py`

Managing Memory

To improve performance of your application that calls Intel DAAL, align your arrays on 64-byte boundaries and ensure that the leading dimensions of the arrays are divisible by 64. For that purpose Intel DAAL provides `daal_malloc()` and `daal_free()` functions to allocate and deallocate memory.

To allocate memory, call `daal_malloc()` with the specified size of the buffer to be allocated and the alignment of the buffer, which must be a power of 2. If the specified alignment is not a power of 2, the library uses the 32-byte alignment.

To deallocate memory allocated earlier by the `daal_malloc()` function, call the `daal_free()` function and set a pointer to the buffer to be freed.

Managing the Computational Environment

Intel DAAL provides the `Environment` class to manage settings of the computational environment in which the application that uses the library runs. The methods of this class enable you to specify the number of threads for the application to use or to check the type of the processor running the application. The

`Environment` class is a singleton, which ensures that only one instance of the class is available in the Intel DAAL based application. To access the instance of the class, call the `getInstance()` method which returns a pointer to the instance. Once you get the instance of the class, you can use it for multiple purposes:

- Detect the processor type.
To do this, call the `getCpuId()` method.
- Detect and modify the number of threads used by the Intel DAAL based application.
To do this, call the `getNumberOfThreads()` or `setNumberOfThreads()` method, respectively.
- Specify the single-threaded or multi-threaded mode for Intel DAAL on Windows.
To do this, call to the `setDynamicLibraryThreadingTypeOnWindows()` method.

Call the `freeInstance()` method of the `Environment` class when you complete configuring the computational environment.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Examples

Refer to the following examples in your Intel DAAL directory:

C++: `./examples/cpp/source/set_number_of_threads/set_number_of_threads.cpp`

Java*: `./examples/java/source/com/intel/daal/examples/set_number_of_threads/SetNumberOfThreads.java`

Python*: `./examples/python/source/set_number_of_threads/set_number_of_threads.py`

Bibliography

For more information about algorithms implemented in the Intel® Data Analytics Acceleration Library, refer to the following publications:

- [Agrawal94] Rakesh Agrawal, Ramakrishnan Srikant. *Fast Algorithms for Mining Association Rules*. Proceedings of the 20th VLDB Conference Santiago, Chile, 1994.
- [Arthur2007] Arthur, D., Vassilvitskii, S. *k-means++: The Advantages of Careful Seeding*. Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms. Society for Industrial and Applied Mathematics Philadelphia, PA, USA, 2007, pp. 1027-1035. Available from <http://ilpubs.stanford.edu:8090/778/1/2006-13.pdf>.
- [Bahmani2012] B. Bahmani, B. Moseley, A. Vattani, R. Kumar, S. Vassilvitskii. *Scalable K-means++*. Proceedings of the VLDB Endowment, 2012. Available from http://vldb.org/pvldb/vol5/p622_bahmanbahmani_vldb2012.pdf.
- [Ben05] Ben-Gal I. *Outlier detection*. In: Maimon O. and Rockach L. (Eds.) *Data Mining and Knowledge Discovery Handbook: A Complete Guide for Practitioners and Researchers*", Kluwer Academic Publishers, 2005, ISBN 0-387-24435-2.
- [Biernacki2003] C. Biernacki, G. Celeux, and G. Govaert. *Choosing Starting Values for the EM Algorithm for Getting the Highest Likelihood in Multivariate Gaussian Mixture Models*. Computational Statistics & Data Analysis, 41, 561-575, 2003.
- [Billor2000] Nedret Billor, Ali S. Hadib, and Paul F. Velleman. *BACON: blocked adaptive computationally efficient outlier nominators*. Computational Statistics & Data Analysis, 34, 279-298, 2000.
- [Bishop2006] Christopher M. Bishop. *Pattern Recognition and Machine Learning*, p.198, Computational Statistics & Data Analysis, 34, 279-298, 2000. Springer Science +Business Media, LLC, ISBN-10: 0-387-31073-8, 2006.
- [Boser92] B. E. Boser, I. Guyon, and V. Vapnik. *A training algorithm for optimal margin classifiers*. Proceedings of the Fifth Annual Workshop on Computational Learning Theory, pp: 144-152, ACM Press, 1992.
- [Byrd2015] R. H. Byrd, S. L. Hansen, Jorge Nocedal, Y. Singer. *A Stochastic Quasi-Newton Method for Large-Scale Optimization*, 2015. arXiv:1401.7020v2 [math.OC]. Available from <http://arxiv.org/abs/1401.7020v2>.
- [bzip2] <http://www.bzip.org/>
- [Dempster77] A.P.Dempster, N.M. Laird, and D.B. Rubin. *Maximum-likelihood from incomplete data via the em algorithm*. J. Royal Statist. Soc. Ser. B., 39, 1977.
- [Duchi2011] Elad Hazan, John Duchi, and Yoram Singer. *Adaptive subgradient methods for online learning and stochastic optimization*. The Journal of Machine Learning Research, 12:2121-2159, 2011.
- [Fan05] Rong-En Fan, Pai-Hsuen Chen, Chih-Jen Lin. *Working Set Selection Using Second Order Information for Training Support Vector Machines*. Journal of Machine Learning Research 6 (2005), pp: 1889-1918.
- [Fleischer2008] Rudolf Fleischer, Jinhui Xu. *Algorithmic Aspects in Information and Management*. 4th International conference, AAIM 2008, Shanghai, China, June 23-25, 2008. Proceedings, Springer.

