Squirrel Manual

July 19, 2017

Contents

1 The Core Module

The `Core` module offers several classes that are used in most other modules. Particularly, it provides functionality to read `config` files, define external parameters in classes, write `log` files, print error messages, and to create instances for easy file I/O.

1.1 Types

see Core/Types.hh

Defines the typenames of variables used throughout the framework, e.g. `u32` for `unsigned int` or `f32` for a 32-bit `float`. Moreover, the following functions return type limits and `NaN` checks:

`static const T min()`
return the minimal value for a variable of type `T`, e.g. `Types::min<f32>()` is the most negative float number possible

`static const T max()`
return the maximal value for a variable of type `T`

`static const T absmin()`
return the smallest value in magnitude for type `T`, e.g. `Types::absmin<f32>()` is the positive float value closest to zero

`static const bool isNan(T value)`	
`value`	a floating point number
`return`	`true` if `value = NaN`, else `false`

`static const T inf()`
return the ∞ representation of a floating point type `T`

1.2 Configuration and Parameters

see Configuration.hh and Parameters.hh

The framework offers an easy way to define parameters within classes and to read them from the command line or from a config file.

1.2.1 Specifying Parameters in the Code

There is a parameter class for a variety of primitive datatypes:

Parameter specifications	
`Core::ParameterInt`	a `s32` (signed int) parameter
`Core::ParameterFloat`	a `f32` (32-bit float) parameter
`Core::ParameterChar`	a `char` parameter
`Core::ParameterBool`	a `bool` parameter
`Core::ParameterString`	a `std::string` parameter
`Core::ParameterIntList`	a `std::vector<s32>` parameter
`Core::ParameterFloatList`	a `std::vector<f32>` parameter
`Core::ParameterCharList`	a `std::vector<char>` parameter
`Core::ParameterBoolList`	a `std::vector<bool>` parameter
`Core::ParameterStringList`	a `std::vector<std::string` parameter
`Core::ParameterEnum`	an `enum` parameter with some special properties, see below

In the code, each of these parameters can be instanciated in the same way. All primitive types and all list types follow the same concept, which is here illustrated with `ParameterFloat` and `ParameterFloatList`.

Consider the following class definition in a header file `MyClass.hh`:

```
#include <Core/CommonHeaders.hh>

class MyClass {
private:
    static const Core::ParameterFloat myFloatParam_;
    static const Core::ParameterFloatList myFloatListParam_;
    Float myFloat_;
    std::vector<Float> myFloatList_;
public:
    MyClass();
}
```

The parameter instanciation in `MyClass.cc` looks like this:

```
#include "MyClass.hh"

const Core::ParameterFloat MyClass::myFloatParam_("my-float", 0.0, "foo");
const Core::ParameterFloatList MyClass::myFloatListParam_("my-float-list",
                                                  "1.0, 2.0, 3.0", "foo");
// constructor
MyClass::MyClass() :
    myFloat_(Core::Configuration::config(myFloatParam_);
    myFloatList_(Core::Configuration::config(myFloatListParam_, "bar");
{}
```

There are three values passed to the specific parameter class. The first is the name of the parameter, the second is a default value, and the third is a prefix. So, when running the program with

```
./program --foo.my-float=1.0
```

the value of `myFloat_` will be set to 1.0 in the constructor, while `myFloatList_` will back up to the default values which is the list $[1.0, 2.0, 3.0]$. The third parameter, the prefix, can also be altered by specifying another value when getting the parameter with `Core::Configuration::config(...)` in the constructor. Particularly,

```
./program --bar.my-float-list=2.0,4.0
```

would result in `myFloatList_` being the list $[2.0, 4.0]$, whereas

```
./program --foo.my-float-list=2.0,4.0
```

would not be the correct path to address the `my-float-list` parameter, so in the end `myFloatList_` would contain the default value $[1.0, 2.0, 3.0]$. Be aware that for list parameters, the default option is always given as a comma separated string.

ParameterEnum

`ParameterEnum` allows to use `enums` as configurable parameters. The parameter configuration is quite similar to the regular parameters, but has some special requirements. A simple example will show the difference.

```
// Header file MyClass.hh
#include <Core/CommonHeaders.hh>

class MyClass {
private:
    static const Core::ParameterEnum myEnumParam_;
    enum MyEnum { optionA, optionB, optionC };
    MyEnum option_;
public:
    MyClass();
}
```

```
// Source file MyClass.cc
#include "MyClass.hh"

const Core::ParameterEnum MyClass::myEnumParam_(
                       "my-enum", "optionA, optionB, optionC", "optionA", "foo");
// constructor
MyClass::MyClass() :
    option_((MyEnum) Core::Configuration::config(myEnumParam_))
{}
```

For `ParameterEnum`, four arguments are required. The first is again the parameter name, the second is the list of all possible options, the third is the default option, and the fourth is again the prefix to address the parameter. Note that the second parameter, the list of options, has to be in the same order as the options defined in `MyEnum`.

