Multi-layer perceptrons using RPROP

nn is a lightweight neural network library using resilient propagation for adapting the weights

Installation

nn was tested on Ubuntu, Arch Linux and MacOS

- install CMake, Eigen3 and subversion. On Ubuntu this is done as follows:

```
sudo apt-get install cmake subversion libeigen3-dev
```

- clone the **nn** repository or download it here
- change to the **nn** directory and create a build folder

```
cd path/to/nn
mkdir build
```

- run cmake from within the build folder and compile the library using make

```
cd build
cmake ..
make
```

- run the example code

```
./tutorial
```

- to compile unit tests for **nn**, run cmake with the option `-DWITH_GTEST=ON`

```
cmake .. -DWITH_GTEST=ON
make
./nntest
```

License

Usage of the library

The source code for this tutorial can be found in `tutorial.cpp`.

Preparing your data

Organize your training data into a *(m x n_input)* matrix containing the training inputs. Each row of this matrix corresponds to a training sample and each column to a feature. Prepare a matrix of size *(m x n_output)* containing the target values, where *n_output* is the number of dimensions of the output.

```
matrix_t X(m, n_input);
matrix_t Y(m, n_output);

// fill here with data
```

Initializing the neural network

This neural network implementation only supports fully connected feed forward multi-layer perceptrons (MLPs) with sigmoidal activation functions. The neurons are organized into *k* layers. There is at least one input layer, one output layer and an arbitrary number of hidden layers. Each neuron has outgoing connections to all neurons in the subsequent layer. The number of neurons in the input and the output layer is given by the dimensionality of the training data. After specifying the network topology you can create the `NeuralNet` object. The weights will be initialized randomly.

```
Eigen::VectorXi topo(k);
topo << n_input, n1, n2, ..., n_output;

// initialize a neural network with given topology
NeuralNet nn(topo);
```

Scaling the data

When working with MLPs you should always scale your data, such that all the features are in the same range and the output values are between 0 and 1. You can do this by passing your training data to the `autoscale` function, which computes the optimal mapping. After calling `autoscale` this mapping will be performed automatically, so you only have to do this once. To reset the scaling parameters to standard values call `autoscale_reset`.

```
nn.autoscale(X,Y);
```

Training the network

Alternate between computing the quadratic loss of the MLP and adapting the parameters until the loss converges. You can also specify a regularization parameter *lambda*, which punishes large weights and thereby avoids overfitting.

```
for (int i = 0; i < max_steps; ++i) {
  err = nn.loss(X, Y, lambda);
  nn.rprop();
}
```

Making predictions

If you trained a model, you can make predictions on new data by passing it through the network and observe the activation on the output layer.

```
nn.forward_pass(X_test);
matrix_t Y_test = nn.get_activation();
```

Reading and writing models to disk

You can read and write MLPs to binary files.

```
// write model to disk
nn.write(filename);

// read model from disk
NeuralNet nn(filename);
```

Changing the floating number precision

nn uses double precision floats by default. You can change this behaviour in the file `nn.h`.

```
#define F_TYPE double
```

MNIST dataset

In order to test **nn** on the MNIST dataset, download the dataset from here and run the `mnist` tool.

```
./mnist path/to/data
```

The tool will train a MLP with two hidden layers, containing 300 and 100 neurons respectively connected by 266.610 weights. Using this setup error rates below 5% are accomplished on the test dataset.

Using nn in your own project

Just copy `nn.h` and `nn.cpp` into your workspace, make sure that the `Eigen` headers are found and start coding!