- [Freund99] Yoav Freund, Robert E. Schapire. *Additive Logistic regression: a statistical view of boosting*. Journal of Japanese Society for Artificial Intelligence (14(5)), 771-780, 1999.
- [Freund01] Yoav Freund. *An adaptive version of the boost by majority algorithm*. Machine Learning (43), pp. 293-318, 2001.
- [Friedman00] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. *Additive Logistic regression: a statistical view of boosting*. The Annals of Statistics, 28(2), pp: 337-407, 2000.
- [Glorot2010] Xavier Glorot and Yoshua Bengio. *Understanding the difficulty of training deep feedforward neural networks*. International conference on artificial intelligence and statistics, 2010.
- [GregorLecun2010] Karol Gregor, Yann LeCun. *Emergence of Complex-Like Cells in a Temporal Product Network with Local Receptive Fields*. ArXiv:1006.0448v1 [cs.NE] 2 Jun 2010.
- [Hastie2009] Trevor Hastie, Robert Tibshirani, Jerome Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Second Edition (Springer Series in Statistics), Springer, 2009. Corr. 7th printing 2013 edition (December 23, 2011).
- [HeZhangRenSun] Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun. *Delving Deep into Rectifiers: Surpassing Human-Level Performance on Image Net*, arXiv: 1502.01852v1 [cs.CV] 6 Feb 201, available from <https://arxiv.org/pdf/1502.01852.pdf>.
- [Hoerl70] Arthur E. Hoerl and Robert W. Kennard. *Ridge Regression: Biased Estimation for Nonorthogonal Problems*. Technometrics, Vol. 12, No. 1 (Feb., 1970), pp. 55-67.
- [Hsu02] Chih-Wei Hsu and Chih-Jen Lin. *A Comparison of Methods for Multiclass Support Vector Machines*. IEEE Transactions on Neural Networks, Vol. 13, No. 2, pp: 415-425, 2002.
- [Iba92] Wayne Iba, Pat Langley. *Induction of One-Level Decision Trees*. Proceedings of Ninth International Conference on Machine Learning, pp: 233-240, 1992.
- [Ioffe2015] Sergey Ioffe, Christian Szegedy. *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*, arXiv:1502.03167v3 [cs.LG] 2 Mar 2015, available from <http://arxiv.org/pdf/1502.03167.pdf>.
- [James2013] Gareth James, Daniela Witten, Trevor Hastie, and Rob Tibshirani. *An Introduction to Statistical Learning with Applications in R*. Springer Series in Statistics, Springer, 2013 (Corrected at 6th printing 2015).
- [Jarrett2009] K. Jarrett, K. Kavukcuoglu, M. A. Ranzato, and Y. LeCun. *What is the best multi-stage architecture for object recognition?* International Conference on Computer Vision, pp. 2146-2153, IEEE, 2009.
- [Joachims99] Thorsten Joachims. *Making Large-Scale SVM Learning Practical*. Advances in Kernel Methods - Support Vector Learning, B. Schölkopf, C. Burges, and A. Smola (ed.), pp: 169 – 184, MIT Press Cambridge, MA, USA 1999.
- [Krizh2012] Alex Krizhevsky, Ilya Sutskever, Geoffrey E. Hinton. *ImageNet Classification with Deep Convolutional Neural Networks*. Available from <http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf>.
- [LeCun15] Yann LeCun, Yoshua Bengio, Geoffrey E. Hinton. *Deep Learning*. Nature (521), pp. 436-444, 2015.