1.2.2 Command Line Parameters and Configuration Files

Passing parameters to the program is possible via two ways: either as command line parameters or via `config` files.

Command Line Parameters

Command line parameters can be given in the format

```
--prefix.parameter-name=value
```

or

```
--*.parameter-name=value
```

Whitespaces and tabs are not allowed for command line parameters.

Config Files

A config file can be passed to the program using the command line parameter `--config=<config-file>`. Config files are read line-by-line with the following specification, where tabs and whitespaces are ignored:

Comments `#` starts a comment:

```
# this is a comment
parameter-name = value # this is another comment
```

Comments can be anywhere in the config file. Everything in a line coming after a `#` will be ignored.

Includes Other configuration files can be included via

```
include myConfigFile.config
```

Parameters Parameters can be specified via

```
prefix-path.parameter-name = parameter-value
```

or

```
*.parameter-name = parameter-value
```

First, it is searched for ¡prefix-path.parameter-name¿, then for ¡*.parameter-name¿. If none of them are found, the default value is used for the parameter

Global Prefixes Global prefixes can be defined via

```
[global-prefix]
```

All following parameter specifications are assumed to have this global prefix, e.g.

```
[global-prefix]
parameter-name = parameter-value
```

is interpreted as `global-prefix.parameter-path.parameter-name = parameter-value`. The global prefix is valid until a new one is defined. `[]` resets the prefix, i.e.

```
[]
parameter-name = parameter-value
```

is interpreted as `--paramter-name=parameter-value` (without any preceding context).

For examples, see a `config` file of one of the example setups.

1.2.3 Log and Error Messages

see Core/Log.hh and Core/Error.hh

The interfaces for log and error messages are very simple to use. In case of an error, use this syntax:

```
    if (error_condition) {
        Core::Error::msg("Something happend") << Core::Error::abort;
    }
```

The error message will be printed and the program exits.
For log messages, use the following syntax:

```
    u32 x = 10;
    // one line log message
    Core::Log::os("Variable x has value") << x;
    // xml-style log message
    Core::Log::openTag("my-tag");
    Core::Log::os("Variable x has value") << x;
    Core::Log::closeTag();
```

The output of this is

```
    Variable x has value 10
    <my-tag>
        Variable x has value 10
    </my-tag>
```

By default, log messages are written to `stdout`. If you want to write them to a file, specify the variable `log-file=<filename>`.

1.2.4 File I/O: IOStream

see Core/IOStream.hh

The `IOStream` class provides a simple interface for file I/O on binary, gzipped, and ascii files. There is a `AsciiStream`, a `CompressedStream`, and a `BinaryStream`, all implementing the same functions defined in `IOStream` for file handling.

`void open(const std::string& filename, const std::ios_base::openmode mode)`	
opens a file handle	
`filename`	the name of the file to be opened
`openmode`	either `std::ios::in` or `std::ios::out` for input or output

`bool is_open()`
return true if file is open

`void close()`

close the opened file

`bool eof()`

return true if openmode is `std::ios::in` and end-of-file is reached

`void endl(std::ostream& stream)`

`std::endl` version for the `IOStream` interface

`IOStream& operator<<(u8)`

write a `u8` unsigned short to the file and return a reference to the `IOStream` object. The same function also exists for datatypes `u32`, `u64`, `s8`, `s32`, `s64`, `f32`, `f64`, `bool`, `char`, `const char*`, `std::string`.

`IOStream& operator>>(u8&)`

read a `u8` unsigned short from the file and return a reference to the `IOStream` object. The same function also exists for datatypes `u32`, `u64`, `s8`, `s32`, `s64`, `f32`, `f64`, `bool`, and `char`.

`bool getline(std::string&)`

read a \n-terminated string from the file and return `true` on success

Code Example

```
/*
 * content of inputfile.txt:
 * Hello World.
 * 10
 *
 */
#include <Core/IOStream.hh>

u32 x;
std::string line;

// read input from ascii file
Core::AsciiStream in("inputfile.txt", std::ios::in);
in.getline(line); // read 'Hello World'
in >> x; // read '10'

// write to gzipped file
Core::CompressedStream out("outputfile.gz", std::ios::out);
out << x;
out << line;
```

2 The Math Module

In this chapter, the basic functions of the classes `Math::Matrix` and `Math::Vector` are documented.

