Deep Neural Nets Documentation

Release 0.9.1

Liangfu Chen

Apr 20, 2018

Contents

Contents:

CHAPTER 1

dnn: A light-weight yet efficient framework for deep learning

1.1 Overview

The Deep Neural Nets (DNN) library is a deep learning framework designed to be small in size, computationally efficient and portable.

We started the project as a fork of the popular OpenCV library, while removing some components that is not tightly related to the deep learning framework. Comparing to Caffe and many other implements, DNN is relatively independent to third-party libraries, (Yes, we don't require Boost and Database systems to be install before crafting your own network models) and it can be more easily portable to mobile systems, like iOS, Android and RaspberryPi etc.

1.2 Available Modules

The following features have been implemented:

- Mini-batch based learning, with OpenMP support
- YAML based network definition
- Gradient checking for all implemented layers

The following modules are implemented in current version:

Module Name	Description
InputLayer	Data Container Layer, for storing original input images
ConvolutionLayer	Convolutional Neural Network Layer, performs 2d convolution upon images
MaxPoolingLayer	Sub-Sampling Layer, performs max-pooling operation
DenseLayer	Fully Connected Layer, with activation options, e.g. tanh, sigmoid, softmax, relu etc.
SimpleRNNLayer	vallina Recurrent Neural Network (RNN) Layer, for processing sequence data
MergeLayer	Merge Layer, for combining output results from multiple different layers

More modules will be available online !

1.2.1 Network Definition

Layer Type	Attributes
Input	*name*, *n_input_planes*, *input_height*, *input_width*, *seq_length*
Convolution	*name*, *visualize*, *n_output_planes*, *ksize*
MaxPooling	*name*, *visualize*, *ksize*
Dense	*name*, *input_layer(optional)*, *visualize*, *n_output_planes*, *activation_type*
SimpleRNN	*name*, *n_output_planes*, *seq_length*, *time_index*, *activation_type*
Merge	*name*, *input_layers*, *visualize*, *n_output_planes*

With the above parameters given in YAML format, one can simply define a network. For instance, a modifed lenet can be:

```
%YAML:1.0
layers:
 - {type: Input, name: input1, n_input_planes: 1, input_height: 28, input_width: 28,␣
↪seq_length: 1}
 - {type: Convolution, name: conv1, visualize: 0, n_output_planes: 6, ksize: 5,␣
↪stride: 1}
 - {type: MaxPooling, name: pool1, visualize: 0, ksize: 2, stride: 2}
 - {type: Convolution, name: conv2, visualize: 0, n_output_planes: 16, ksize: 5,␣
↪stride: 1}
 - {type: MaxPooling, name: pool2, visualize: 0, ksize: 2, stride: 2}
 - {type: Dense, name: fc1, visualize: 0, n_output_planes: 10, activation_type: tanh}
```

Then, by ruuning network training program:

```
$ network train --solver data/mnist/lenet_solver.xml
```

one can start to train a simple network right away. And this is the way the source code and data models are tested in Travis-Ci. (See .travis.yml in the root directory)

1.3 Installation

CMake is required for successfully compiling the project.

Under root directory of the project:

```
$ cd $DNN_ROOT
$ mkdir build
$ cmake ..
$ make -j4
```

1.4 License

MIT