- [Lloyd82] Stuart P Lloyd. *Least squares quantization in PCM*. IEEE Transactions on Information Theory 1982, 28 (2): 1982pp: 129–137.
- [lzo] <http://www.oberhumer.com/opensource/lzo/>
- [Maitra2009] Maitra, R. *Initializing Optimization Partitioning Algorithms*. ACM/IEEE Transactions on Computational Biology and Bioinformatics 2009, 6 (1): pp: 144-157.
- [Matthew2013] Matthew D. Zeiler, Rob Fergus. *Stochastic Pooling for Regularization of Deep Convolutional Neural Networks*. 2013. Available from <http://www.matthewzeiler.com/pubs/iclr2013/iclr2013.pdf>.
- [Mu2014] Mu Li, Tong Zhang, Yuqiang Chen, Alexander J. Smola. *Efficient Mini-batch Training for Stochastic Optimization*, 2014. Available from https://www.cs.cmu.edu/~muli/file/minibatch_sgd.pdf.
- [Patwary2016] Md. Mostofa Ali Patwary, Nadathur Rajagopalan Satish, Narayanan Sundaram, Jialin Liu, Peter Sadowski, Evan Racah, Suren Byna, Craig Tull, Wahid Bhimji, Prabhat, Pradeep Dubey. *PANDA: Extreme Scale Parallel K-Nearest Neighbor on Distributed Architectures*, 2016. Available from <https://arxiv.org/abs/1607.08220>.
- [Renie03] Jason D.M. Rennie, Lawrence, Shih, Jaime Teevan, David R. Karger. *Tackling the Poor Assumptions of Naïve Bayes Text classifiers*. Proceedings of the Twentieth International Conference on Machine Learning (ICML-2003), Washington DC, 2003.
- [rle] <http://data-compression.info/Algorithms/RLE/index.htm>
- [Rumelhart86] David E. Rumelhart, Geoffrey E. Hinton, Ronald J. Williams. *Learning representations by back-propagating errors*. Nature (323), pp. 533-536, 1986.
- [Sokolova09] Marina Sokolova, Guy Lapalme. *A systematic analysis of performance measures for classification tasks*. Information Processing and Management 45 (2009), pp. 427–437. Available from <http://atour.iro.umontreal.ca/rali/sites/default/files/publis/SokolovaLapalme-JIPM09.pdf>.
- [Szegedy13] Christian Szegedy, Alexander Toshev, Dumitru Erhan. *Scalable Object Detection Using Deep Neural Networks*. Advances in Neural Information Processing Systems, 2013.
- [West79] D.H.D. West. *Updating Mean and Variance Estimates: An improved method*. Communications of ACM, 22(9), pp: 532-535, 1979.
- [Wu04] Ting-Fan Wu, Chih-Jen Lin, Ruby C. Weng. *Probability Estimates for Multi-class Classification by Pairwise Coupling*. Journal of Machine Learning Research 5, pp: 975-1005, 2004.
- [zLib] <http://www.zlib.net/>