2.1 Math::Matrix

`Math::Matrix<T>(u32 rows, u32 cols)`	
constructor. `rows` $\times$ `cols` elements of type `T` (usually `Float`) are allocated.	
`rows`	an u32 specifying the number of rows
`cols`	an u32 specifying the number of cols

2.1.1 Helper functions for Neural Networks

`void Math::Matrix::sigmoid(T gamma = 1.0)`	
applies the sigmoid function ($sigmoid(x) = \frac{1}{1+e^{-\gamma x}}$), to each element of the matrix	
`gamma`	the scaling factor for the sigmoid function. Default is 1.0

`void Math::Matrix::triangle()`
applies the triangle function ($triangle(x) = \|x\| \; if \; -1 \leq x \leq 1; \; 0 \; else$), to each element of the matrix

`void Math::Matrix::softmax()`
applies the softmax function ($softmax(x_{ij}) = \frac{e^{x_{ij}}}{\sum_k e^{x_{ik}}}$, where x_{ij} is j-th number in i-th column), columnwise to each element of the matrix

`T Math::Matrix::sum()`
returns sum of each element of the matrix.

`void Math::Matrix::max()`
applies the max function ($max(x_{ij}) = 1 \; if \; x_{ij} \geq x_{ik} \; \forall k; \; 0 \; else$, where x_{ij} is j-th number in the i-th column), columnwise to each element of the matrix

`void Math::Matrix::max(const Matrix<T> &A, const Matrix<T> &B)`	
assigns elementwise maximum from A and B to this matrix e.g $this_{ij} = maximum(A_{ij}, B_{ij})$	
`A`	the first input matrix
`B`	the second input matrix

```
void Math::Matrix::tanh()
```
applies the hyperbolic tangent function to each element of the matrix

```
void Math::Matrix::elementwiseMultiplicationWithSigmoidDerivative(
                                const Matrix<T> &X)
```
multiplies each element of this matrix with the derivative of the sigmoid function. e.g. $this_{ij} = this_{ij} * (X_{ij} * (1 - X_{ij}))$

`X`	The output of the sigmoid function

```
void Math::Matrix::elementwiseMultiplicationWithTanhDerivative(
                               const Matrix<T> &X)
```
multiplies each element of this matrix with the derivative of the tanh function. e.g. $this_{ij} = this_{ij} * (1 - X_{ij}^2)$

`X`	The output of the tanh function

```
void Math::Matrix::elementwiseMultiplicationWithLogDerivative(
                              const Matrix<T> &X)
```
multiplies each element of this matrix with the derivative of the log function. e.g. $this_{ij} = this_{ij}/e^{X_{ij}}$

`X`	The output of the log function

```
void Math::Matrix::elementwiseMultiplicationWithSignedPowDerivative(
                               const Matrix<T> &X, T p)
```
multiplies each element of this matrix with the derivative of the signedPower function. e.g. $this_{ij} = this_{ij} \times p \times |X_{ij}|^{p-1}$

`X`	The input of the signedPower function
`p`	The exponent

2.1.2 General mathematical functions

```
void Math::Matrix::exp()
```
exponentiate each element of the matrix e.g. $(exp(x) = e^x)$

```
void Math::Matrix::signedPow(T p)
```
applies power function to the absolute value of each element of the matrix and keeps the original sign e.g. $signedPow(x, p) = |x|^p \ if \ x \geq 0; -|x|^p \ else$

`p`	the exponent

```
void Math::Matrix::log()
```
applies the natural logrithm function to each element of the matrix

```
void Math::Matrix::sin()
```
applies the sin function to each element of the matrix

`void Math::Matrix::cos()`

applies the cos function to each element of the matrix

`void Math::Matrix::asin()`

applies the arcsin function to each element of the matrix

`void Math::Matrix::acos()`

applies the arccos function to each element of the matrix

`void Math::Matrix::abs()`

updates each element of the matrix with its absolute value

`T Math::Matrix::maxValue() const`

returns the maximum value in the matrix

`u32 Math::Matrix::argAbsMin(u32 column) const`

returns the index of the minimum absolute value in the column

`column`	the index of the column

`u32 Math::Matrix::argAbsMax(u32 column) const`

returns the index of the maximum absolute value in the column

`column`	the index of the column

`void Math::Matrix::argMax(Vector<S>& v) const`

saves the index of maximum value from each column of the matrix in rows of the vector

`v`	the vector which will contain the indecies of maximum values of columns

`void Math::Matrix::elementwiseMultiplication(const Matrix<T> &X)`

multiplies each element of this matrix with corresponding element of the input matrix e.g. $this_{ij} = this_{ij} \times X_{ij}$

`X`	the input matrix

`void Math::Matrix::elementwiseDivision(const Matrix<T> &X)`

divides each element of this matrix by corresponding element of the input matrix e.g. $this_{ij} = this_{ij}/X_{ij}$

`X`	the input matrix

`void Math::Matrix::addConstantElementwise(T C)`

adds the input constant C to each element of this matrix e.g. $this_{ij} = this_{ij} + C$

`C`	the input constant

```
void Math::Matrix::addToColumn(const Vector<T> &v, u32 column, T alpha = 1.0)
```
adds a scaled vector to a column of this matrix

