cppNet

cppNet is small and easy to use library for neural networks. It is based on Eigen, C++ template library for linear algebra.

The library supports:

- Fully connected layers
- Convolution layers
- Max pooling layers
- Dropout
- SGD - stochastic gradient decent
- Loading and storing trained network
- Live graphs (Python)

Installation

Here are the steps to install **cppNet**.

1. First thing you need are the repository files, get them by typing `git clone https://github.com/soCzech/cppNet.git` into terminal.
2. Run the `download.sh` script. It downloads MNIST dataset (used in the example files) and Eigen library which is needed for the vector operations.
3. You are ready to rock. You can try it by typing `make [all]` and running `fc` or `cn` demo (see example usage).

Example usage

In `test` directory the repository contains two simple networks `fully_connected_test.cpp` and `convolutional_test.cpp`. Both are to be run independently of each other.

Both can be compiled by running `make [all]`. It creates executable files `fc` for the fully connected demo and `cn` for the convolutional. Running the files will start the training. To see progress, start a HTTP server by `python server.py --logdir=../bin/logs/(fc|cn)` while in the `dashboard` directory. Afterwards, you can go to `127.0.0.1:8080` in your browser.

After each run, the network values are saved to checkpoint file `fc.ckpt` or `cn.ckpt` in `bin/models`. When the network is run, it tries to load those values. So in order to improve results you can run the network a few times.

In order to run those networks you need the MNIST dataset located in `dataset/mnist` and a script `test/mnist.cpp` that loads it. When creating your own networks over other datasets, all you need are the files in `core` and in `layers`. But you need to fill the `data_vector` object with training data yourself.

Network layers

Building your own network requires layers, here are the ones we support and how to use them.

- `fully_connected_layer(input, output, fn, fn_prime)`
  The most basic layer, works by connecting each neuron from the previous layer to all neurons in this layer. Each connection results in one unique weight, that means `input*output` values represented as a matrix internally.

  Arguments:
  - `input`: number of neurons in the previous layer
  - `output`: number of neurons on the output of this layer
  - `fn`: function which should by applied to the weights on output, the functions are specified in `core.hpp` and for a detailed explanation, see next chapters
  - `fn_prime`: prime function to the one above, used for backpropagation

- `convolutional_layer(input, kernel, stride, channels, ...)`
  2D convolutional network has a 2D kernel – small fully connected layer with dimensions `kernel_size*1`. It then slides over the output of the previous layer and for each its neuron computes its value using the same kernel.

  Arguments:
  - `input` as `{rows, columns, channels}`: the numbers usually represents the actual meaning of the data;

    a color image would be represented as $\{height, width, 3\}$ where the 3 represents RGB channels, deeper in the network we may need more channels that represent some structural data like if the image contains lines or circles;

    `rows*columns*channels` has to equal to the size of the output of the previous layer
  - `kernel` as `{rows, columns}`: size of the kernel
  - `stride` as `{rows, columns}`: by how many rows/columns to slide each time the kernel is applied; if not desired, use {1,1}
  - `channels`: number of output channels
  - `fn` and `fn_prime` same as in the fully connected layer

- `max_pooling_layer(input, pool_size)`
  This layer outputs only the maximal value in each pool of the previous layer's output resulting in smaller size.

  Arguments:
  - `input` as `{rows, columns, channels}` same as in the convolutional layer
  - `pool_size` as `{rows, columns}`: size of the pool from which to chose the maximum

- `dropout_layer(input, probability)`
  The layer sets each neuron's value to 0 with probability from the argument. This should prevent overfitting the network on the training data because it needs to learn to distinguish objects even if complete information is not available.

  Arguments:
  - `input`: number of neurons in the previous layer
  - `probability`: the probability a neuron value is set to zero

Cost layers

To transform network results to meaningful numbers and to ensure the backpropagation algorithm works as desired the cost layer needs to be the last layer of your network. We support two of those.

- `cross_entropy_cost_layer(fn)`
  The layer computes the loss as $y \log a + (1 - y) \log(1 - a)$ where y is the correct vector from a dataset and a is the output of the network.

  Arguments:
  - `fn`: function which should by applied to the weights on output, see that the derivative does not depend on a prime function of the output

- `quadratic_cost_layer(fn, fn_prime)`
  The basic cost layer, computes the loss as $\frac{1}{2} |a - y|^2$ where y is the correct vector from a dataset and a is the output of the network.

  Arguments:
  - `fn`: function which should by applied to the weights on output
  - `fn_prime`: prime function to the one above, used for backpropagation

Neuron functions

For the network to compute something, you need to apply a function to at least some neurons. Each functions takes one argument, a 1D vector `const VectorF&`, and returns `VectorF`. Here are the functions we support all of which are defined in `core.hpp`. All of the functions have their derivatives with the same name just ending with `prime_function`.

- `sigmoid_function(v)`
  The function returns $\frac{1}{1+e^{-v}}$. That means the input value is normalized to interval [0,1]. Note the closed interval because of the rounding errors.

- `relu_function(v)`
  Easy to compute function $max(0,v)$ that is quite useful because its derivative does not converges to 0 as the argument of the function gets further from 0.

- `identity_function(v)`
  In the case you do not want to apply any function, e.g. in the last layer, use this one. Do not forget you need to provide the prime function `identity_prime_function` when needed.

- `softmax_function`
  The usual cost layer function $\frac{e^x}{\sum_i e^{x_i}}$ returning probability distribution. Implemented as numerically stable.

Creating a network

To create and train network you need the `cppNet::network` class. For example usage, see files in `test` directory. Here are all the methods you will need.

- `void add_layer(layer_ptr)`
  The function takes a unique pointer to a layer which it adds to the network.

- `bool load(filename)`
  The function tries to load the weights of the network from the file `filename`. Returns `true` if successful, `false` otherwise.

- `void save(filename)`
  Saves the network to the file `filename`. Be sure the directory specified in `filename` exists. If not, the network will not be saved.

- `layer* operator[](i)`
  Returns a pointer to the `i`-th layer in the network starting from 0.

- `void SGD(training_data, epochs, batch_size, learning_rate, l2_lambda, test_data, summary_dir)`

  The core of the network training. It takes the `training_data`; shuffles them; splits them to batches; feeds them to the network; and via backpropagation computes the derivatives. After each batch, it updates weights of the network based on the cumulative derivatives of the batch.

  When finished with all the `training_data`, it evaluates the network on the `test_data`. This whole process is repeated `epochs`-times.

  Arguments:
  - `training_data`: the `std::vector` filled with `training_data` objects specified in `core.hpp`
  - `epochs`: the number of times to repeat the process described above
  - `batch_size`: the number of training examples to be used before the weights are updated
  - `learning_rate`: how fast to learn; setting it to too big values can overshoot the minimum we are looking for; with too small values, it takes longer to learn

  - `l2_lambda`: parameter of the L2 regularization, from interval [0,1); bigger values force the network to use only small weights; 0 means no L2 regularization.
  - `test_data`: the same as `training_data`, but used for network evaluation instead
  - `summary_dir`: where to store the log files; how to access the graphs of the training, see example usage

- `int evaluate_network(test_data)`
  Returns number of correct prediction from all the `test_data`. The format of the argument is the same as in `SGD` function.

- `VectorF feedforward(input)`
  Returns raw output of the network given the `input` vector.