Index

1D average pooling backward layer, in neural network
 usage in batch processing mode328
 1D average pooling forward layer, in neural network
 usage in batch processing mode326
 1D max pooling backward layer, in neural network
 usage in batch processing mode314
 1D max pooling forward layer, in neural network
 usage in batch processing mode311
 2D average pooling backward layer, in neural network
 usage in batch processing mode332
 2D average pooling forward layer, in neural network
 usage in batch processing mode330
 2D convolution backward layer, in neural network
 usage in batch processing mode353
 2D convolution forward layer, in neural network
 usage in batch processing mode350
 2D locally-connected backward layer, in neural network
 usage in batch processing mode365
 2D locally-connected forward layer, in neural network
 usage in batch processing mode361
 2D max pooling backward layer, in neural network
 usage in batch processing mode319
 2D max pooling forward layer, in neural network
 usage in batch processing mode316
 2D spatial pyramid pooling backward layer, in neural
 network
 usage in batch processing mode347
 2D spatial pyramid pooling forward layer, in neural
 network
 usage in batch processing mode344
 2D stochastic pooling backward layer, in neural network
 usage in batch processing mode341
 2D stochastic pooling forward layer, in neural network
 usage in batch processing mode339
 2D transposed convolution backward layer, in neural
 network
 usage in batch processing mode358
 2D transposed convolution forward layer, in neural
 network
 usage in batch processing mode356
 3D average pooling backward layer, in neural network
 usage in batch processing mode336
 3D average pooling forward layer, in neural network
 usage in batch processing mode334
 3D max pooling backward layer, in neural network
 usage in batch processing mode324
 3D max pooling forward layer, in neural network
 usage in batch processing mode321

A

absolute value (abs) backward layer, in neural network
 usage in batch processing mode274
 absolute value (abs) forward layer, in neural network
 usage in batch processing mode273
 absolute value function
 definition168
 usage in batch processing mode168
 AdaBoost classifier
 definition219
 in brief219
 usage in batch processing mode220
 AdaGrad method

 computation188
 adaptive subgradient method
 computation188
 definition187
 association rules
 definition118
 in brief118
 performance considerations121
 usage in batch processing mode118

B

batch normalization backward layer, in neural network
 usage in batch processing mode294
 batch normalization forward layer, in neural network
 usage in batch processing mode290
 bibliography393
 boosting algorithms
 definition217
 performance considerations217
 BrownBoost classifier
 definition221
 in brief221
 usage in batch processing mode222

C

Cholesky decomposition
 definition107
 in brief107
 performance considerations108
 usage in batch processing mode107
 classification
 in brief204
 usage model204
 compression of data32
 computation modes38
 concat backward layer, in neural network
 usage in batch processing mode371
 concat forward layer, in neural network
 usage in batch processing mode370
 correlation distance
 definition68
 in brief68
 performance considerations69
 usage in batch processing mode69
 correlation matrix
 in brief58
 performance considerations66
 usage in batch processing mode58
 usage in distributed processing mode63
 usage in online processing mode60
 cosine distance
 definition67
 in brief67
 performance considerations68
 usage in batch processing mode67

D

data dictionary30
 data model35
 data source29
 deserialization31

dropout backward layer, in neural network
 usage in batch processing mode310
dropout forward layer, in neural network
 usage in batch processing mode308

E

EM
 computation in batch processing mode136
 definition133
 in brief133
 initialization
 in batch processing mode135
 performance considerations138
environment, computational, management390
error handling390
expectation-maximization
 computation in batch processing mode136
 definition133
 in brief133
 initialization
 in batch processing mode135
 performance considerations138