`v`	the input vector
`column`	index of column to which vector should be added
`alpha`	scale factor. Default is 1.0

```
void Math::Matrix::addToRow(const Vector<T> &v, u32 row, T alpha = 1.0)
```
adds a scaled vector to a row of this matrix

`v`	the input vector
`row`	index of row to which vector should be added
`alpha`	scale factor. Default is 1.0

```
void Math::Matrix::multiplyColumnByScalar(u32 column, T alpha)
```
multiplies a column of this matrix by a scalar

`column`	the index of the column
`alpha`	input scalar

```
void Math::Matrix::multiplyRowByScalar(u32 row, T alpha)
```
multiplies a row of this matrix by a scalar

`row`	the index of the row
`alpha`	input scalar

```
void Math::Matrix::addToAllColumns(const Vector<T> &v, T alpha = 1.0)
```
adds a scaled vector to all columns of this matrix

`v`	the input vector
`alpha`	scale factor. Default is 1.0

```
void Math::Matrix::addToAllRows(const Vector<T> &v, T alpha = 1.0)
```
adds a scaled vector to all rows of this matrix

`v`	the input vector
`alpha`	scale factor. Default is 1.0

```
void Math::Matrix::multiplyColumnsByScalars(const Vector<T> &scalars)
```
scales each column of this matrix by a scalar, e.g. $this.col_i = this.col_i \times scalars_i$

`scalars`	the input vector, that contains scalars

```
void Math::Matrix::divideColumnsByScalars(const Vector<T> &scalars)
```
divides each column of this matrix by a scalar, e.g. $this.col_i = this.col_i / scalars_i$

`scalars`	the input vector, that contains scalars

```
void Math::Matrix::multiplyRowsByScalars(const Vector<T> &scalars)
```
scales each row of this matrix by a scalar, e.g. $this.row_i = this.row_i \times scalars_i$

`scalars`	the input vector, that contains scalars

```
void Math::Matrix::divideRowsByScalars(const Vector<T> &scalars)
```

divides each row of this matrix by a scalar, e.g. $this.row_i = this.row_i / scalars_i$

`scalars`	the input vector, that contains scalars

2.2 Math::Vector

```
void Math::Vector::addConstantElementwise(T c)
```

adds a constant to each element of this vector, e.g. $this_i = this_i + c$

`c`	a constant to add to each element

```
void Math::Vector::scale(T value)
```

scales this vector, e.g. $this_i = value \times this_i$

`value`	the scaling factor.

```
T Math::Vector::sumOfSquares() const
```

returns the sum of squares of this vector, e.g. $return\ this^T \times this$

```
T Math::Vector::dot(const Vector<T>& vector) const
```

returns scalar/dot product of this vector with the given vector, e.g. $return\ this^T \times vector$

`vector`	the input vector.

```
void Math::Vector::columnwiseSquaredEuclideanDistance(const Matrix<T>& A,
                                                      const Vector<T>& v)
```

computes the squared Euclidean distance of each column vector of the input matrix with the input vector, and stores results in this vector, e.g. $this_i = (A_i - v)^T (A_i - v)$

`A`	the input matrix.
`v`	the input vector.

```
void Math::Vector::multiply(const Matrix<T> &A, const Vector<T> &x,
                            bool transposed = false, T alpha = 1.0,
                                T beta = 0.0, u32 lda = 0) const
```

multiplies the input matrix or its transpose with the input vector and stores the result in this vector, e.g. $this = \alpha Ax + \beta this$ or $this = \alpha A^T x + \beta this$

`A`	the input matrix.
`x`	the input vector.
`transposed`	the input matrix should be transposed or not.
`alpha`	the scaling factor for the input matrix. Default is 1.0
`beta`	the scaling factor for the this vector. Default is 0.0

```
void Math::Vector::columnwiseInnerProduct(const Matrix<T>& A,
                                          const Matrix<T>& B)
```

computes inner product of each column vector of matrix A with the corresponding column vector of matrix B, and stores results in this vector, e.g. $this_i = A_i^T B_i$

`A`	the input matrix A
`B`	the input matrix B

```
void Math::Vector::elementwiseMultiplication(const Vector<T>& v)
```

multiplies each element of this vector with the corresponding element of the input vector, e.g. $this_i = this_i \times v_i$

`v`	the input vector.

```
void Math::Vector::elementwiseDivision(const Vector<T>& v)
```

divides each element of this vector with the corresponding element of the input vector, e.g. $this_i = this_i / v_i$

`v`	the input vector.

```
void Math::Vector::elementwiseDivision(const Vector<T>& v)
```

divides each element of this vector with the corresponding element of the input vector, e.g. $this_i = this_i / v_i$

