tiny-dnn Documentation

Release 1.0.0a1

Taiga Nomi

Jul 27, 2017

Contents

tiny-dnn is a header only, dependency free deep learning library written in C++. It is designed to be used in the real applications, including IoT devices and embedded systems.

CHAPTER 1

Table of Contents

Getting Started:

A quick introduction to tiny-dnn

Include tiny_dnn.h:

```
#include "tiny_dnn/tiny_dnn.h"
using namespace tiny_dnn;
using namespace tiny_dnn::layers;
using namespace tiny_dnn::activation;
```

Declare the model as `network`. There are 2 types of network: `network<sequential>` and `network<graph>`. The sequential model is easier to construct.

```
network<sequential> net;
```

Stack layers:

```
net << conv(32, 32, 5, 1, 6, padding::same) << tanh()  // in:32x32x1, 5x5conv,␣
↪6fmaps
    << max_pool(32, 32, 6, 2) << tanh()                // in:32x32x6, 2x2pooling
    << conv(16, 16, 5, 6, 16, padding::same) << tanh() // in:16x16x6, 5x5conv,␣
↪16fmaps
    << max_pool(16, 16, 16, 2) << tanh()               // in:16x16x16, 2x2pooling
    << fc(8*8*16, 100) << tanh()                       // in:8x8x16, out:100
    << fc(100, 10) << softmax();                       // in:100 out:10
```

Declare the optimizer:

```
adagrad opt;
```

In addition to gradient descent, you can use modern optimizers such as adagrad, adadelta, adam.

Now you can start the training:

```
int epochs = 50;
int batch = 20;
net.fit<cross_entropy>(opt, x_data, y_data, batch, epochs);
```

If you don't have the target vector but have the class-id, you can alternatively use `train`.

```
net.train<cross_entropy, adagrad>(opt, x_data, y_label, batch, epochs);
```

Validate the training result:

```
auto test_result = net.test(x_data, y_label);
auto loss = net.get_loss<cross_entropy>(x_data, y_data);
```

Generate prediction on the new data:

```
auto y_vector = net.predict(x_data);
auto y_label = net.predict_max_label(x_data);
```

Save the trained parameter and models:

```
net.save("my-network");
```

For a more in-depth about tiny-dnn, check out MNIST classification where you can see the end-to-end example. You will find tiny-dnn's API in How-to.

How-Tos:

How-Tos

Details about tiny-dnn's API and short examples.

construct the network model

There are two types of network model available: sequential and graph. A graph representation describe network as computational graph - each node of graph is layer, and each directed edge holds tensor and its gradients. Sequential representation describe network as linked list - each layer has at most one predecessor and one successor layer. Two types of network is represented as network and network class. These two classes have same API, except for its construction.

sequential model

You can construct networks by chaining `operator <<` from top(input) to bottom(output).

```
// input: 32x32x1 (1024 dimensions)  output: 10
network<sequential> net;
net << convolutional_layer(32, 32, 5, 1, 6) << tanh() // 32x32in, conv5x5
    << average_pooling_layer(28, 28, 6, 2) << tanh()  // 28x28in, pool2x2
    << fully_connected_layer(14 * 14 * 6, 120) << tanh()
    << fully_connected_layer(120, 10);
```

```
// input: 32x32x3 (3072 dimensions)  output: 40
network<sequential> net;
net << convolutional_layer(32, 32, 5, 3, 9) << relu()
    << average_pooling_layer(28, 28, 9, 2) << relu()
    << fully_connected_layer(14 * 14 * 9, 120) << tanh()
    << fully_connected_layer(120, 40) << softmax();
```

If you feel these syntax a bit redundant, you can also use "shortcut" names defined in tiny_dnn.h.

```
using namespace tiny_dnn::layers;
net << conv(32, 32, 5, 3, 9) << relu()
    << ave_pool(28, 28, 9, 2) << relu()
    << fc(14 * 14 * 9, 120) << tanh()
    << fc(120, 40) << softmax();
```

If your network is simple mlp(multi-layer perceptron), you can also use `make_mlp` function.

```
auto mynet = make_mlp<tanh>({ 32 * 32, 300, 10 });
```