F

fully-connected backward layer, in neural network
 usage in batch processing mode271
fully-connected forward layer, in neural network
 usage in batch processing mode269

H

hyperbolic tangent backward layer, in neural network
 usage in batch processing mode288
hyperbolic tangent forward layer, in neural network
 usage in batch processing mode287
hyperbolic tangent function
 definition166
 usage in batch processing mode166

I

implicit ALS
 computation in batch processing mode239
 computation in distributed processing mode240
 definition236
 in brief236
 initialization in batch processing mode237
 initialization in distributed processing mode237
 performance considerations250
 ratings prediction in distributed processing mode248
 training in distributed processing mode240
implicit alternating least squares
 computation in batch processing mode239
 computation in distributed processing mode240
 definition236
 in brief236
 initialization in batch processing mode237
 initialization in distributed processing mode237
 performance considerations250
 ratings prediction in distributed processing mode248
 training in distributed processing mode240
iterative solver
 definition177
iterative solver algorithm
 computation178

K

K-Means clustering
 computation in batch processing mode89
 computation in distributed processing mode91
 definition70
 in brief70
 initialization in batch processing mode72
 initialization in distributed processing mode73
 performance considerations97

k-Nearest Neighbors classifier
 definition231
 in brief231
 usage in batch processing mode232
kernel function
 in brief144
 linear
 definition144
 usage in batch processing mode145
 RBF
 definition146
 usage in batch processing mode147
kNN classifier
 definition231
 in brief231
 usage in batch processing mode232

L

layers, in neural network
 1D average pooling backward327
 1D average pooling forward325
 1D max pooling backward313
 1D max pooling forward311
 2D average pooling backward331
 2D average pooling forward329
 2D convolution backward352
 2D convolution forward349
 2D locally-connected backward363
 2D locally-connected forward360
 2D max pooling backward318
 2D max pooling forward315
 2D spatial pyramid pooling backward346
 2D spatial pyramid pooling forward343
 2D stochastic pooling backward340
 2D stochastic pooling forward337
 2D transposed convolution backward357
 2D transposed convolution forward355
 3D average pooling backward335
 3D average pooling forward333
 3D max pooling backward323
 3D max pooling forward321
 absolute value (abs) backward274
 absolute value (abs) forward273
 batch normalization backward292
 batch normalization forward289
 common parameters267
 concat backward370
 concat forward369
 dropout backward309
 dropout forward308
 fully-connected backward270
 fully-connected forward268
 hyperbolic tangent backward288
 hyperbolic tangent forward286
 local contrast normalization backward302
 local contrast normalization forward299
 local response normalization backward297
 local response normalization forward296
 logistic backward277
 logistic forward275
 loss backward378
 loss forward377
 loss logistic cross-entropy backward385
 loss logistic cross-entropy forward383
 loss softmax cross-entropy backward381
 loss softmax cross-entropy forward379
 pReLU backward280
 pReLU forward278