`v`	the input vector.

```
void Math::Vector::setToZero()
```

sets each element of this vector to zero, e.g. $\forall i \; this_i = 0$

```
void Math::Vector::fill(T value)
```

sets each element of this vector to the input value, e.g. $\forall i \; this_i = value$

`value`	the input value.

```
void Math::Vector::ensureMinimalValue(const T threshold)
```

sets each element of this vector less than the threshold to threshold, e.g. $\forall i \; this_i = this_i \; if \; this_i \geq threshold; threshold \; else$

`threshold`	the input threshold.

```
u32 Math::Vector::argAbsMin() const
```

returns the index of absolute minimum value.

```
u32 Math::Vector::argAbsMax() const
```

returns the index of absolute maximum value.

```
T Math::Vector::max() const
```

returns the maximum value.

```
void Math::Vector::exp() const
```

applies the exponential function to each element of this vector, e.g $this_i = e^{this_i}$

```
void Math::Vector::signedPow(T p)
```

applies the signed power function to each element of this vector, e.g. $this_i = this_i^p \; if \; this_i \geq 0; -|this_i|^p \; else$

`p`	the exponent.

```
void Math::Vector::log() const
```

applies the log function to each element of this vector, e.g $this_i = log(this_i)$

```
void Math::Vector::abs() const
```

applies the absolute function to each element of this vector, e.g $this_i = |this_i|$

```
T Math::Vector::asum() const
```

returns the absolute sum over each element of this vector or L1 Norm of this vector, e.g. $return \; \sum_i |this_i|$

```
T Math::Vector::l1norm() const
```

returns the absolute sum over each element of this vector or L1 Norm of this vector, e.g. $return \; \sum_i |this_i|$

```
T Math::Vector::sum() const
```

returns the sum over each element of this vector, e.g. $return \; \sum_i this_i$

```
void Math::Vector::addSummedColumns(const Matrix<T>& matrix,
                  const T scale = 1.0) const
```

adds scaled column vectors of the input matrix to this vector, e.g. $this = this + \sum_i scale \times matrix.col_i$

`matrix`	the input matrix
`scale`	the scale factor. Default is 1.0

```
void Math::Vector::addSquaredSummedColumns(const Matrix<T>& matrix,
                  const T scale = 1.0) const
```

adds scaled squared (elementwise) column vectors of the input matrix to this vector, e.g. $this = this + \sum_i scale \times matrix.col_i \odot matrix.col_i$

`matrix`	the input matrix
`scale`	the scale factor. Default is 1.0

`void Math::Vector::addSummedRows(const Matrix<T>& matrix, const T scale = 1.0)`

adds scaled row vector of the input matrix to this vector, e.g. $this = this + \sum_i scale \times matrix.row_i$

`matrix`	the input matrix
`scale`	the scale factor. Default is 1.0

`void Math::Vector::getMaxOfColumns(const Matrix<T>& X)`

saves the maximum of each column vector of the input matrix in this vector, e.g. $this_i = max(X.col_i)$

`X`	the input matrix

`T normEuclidean() const`

returns the Euclidean norm of this vector, e.g. $return \sqrt{this^T this}$

`T chiSquareDistance(const Vector<T>& v) const`

returns the chi square distance of this vector with the input vector, e.g. $return \sum_i (this_i - v_i)^2/(this_i + v_i)$

`v`	the input vector.

3 The Features Module

3.1 Reading Data

3.1.1 Input Format

The framework supports six different types of feature inputs which are `vectors`, `sequences`, `images`, `videos`, `labels`, and `sequencelabels`. Each format has two header lines followed by the actual data. Details are described in the following.

Vectors

Indicated by `#vectors` in the first row and followed by `<num-vectors>` and `<feature-dimension>` in the second row. For example, a data file with five vectors each of dimension three could look as follows:

```
#vectors
5 3
1.0 1.2 -2.3
2.3 -1.5 -2.0
-0.5 0.5 0
1.0 1.5 1.0
2.0 2.3 -3.4
```

Sequences

Sequence file are similar to vector files but in addition to the total number of vectors and the feature dimension, the number of sequences in the file is also given as third element in the second row. The end of a sequence is indicated by `#`. A data file with two sequences containing two and three vectors could look as follows:

```
#sequences
5 3 2
1.0 1.2 -2.3
2.3 -1.5 -2.0
#
-0.5 0.5 0
1.0 1.5 1.0
2.0 2.3 -3.4
#
```

`5 3 2` indicates that there are in total 5 vectors, each of dimension 3, and there are 2 sequences.

Images

Images follow a similar structure. The first header row states `#images` followed by a second row specifying the image width and height as well as the number of channels (one or three). The data is the absolute path to the image. If images do not match the specified width and height, they are resized while reading. An image file could look as follows:

```
#images
400 300 3
/path/to/img1.jpg
/path/to/img2.jpg
/path/to/img3.jpg
/path/to/img4.jpg
```

While reading, the images would be resized to 400 pixels in width and 300 pixels in height and they would be treated as RGB images (3 channels).

