2017/07/21
© Kazumasa Horie

Static library usage

B1 Introduction

This document explains the details of LIBSDNN.lib, the static library for Visual Studio. LIBSDNN.lib is written in C++ and OpenMP using Visual Studio 2017 and is used to create other applications (mex files and command line tools).

B2 How to use the static library in your project

First, save bin\LIBSDNN.lib and its header file, bin\LIBSDNN.h, to your project folder. LIBSDNN.lib can be used as a general static library. For instructions on using static libraries, please refer to the MSDN tutorials (Microsoft, 2017).

The source files of LIBSDNN.lib are located in the LIBSDNN/source folder. The library source files can be compiled using other compilers such as GCC, but the performance of the library is not guaranteed in such cases. We recommend using the library included in the Microsoft Visual C++ compiler.

B2-1 General flow of library use

LIBSDNN.lib provides the class libsdnn::SDNN, which comprises the SDNN itself and contains all functions for training and testing the SDNN. Generally, the library is used in the following steps:

1. Include LIBSDNN.h and load LIBSDNN.lib.
2. Instantiate the libsdnn::SDNN class.
3. Initialize the SDNN using the InitSDNN function.
4. Prepare the training samples.
5. Train the SDNN using the Train function.
6. Estimate/approximate the samples with SDNN using the Estimate function.

The details of the member functions of the libsdnn::SDNN class and usage examples are shown in Chapters B3 and B4, respectively.

B3 Details of the member functions of the libsdnn::SDNN class

The libsdnn::SDNN class has the following member functions:

B3-1 InitSDNN

```
void InitSDNN(
        const std::string &parameter_filename)
;
```

This initializes the SDNN according to a parameter file.
Arguments:

- parameter_filename (const std::string &)

  The name of the parameter file (see also **How to describe a parameter file**).

Returns:

  void

Error:

  Error Number 65797 is returned when the SDNN is initialized twice.

B3-2 Reset

```
void Reset(void);
```

This resets the SDNN to the state prior to initialization. Please re-initialize using the InitSDNN function after resetting.

Arguments:

`void`

Returns:

`void`

Error:

None

B3-3 TrainOneSample

```
void TrainOneSample(
            const std::vector<double> &input,
            const double target
);
```

This trains the SDNN once using one training sample.

Arguments:

- `input (const std::vector<double> &)`
  The input vector of the training sample. Note that numerical inputs must be normalized in the range [0, 1] and symbolic inputs must be specified using integers starting from 0.
- `target (const double)`
  The target value (tutor signal) of the training sample. Note that target values must be specified as integers starting from 0 when applying pattern recognition issues.

Returns:

`void`

Error:

Error Number 65798 is returned when the SDNN is not initialized.

B3-4 Train

```
void Train(
            const std::vector<std::vector<double>>&input,
            const std::vector<double> target
            const std::string &completion_condition
);
```

This trains the SDNN with samples until the `completion_condition`, which is given as a string argument, is met.

Arguments:

- `input (const std::vector<std::vector<double>> &)`
  The list of input vectors of the training samples. Note that numerical inputs must be normalized in the range of [0, 1] and symbolic inputs must be specified using integers starting from 0.
- `target (const std::vector<double>)`
  The list of training sample target values (tutor signals). Note that the number and ordering of the elements must be the same as in the input vector list and that the target values must be specified as integers starting from 0 when applying pattern recognition issues.
- `completion_condition (const std::string &)`
  The training-completion condition, which is represented in the form of `std::string`, can be set using one of the following strings:
  - iteration(*n*)
    This tells the SDNN to repeat training procedure *n* times.
  - rmse(*p*,*m*)
    This tells the SDNN to repeat the training process until the root-mean-square error is less than *p*. If the SDNN cannot satisfy the condition after *m* iterations, the training

process terminates. This completion condition can only be applied to function approximation issues.

- `accuracy(a,m)`
  This tells the SDNN to repeat the training process until the classification accuracy becomes greater than *a*. If the SDNN cannot satisfy this condition after *m* iterations, the training process terminates. This completion condition can only be applied to pattern recognition issues.

Returns:
`void`
Errors:
- Error Number 65798 is returned if the SDNN is not initialized.
- Error Number 65792 is returned if the number of the elements in the target value list is not the same as in the input vector list.

B3-5 Estimate

```
double Estimate(
        const std::vector<double> &input
);
```

This calculates the recognizing/estimating result for one sample. Note that this function does not check the number of input vector elements.
Arguments:
- `input (const std::vector<std::vector<double>> &)`
  The input vector of the recognizing/estimating sample. Note that numerical inputs must be normalized in the range of [0, 1] and symbolic inputs must be specified using integers starting from 0.

Returns:
`double`
Note that the recognized results are described using integers starting from 0 when pattern recognition issues are applied.
Error:
Nothing

B3-6 Save

```
void Save(
        const std::string &filename
);
```

This saves the trained model (the parameters and synaptic weights of the SDNN) as an SDNN-model file that can be loaded into other projects using the Load function (see also Chapter B3-7).
Arguments:
- `filename (const std::string &)`
  The name of the SDNN-model file to be saved. We recommend to using .bin as the file extension.