- rectifier linear unit (ReLU) backward283
- rectifier linear unit (ReLU) forward281
- reshape backward368
- reshape forward367
- SmoothReLU backward285
- SmoothReLU forward284
- softmax backward375
- softmax forward374
- split backward373
- split forward372
- limited-memory BFGS
 - computation184
 - definition183
- limited-memory Broyden-Fletcher-Goldfarb-Shanno
 - algorithm
 - computation184
- linear kernel
 - usage in batch processing mode145
- linear kernel function
 - in brief144
- linear regression
 - definition194
 - usage in batch processing mode195
 - usage in distributed processing mode199
 - usage in online processing mode197
- local contrast normalization backward layer, in neural
 - network
 - usage in batch processing mode306
- local contrast normalization forward layer, in neural
 - network
 - usage in batch processing mode301
- local response normalization backward layer, in neural
 - network
 - usage in batch processing mode298
- local response normalization forward layer, in neural
 - network
 - usage in batch processing mode296
- logistic backward layer, in neural network
 - usage in batch processing mode277
- logistic forward layer, in neural network
 - usage in batch processing mode276
- logistic function
 - definition164
 - usage in batch processing mode164
- LogitBoost classifier
 - definition224
 - in brief223
 - usage in batch processing mode224
- loss backward layer, in neural network
 - usage in batch processing mode378
- loss forward layer, in neural network
 - usage in batch processing mode377
- loss logistic cross-entropy backward layer, in neural
 - network
 - usage in batch processing mode385
- loss logistic cross-entropy forward layer, in neural network
 - usage in batch processing mode384
- loss softmax cross-entropy backward layer, in neural
 - network
 - usage in batch processing mode381
- loss softmax cross-entropy forward layer, in neural
 - network
 - usage in batch processing mode379
- low order moments
 - definition47
 - in brief47
 - performance considerations56
 - usage in batch processing mode48
 - usage in distributed processing mode53
 - usage in online processing mode50

M

- manage computational environment390
- math functions
 - absolute value168
 - hyperbolic tangent166
 - in Intel(R) Data Analytics Acceleration Library164
 - logistic164
 - performance considerations171
 - ReLU169
 - SmoothReLU170
 - softmax165
- mean squared error algorithm
 - computation175
 - definition174
- memory management390
- min-max normalization
 - definition162
 - usage in batch processing mode162
- model, in Intel® Data Analytics Acceleration Library35
- moments of low order
 - definition47
 - in brief47
 - performance considerations56
 - usage in batch processing mode48
 - usage in distributed processing mode53
 - usage in online processing mode50
- MSE algorithm
 - computation175
 - definition174
- multi-class classifier
 - definition229
 - in brief229
 - usage in batch processing mode229
- multivariate outlier detection
 - in brief139
 - performance considerations142
 - usage in batch processing mode139

N

- Naive Bayes classifier
 - definition208
 - in brief208
 - performance considerations216
 - usage in batch processing mode208
 - usage in distributed processing mode211
 - usage in online processing mode210
- neural network
 - concept251
 - Gaussian initializer265
 - initializer263
 - initializer, Gaussian265
 - initializer, truncated Gaussian265
 - initializer, uniform264
 - initializer, Xavier264
 - layers266
 - prediction in batch processing mode255
 - training in batch processing mode254
 - training in distributed processing mode259
 - truncated Gaussian initializer265
 - uniform initializer264
 - usage model252
 - Xavier initializer264
- normalization
 - min-max162
 - Z-score160
- normalization algorithms
 - performance considerations163
- numeric table20

O

- objective function
 - definition172
- Objective function
 - computation172
- objective function with precomputed characteristics
 - algorithm
 - computation176
 - definition176
- optimization solver
 - adaptive subgradient187
 - in brief172
 - iterative solver177
 - limited-memory BFGS183
 - limited-memory Broyden-Fletcher-Goldfarb-Shanno algorithm183
 - mean squared error174
 - MSE174
 - objective function172
 - objective function with precomputed characteristics algorithm176
 - performance considerations189
 - stochastic gradient descent179
 - sum of functions173
- outlier detection
 - definition139
 - multivariate, in brief139
 - multivariate, usage in batch processing mode139
 - univariate, definition142
 - univariate, in brief142
 - univariate, usage in batch processing mode143
- overview, of Intel® Data Analytics Acceleration Library13