Videos

Videos look like image files. Actually, a video in our framework is defined as a sequence of frames. Thus, it is basically an image file apart from the first row of the header. Similar to sequence files, the end of a video is indicated with `#`. A video file containing two videos could look as follows:

```
#videos
400 300 3
/path/to/video1/frame1.jpg
/path/to/video1/frame2.jpg
/path/to/video1/frame3.jpg
#
/path/to/video2/frame1.jpg
/path/to/video2/frame2.jpg
/path/to/video2/frame3.jpg
/path/to/video2/frame4.jpg
/path/to/video2/frame5.jpg
#
```

Labels

Particularly for classification tasks, it is convenient to provide a class label for each vector or sequence of an input file. Therefore, label files are also provided. The vector equivalent `#labels` requires the second row of the header to give the total number of labels and the number of classes as `<total-number-of-labels> <number-of-classes>`. Consider, for instance, a problem with three classes and five observations. We can define the labels as follows:

```
#labels
5 3
0
2
1
0
2
```

`5 3` indicates that there are 5 observations and 3 classes. Class labels always start at zero, so the possible labels for 3 classes are $0, \dots, 2$.

Sequencelabels

As sometimes a label for each frame of a video or a sequence is required, in addition to regular labels, sequencelabels can be defined. Similar to sequence files, the second row of the header is extended by the number of sequences and sequences are again separated by `#`. A sequence label file with two sequences containing two and three labels from a set of three classes could look as follows:

```
#sequencelabels
5 3 2
0
2
#
1
0
2
#
```

3.1.2 Features::FeatureReader

see Features/FeatureReader.hh

In order to read features in the framework, the `FeatureReader` and `AlignedFeatureReader` classes from the `Features` module can be used. While the first reads a features or label file alone, the second reads a source and a target file that need to match in their number of observations or sequences. Find function definitions and examples for different cases below.

FeatureReader

The `FeatureReader` class provides functions to read input files of type `#vectors` and `#images`. Files of the format `#sequences` and `#videos` are intepreted as `#vectors` and `#images`, respecively.

Parameters

- `features.feature-reader.feature-cache` (string) the input file
- `features.feature-reader.buffer-size` (u32) number of vectors/sequences to load into main memory at a time
- `features.feature-reader.shuffle-buffer` (bool) randomly shuffle the order of vectors/sequences
- `features.feature-reader.preprocessors` (list of strings) list of preprocessors to apply to the data, see below

Most Important Functions

`FeatureReader(const char* name = "features.feature-reader")`

constructor

`name`	the name to address the feature reader in the configuration. Default is `features.feature-reader`

`void Features::FeatureReader::initialize()`

initializes the FeatureReader. Needs to be called before using any other function.

`u32 Features::FeatureReader::totalNumberOfFeatures() const`

returns the number of feature vectors in the input file

`u32 Features::FeatureReader::featureDimension() const`

returns the dimension of the feature vectors or the number of classes for label files

`u32 Features::FeatureReader::newEpoch()`

starts a new epoch, i.e. resets the feature reader to its state after `initialized` as been called

`bool Features::FeatureReader::hasFeatures()`

returns true if there are unprocessed feature vectors

`const Math::Vector<Float>& next()`

returns a Math::Vector of floats containing the next-to-read feature vector

Configuration Example

```
[features.feature-reader]
feature-cache=<my-input-file.txt>    # input file
shuffle-buffer=true                  # shuffle the buffer
buffer-size=10                       # load at most 10 vectors/sequences
```

Code Example

```
#include <Features/FeatureReader.hh>

Features::FeatureReader reader;
reader.initialize();
// run twice over the data
for (u32 epoch = 0; epoch < 2; epoch++) {
    // read all feature vectors
    while (reader.hasFeatures()) {
        const Math::Vector<Float>& f = reader.next();
        // ... do something with f ...
    }
    reader.newEpoch();
}
```

SequenceFeatureReader

Mostly same function as the `FeatureReader` but this time, sequences are read instead of single vectors. The main differences to the `FeatureReader` are described below.