Returns:
`void`
Error:
Error Number 65799 is returned if the SDNN is saved before initialization.

B3-7 Load

```
void Load(
        const std::string &filename
);
```

This loads the SDNN-model file and sets the parameters and synaptic weights according to the SDNN-model file. This function should be used prior to initialization.

Arguments:

- filename (const std::string &)
  The name of the SDNN-model file to be loaded. SDNN-model files created using mex or command line tools can be used.

Returns:

  void

Error:

  Error Number 65800 is returned if the SDNN has already been initialized.

B4 Example

Here, we demonstrate an example of static library usage by replicating Nonaka's study (Nonaka et al. 2011), in which the following two-variable function with some non-uniformity was approximated:

$$\mathbf{f}(x,y) = \begin{cases} \mathbf{1}\ ((x-0.5)^2+(y-0.5)^2 \le 0.04) \\ \dfrac{1+x}{2}\sin^2\left(6\pi\sqrt{x}y^2\right)(\text{otherwise}) \end{cases} \quad (x,y \in [0,1]) \qquad \textbf{(B1)}$$

This function is defined as follows using Nonaka_Function(*input_vector)* within the source file for this example, doc\nonaka.cpp, which is located in the Static Library:

```
double Nonaka_Function(std::vector<double> &input)
{
        if (input.size() != 2)
                return 0;
          return (pow(input[0] - 0.5, 2) + pow(input[1] - 0.5, 2) <= 0.04)?1:
          (1 + input[0]) / 2 * pow(sin(6 * 3.141592*sqrt(input[0])*pow(input[1], 2)), 2);
}
```

B4-1 Instantiating and Initializing

The libsdnn::SDNN class must be instantiated and then initialized through the InitSDNN function. The following example demonstrates the instantiating and initializing of libsdnn::SDNN:

```
libsdnn::SDNN sdnn_nonaka;              // Instantiating
sdnn_nonaka.InitSDNN("TestNNK.txt");         // Initializing
```

The function requires a parameter file; please refer to the document **How to describe a parameter file** for further information

B4-2 Preparing the training sample set

After instantiation and initializing, the training sample set is prepared. The training sample set comprises two lists: the input list (an array containing the input vectors), and the target list (an array of target values). The number of elements and their ordering must be the same in the input and target lists.

In the following example, 2,000 samples are randomly chosen from the 1,001*1,001 lattice points within the defined domain:

```
// instantiating training sample set
std::vector<std::vector<double>> input_list;
```

```
std::vector<double> target_list;

// initialize pseudorandom number generator
std::vector<unsigned int> random_seed{2466971321,687495437};
std::seed_seq random_seed_seq(random_seed.begin(), random_seed.end());
std::mt19937 mt;
mt.seed(random_seed_seq);
std::uniform_int_distribution<int> kernel(0, 1000);

for (int i = 0; i < 2000; i++)
{
	std::vector<double> sample_buffer;
	sample_buffer.push_back((double)kernel(mt) / 1000);
	sample_buffer.push_back((double)kernel(mt) / 1000);

	target_list.push_back(Nonaka_Function(sample_buffer));
	input_list.push_back(sample_buffer);
}
```

B4-3 Training the SDNN

After preparing a training sample, the SDNN can be trained to approximate a function using the `Train` function (see Chapter B3-4 for details). The `Save` function is useful for saving the trained model (parameters and synaptic weights of the SDNN), which can be reused in other projects by loading the SDNN-model file.

`Train` and `Save` are implemented as follows:

```
sdnn_nonaka.Train(input_list,target_list, “iteration(300)”);
sdnn_nonaka.Save("save.bin");
```

B4-4 Calculating the approximation result using the trained SDNN

The approximation result is calculated using the `Estimate` function. In the following example, the SDNN estimates the output of the target function at the remaining lattice points (1,001 × 1,001 – 2,000):

```
std::vector<double> input = { 0,0 };
double result;
double rmse = 0;
for (int i = 0; i < 1001; i++)
{
	for (int j = 0; j < 1001; j++)
	{
		input[0] = 1.0 / 1000 * i;
		input[1] = 1.0 / 1000 * j;
		result = sdnn_nonaka.Estimate(input);
		rmse += pow(result - Nonaka_Function(input), 2);
	}
}
for (int k = 0; k < 2000; k++)
{
		result = sdnn_nonaka.Estimate(input_list[k]);
		rmse -= pow(result - Nonaka_Function(input), 2);
}

rmse /= (1002001 - 2000);
rmse = sqrt(rmse);
std::cout << "rmse:" << rmse << std::endl;
```

References

Microsoft Developer Network, "Walkthrough: Creating and Using a Static Library (C++)," https://msdn.microsoft.com/en-us/library/ms235627.aspx (accessed online 2017/12/9)

K. Nonaka, F. Tanaka, and M. Morita. Empirical comparison of feedforward neural network on two-variable function approximation (in Japanese). *IEICE TRANSACTIONS on Information and Systems* (in Japanese), J94 (12): 2114-2125, 2011.