P

- PCA
 - definition97
 - in brief97
 - performance106
 - usage in batch processing mode97
 - usage in distributed processing mode102
 - usage in online processing mode99
- performance
 - of association rules121
 - of boosting algorithms217
 - of Cholesky decomposition108
 - of correlation and variance-covariance matrices66
 - of correlation distance matrices69
 - of cosine distance matrices68
 - of expectation-maximization138
 - of implicit alternating least squares recommender250
 - of K-Means clustering97
 - of low order moments56
 - of math functions171
 - of multivariate outlier detection142
 - of Naive Bayes classifier216
 - of normalization algorithms163
 - of optimization solver189
 - of principal component analysis106
 - of QR decomposition132
 - of singular value decomposition117
 - of stump weak learner classifier219
 - of support vector machine classifier228
 - of univariate outlier detection144
- pReLU backward layer, in neural network
 - usage in batch processing mode280
- pReLU forward layer, in neural network

- usage in batch processing mode278
- principal component analysis
 - definition97
 - in brief97
 - performance106
 - usage in batch processing mode97
 - usage in distributed processing mode102
 - usage in online processing mode99

Q

- QR decomposition
 - in brief121
 - performance132
- QR decomposition without pivoting
 - definition122
 - usage in batch processing mode122
 - usage in distributed processing mode123
 - usage in online processing mode122
- QR decomposition, pivoted
 - definition130
 - usage in batch processing mode130
- quality metrics
 - for binary classifiers148
 - for binary classifiers, computation in batch processing mode149
 - for classifiers, user-defined159
 - for linear regression153
 - for linear regression, computation in batch processing mode155
 - for multi-class classifiers151
 - for multi-class classifiers, computation in batch processing mode152
 - in brief148
- quantile
 - definition56
 - in brief56
 - usage in batch processing mode57

R

- radial basis function kernel
 - definition146
 - usage in batch processing mode147
- RBF kernel function
 - definition146
 - usage in batch processing mode147
- recommendation system
 - usage model233
- recommendation systems233
- rectifier linear unit (ReLU) backward layer, in neural network
 - usage in batch processing mode283
- rectifier linear unit (ReLU) forward layer, in neural network
 - usage in batch processing mode282
- regression
 - in brief190
 - usage model191
- ReLU function
 - definition169
- ReLU value function
 - usage in batch processing mode169
- reshape backward layer, in neural network
 - usage in batch processing mode368
- reshape forward layer, in neural network
 - usage in batch processing mode367
- ridge regression
 - definition194
 - usage in batch processing mode195

usage in distributed processing mode199
usage in online processing mode197

S

serialization31
singular value decomposition
 definition109
 in brief108
 performance117
 usage in batch processing mode109
 usage in distributed processing mode110
 usage in online processing mode109
SmoothReLU backward layer, in neural network
 usage in batch processing mode285
SmoothReLU forward layer, in neural network
 usage in batch processing mode284
SmoothReLU function
 definition170
 usage in batch processing mode171
softmax backward layer, in neural network
 usage in batch processing mode376
softmax forward layer, in neural network
 usage in batch processing mode375
softmax function
 definition165
 usage in batch processing mode165
sorting
 definition159
 in brief159
 usage in batch processing mode159
split backward layer, in neural network
 usage in batch processing mode373
split forward layer, in neural network
 usage in batch processing mode372
stochastic gradient descent algorithm
 computation180
 definition179
stump weak learner classifier
 definition218
 performance considerations219
 usage in batch processing mode218
sum of functions
 computation174
 definition173
support vector machine classifier
 definition226
 in brief226
 performance considerations228

usage in batch processing mode227

SVD
 definition109
 in brief108
 performance117
 usage in batch processing mode109
 usage in distributed processing mode110
 usage in online processing mode109
SVM classifier
 definition226
 in brief226
 performance considerations228
 usage in batch processing mode227

T

tensor28

U

univariate outlier detection
 definition142
 in brief142
 performance considerations144
 usage in batch processing mode143

V

variance-covariance matrix
 in brief58
 performance considerations66
 usage in batch processing mode58
 usage in distributed processing mode63
 usage in online processing mode60
version information extraction389

W

weak learner
 definition217

Z

Z-score normalization
 definition160
 usage in batch processing mode161