Most Important Functions

`u32 totalNumberOfSequences() const`
returns the number of sequences in the input file

`bool hasSequences() const`
similar to `hasFeatures()` but this time returns if there are unread sequences

`const Math::Matrix<Float>& next()`
returns the next sequence as a Matrix of floats where each column is one vector of the sequence (i.e. the matrix has size `featureDim` × `sequenceLength`)

Code Example

```
#include <Features/FeatureReader.hh>

Features::SequenceFeatureReader reader;
reader.initialize();
// read all feature sequences
while (reader.hasSequences()) {
    const Math::Matrix<Float>& matrix = reader.next();
    // ... do something with matrix ...
}
```

LabelReader

Same as the `FeatureReader` but reads `#labels` files. Get next label via

`u32 nextLabel()`
returns the next label in the input file

SequenceLabelReader

Same as the `SequenceFeatureReader` but reads `#sequencelabels` files. Get next sequence of labels via `const std::vector<u32>& nextLabelSequence()`

return an std::vector containing a label for each frame of the sequence

3.1.3 Features::AlignedFeatureReader

see Features/AlignedFeatureReader.hh

The `AlignedFeatureReader` aligns two feature files to each other. For classification and regression tasks, there is usually an input file providing the features and a target file providing the labels (for classification) or target features (for regression) that are used as ground truth. This class makes sure that both can be aligned to each other, i.e. they have the same amount of feature vectors and/or sequences. Particularly, it ensures that the correct targets are returned even if the source (the input) is shuffled.

Parameters

- `features.aligned-feature-reader.feature-cache` the source/input feature file
- `features.aligned-feature-reader.target-cache` the target feature/label file

Most methods are inherited from the `FeatureReader` and `SequenceFeatureReader`, respectively. In order to get to corresponding labels/targets, few additional methods are available.

AlignedFeatureReader

Used to align `#vectors` files with other `#vectors` files.

`u32 targetDimension() const`

feature dimension of the target vectors

`const Math::Vector<Float>& target() const`

return the target vector

Code Example

```
#include <Features/AlignedFeatureReader.hh>

Features::AlignedFeatureReader reader;
reader.initialize();
// read all feature vectors
while (reader.hasFeatures()) {
    const Math::Matrix<Float>& source = reader.next();
    const Math::Matrix<Flaot>& target = reader.target();
    std::cout << "source: " << source.toString() << std::endl;
    std::cout << "target: " << target.toString() << std::endl;
}
```

LabeledFeatureReader

Used to align `#vectors` files to `#labels` files.

`u32 label() const`
return the label for the last feature vector obtained with `next()`

`u32 nClasses() const`
returns the number of target classes

Code Example

```
#include <Features/AlignedFeatureReader.hh>

Features::LabeledFeatureReader reader;
reader.initialize();
// read all feature vectors
while (reader.hasFeatures()) {
    const Math::Matrix<Float>& source = reader.next();
    u32 label = reader.label();
    std::cout << "source: " << source.toString() << std::endl;
    std::cout << "label: " << label << std::endl;
}
```

AlignedSequenceFeatureReader

Used to align `#sequences` files to `#vectors` files. The number of sequences in the source file and the number of vectors in the target file must be the same. Methods to get the source features are inherited from `SequenceFeatureReader`, target features can be accessed similar to the `AlignedFeatureReader`.

LabeledSequenceFeatureReader

Used to align `#sequences` files to `#labels` files. The number of sequences in the source file and the number of labels in the target file must be the same. Methods to get the source features are inherited from `SequenceFeatureReader`, target labels can be accessed similar to the `LabeledFeatureReader`.

TemporallyAlignedSequenceFeatureReader

Used to align `#sequences` files to other `#sequences` files. The number of sequences in the source file and the number of sequences in the target file must be the same as well as the number of feature vectors for each source/target sequence pair. Methods to get the source features are inherited from `SequenceFeatureReader`, target sequences can be accessed with this method:

`const Math::Matrix<Float>& target() const`
return the target sequence aligned to the latest (via `next()`) obained source sequence as a matrix of size `targetDimension` $\times$ `numberOfFrames`

Code Example

```
#include <Features/AlignedFeatureReader.hh>
using namespace std;

Features::TemporallyAlignedFeatureReader reader;
reader.initialize();
// read all feature sequences
while (reader.hasSequences()) {
    const Math::Matrix<Float>& source = reader.next();
    const Math::Matrix<Float>& target = reader.target();
    cout << source.nColumns() << " = " << target.nColumns() << endl;
    cout << source.nRows() << " = " << reader.featureDimension() << endl;
    cout << target.nRows() << " = " << reader.targetDimension() << endl;
}
```

TemporallyLabeledSequenceFeatureReader

Used to align `#sequences` files to `#labelsequences` files. The number of sequences in the source file and the number of label sequences in the target file must be the same as well as the number of feature vectors/labels for each source/target sequence pair. Methods to get the source features are inherited from `SequenceFeatureReader`, target label sequences can be accessed with this method:

`const std::vector<u32>& labelSequence() const`
return the target label sequence aligned to the latest (via `next()`) obained source sequence as a std::vector with `numberOfFrames` label entries

Code Example

```
#include <Features/AlignedFeatureReader.hh>

Features::TemporallyLabeledFeatureReader reader;
reader.initialize();
// read all feature sequences
while (reader.hasSequences()) {
    const Math::Matrix<Float>& source = reader.next();
    const std::vector<u32>& labels = reader.labelSequence();
    std::cout << "source: " << source.toString() << std::endl;
    for (u32 t = 0; t < labels.size(); t++)
        std::cout << labels.at(t) << std::endl;
}
```

3.1.4 Features::Preprocessor

see Features/Preprocessor.hh

Input data can be preprocessed directly when being read. Therefore, a variety of feature preprocessors is available. To illustrate how to use those preprocessors, we start with a simple example of subtracting a vector (e.g. the mean) from all feature vectors and afterwards multiplying the result by a matrix (e.g. for a PCA).

