C++ neural network library

517 commits 3 branches 0 releases 14 contributors

Branch: master New pull request New file Find file HTTPS https://github.com/yoavg/cnn. Download ZIP

This branch is 39 commits behind clab:master. Pull request Compare

yoavg pass cmdline args to pycnn init Latest commit 6e49f0c 27 days ago

cmake	test stub	4 months ago
cnn	Merge branch 'master' of github.com:clab/cnn	28 days ago
examples	bad cmake dependencies	a month ago
external	skip-gram RNN language model	7 months ago
pycnn	pass cmdline args to pycnn init	27 days ago
pyexamples	fixed embarassing mistake in RNNs.ipynb	4 months ago
tests	fix test failure	a month ago
.gitignore	rename HSM CFSM	2 months ago
.gitmodules	skip-gram RNN language model	7 months ago
.travis.yml	CI gcc 4.8	3 months ago
CMakeLists.txt	Merge branch 'master' of github.com:clab/cnn	a month ago
INSTALL.md	INSTAll.md	5 months ago
LICENSE	lic terms	7 months ago
README.md	README fix to use Expression notation and fix bugs	3 months ago
TODO.cnn	Update TODO.cnn	3 months ago
config.h.cmake	multibackend support, eigen cleanup and optimizations	9 months ago

README.md

cnn

C++ neural network library

Getting started

You need the development version of the Eigen library for this software to function properly. If you use the current stable release, you will get an error like the following:

```
Assertion failed: (false && "heap allocation is forbidden (EIGEN_NO_MALLOC is defined)"), function check
```

Building

First you need to fetch the dependent libraries

```
git submodule init
git submodule update
```

In `src`, you need to first use `cmake` to generate the makefiles

```
mkdir build
cd build
cmake .. -DEIGEN3_INCLUDE_DIR=/path/to/eigen
```

Then to compile, run

```
make -j 2
```

To see that things have built properly, you can run

```
./examples/xor
```

which will train a multilayer perceptron to predict the xor function.

Building without Eigen installed

If you don't have Eigen installed, the instructions below will fetch and compile both `Eigen` and `cnn`.

```
git clone https://github.com/clab/cnn.git
hg clone https://bitbucket.org/eigen/eigen/

cd cnn/
mkdir build
cd build
cmake .. -DEIGEN3_INCLUDE_DIR=../eigen
make -j 2
```

Debugging

If you want to see the compile commands that are used, you can run

```
make VERBOSE=1
```

Training Models

An illustation of how models are trained (for a simple logistic regression model) is below:

```
// *** First, we set up the structure of the model
// Create a model, and an SGD trainer to update its parameters.
Model mod;
SimpleSGDTrainer sgd(&mod);
// Create a "computation graph," which will define the flow of information.
ComputationGraph cg;
// Initialize a 1x3 parameter vector, and add the parameters to be part of the
// computation graph.
Expression W = parameter(cg, mod.add_parameters({1, 3}));
// Create variables defining the input and output of the regression, and load them
// into the computation graph. Note that we don't need to set concrete values yet.
vector<cnn::real> x_values(3);
Expression x = input(cg, {3}, &x_values);
cnn::real y_value;
Expression y = input(cg, &y_value);
// Next, set up the structure to multiply the input by the weight vector,  then run
// the output of this through a logistic sigmoid function (logistic regression).
Expression y_pred = logistic(W*x);
// Finally, we create a function to calculate the loss. The model will be optimized
```

```
// to minimize the value of the final function in the computation graph.
Expression l = binary_log_loss(y_pred, y);
// We are now done setting up the graph, and we can print out its structure:
cg.PrintGraphviz();

// *** Now, we perform a parameter update for a single example.
// Set the input/output to the values specified by the training data:
x_values = {0.5, 0.3, 0.7};
y_value = 1.0;
// "forward" propagates values forward through the computation graph, and returns
// the loss.
cnn::real loss = as_scalar(cg.forward());
// "backward" performs back-propagation, and accumulates the gradients of the
// parameters within the "Model" data structure.
cg.backward();
// "sgd.update" updates parameters of the model that was passed to its constructor.
// Here 1.0 is the scaling factor that allows us to control the size of the update.
sgd.update(1.0);
```

Note that this very simple example that doesn't cover things like memory initialization, reading/writing models, recurrent/LSTM networks, or adding biases to functions. The best way to get an idea of how to use cnn for real is to look in the `example` directory, particularly starting with the simplest `xor` example.