Configuration Example

```
[features.feature-reader]
feature-cache = input.txt
preprocessors = my-preprocessor1, my-preprocessor2

[my-preprocessor1]
type          = vector-subtraction
vector        = mean.vector

[my-preprocessor2]
type          = matrix-multiplication
matrix        = pca.matrix
```

Note that `preprocessors` can be a comma separated list of arbitrary names that are subsequently defined with the `type` parameter of the `Features::Preprocessor` class. In the following, the most important preprocessors available in Squirrel are specified.

Vector Subtraction Preprocessor

Subtracts a given vector from each feature vector.
`<name>.type = vector-subtraction`

- `<name>.vector` (string) filename of the vector to subtract

Vector Division Preprocessor

Divides each feature vector elementwise by a given vector of the same size.
`<name>.type = vector-division`

- `<name>.vector` (string) filename of the vector to divide by (elementwise division)

Matrix Multiplication Preprocessor

Multiplies each feature vector with a given matrix.
`<name>.type = matrix-multiplication`

- `<name>.matrix` (string) filename of the matrix to multiply each feature vector with
- `<name>.transpose` (bool) flag if matrix should be transposed

Windowing Preprocessor

Applies a temporal windowing of each feature vector with the neighboring vectors. If the current feature vector is at time t, this preprocessor concatenates all feature vectors in the range $t-$`window-size`/2 to $t+$ `window-size`/2. Requires the input features to be of type `#sequences` of `#videos`.
`<name>.type = windowing`

- `<name>.window-size` (u32) size of the temporal window

Z-Score Preprocessor

Applies mean and variance normalization on each feature vector. Mean and variance are computed on sequence level. Thus, input feature file is required to be of type `#sequence` of `#videos`.
`<name>.type = z-score`

L2-Normalization Preprocessor

Normalizes each feature vector by its ℓ_2 norma.
`<name>.type = l2-normalization`

Power-Normalization Preprocessor

Applies power normalization to each feature vector.
`<name>.type = power-normalization`

- `<name>.power` (Float) the exponent used for power normalization. Default is 0.5.

Random-Image-Cropping Preprocessor

Generates random crop out of an input image.Input feature file should be of type `#images` or `#videos`.
`<name>.type = random-image-cropping`

- `<name>.input-width` (u32) width of the input image
- `<name>.input-height` (u32) heigth of the input image
- `<name>.channels` (u32) number of channels of the input image
- `<name>.possible-side-lengths` (list of u32) two lengths from this list are randomly chosen to determine the crop size in x and y direction
- `<name>.crop-width` (u32) width to resize the randomly cropped image to
- `<name>.crop-height` (u32) height to resize the randomly cropped image to

Implementing Your Own Preprocessor

In order to implement your own preprocessor, inherit from the `Features::Preprocessor` class. Add the name of your preprocessor to the `enum Type` in `Features/Preprocessor.hh` and also to `Features::Preprocessor::paramType_` in `Features::Preprocessor.cc`. Note that the order of preprocessors in the `enum` and `paramType_` has to be the same. Then, add your preprocessor to the factory method `Features::Preprocessor::createPreprocessor` in the same `.cc` file.

When writing your own preprocessor class, make sure to override the following functions from the base class:

- `virtual void initialize(u32 inputDimension)` the initialize function to set up the preprocessor
- `virtual bool needsContext()` a method returning if the preprocessor operates on single feature vectors (like `vector-subtraction`) or on sequences (like `windowing`)
- `virtual void work(const Math::Matrix<Float>& in, Math::Matrix<Float>& out)` the actual implementation of the preprocessor

Note that the `work` method receives a `Math::Matrix` as input. In case of feature vectors (`#vectors` or `#images`), the matrix will be of size `input-dimension` $\times$ 1, in case of sequences, it will be of size `input-dimension` $\times$ `sequence-length`. The output sequence length has to be the same as the input sequence length, i.e. `out.nColumns()` must be equal to `in.nColumns()`. The feature dimension may vary, i.e. `out.nRows()` can be different from `in.nRows()` (which is the case e.g. for the `windowing` preprocessor). Make sure to set the variable `outputDimension_` to the correct value in the `initialize` method.

4 The Neural Network Module

Documentation will be added soon... please check again in a few days.