It is equivalent to:

```
network<sequential> mynet;
mynet << fully_connected_layer(32 * 32, 300) << tanh()
      << fully_connected_layer(300, 10) << tanh();
```

graph model

To construct network which has branch/merge in their model, you can use `network<graph>` class. In graph model, you should declare each "node" (layer) at first, and then connect them by `operator <<`. If two or more nodes are fed into 1 node, `operator,` can be used for this purpose. After connecting all layers, call `construct_graph` function to register node connections to graph-network.

```
// declare nodes
layers::input in1(shape3d(3, 1, 1));
layers::input in2(shape3d(3, 1, 1));
layers::add added(2, 3);
layers::fc out(3, 2);
activation::relu r();

// connect
(in1, in2) << added;
added << out << r;

// register to graph
network<graph> net;
construct_graph(net, { &in1, &in2 }, { &out });
```

train the model

regression

Use `network::fit` function to train. Specify loss function by template parameter (`mse`, `cross_entropy`, `cross_entropy_multiclass` are available), and fed optimizing algorithm into first argument.

```
network<sequential> net;
adagrad opt;
net << layers::fc(2, 3) << activation::tanh()
    << layers::fc(3, 1) << activation::softmax();

// 2training data, each data type is 2-dimensional array
std::vector<vec_t> input_data  { { 1, 0 }, { 0, 2 } };
std::vector<vec_t> desired_out {    { 2 },    { 1 } };
size_t batch_size = 1;
size_t epochs = 30;

net.fit<mse>(opt, input_data, desired_out, batch_size, epochs);
```

If you want to do something for each epoch / minibatch (profiling, evaluating accuracy, saving networks, changing learning rate of optimizer, etc), you can register callbacks for this purpose.

```
// test&save for each epoch
int epoch = 0;
timer t;
nn.fit<mse>(opt, train_images, train_labels, 50, 20,
         // called for each mini-batch
         [&](){
           t.elapsed();
           t.reset();
         },
         // called for each epoch
         [&](){
           result res = nn.test(test_images, test_labels);
           cout << res.num_success << "/" << res.num_total << endl;
           ofstream ofs (("epoch_"+to_string(epoch++)).c_str());
           ofs << nn;
         });
```

classification

As with regression task, you can use `network::fit` function in classification. Besides, if you have labels(class-id) for each training data, `network::train` can be used. Difference between `network::fit` and `network::train` is how to specify the desired outputs - `network::train` takes `label_t` type, instead of `vec_t`.

```
network<sequential> net;
adagrad opt;
net << layers::fc(2, 3) << activation::tanh()
    << layers::fc(3, 4) << activation::softmax();

// input_data[0] should be classified to id:3
// input_data[1] should be classified to id:1
std::vector<vec_t> input_data    { { 1, 0 }, { 0, 2 } };
std::vector<label_t> desired_out {        3,        1 };
size_t batch_size = 1;
size_t epochs = 30;

net.train<mse>(opt, input_data, desired_out, batch_size, epochs);
```

train graph model

If you train graph network, be sure to fed input/output data which has same shape to network's input/output layers.

```
network<graph>     net;
layers::input      in1(2);
layers::input      in2(2);
layers::concat     concat(2, 2);
layers::fc         fc(4, 2);
activation::relu   r();
adagrad opt;

(in1, in2) << concat;
concat << fc << r;
construct_graph(net, { &in1, &in2 }, { &r });

// 2training data, each data type is tensor_t and shape is [2x2]
//
//                        1st data for in1       2nd data for in1
//                              |                       |
//                              |   1st data for in2    |   2nd data for in2
//                              |         |             |         |
std::vector<tensor_t> data{ { { 1, 0 }, { 3, 2 } },{ { 0, 2 }, { 1, 1 } } };
std::vector<tensor_t> out { {            { 2, 5 } },{            { 3, 1 } } };

net.fit<mse>(opt, data, out, 1, 1);
```

without callback

```
...
adadelta optimizer;

// minibatch=50, epoch=20
nn.train<cross_entropy>(optimizer, train_images, train_labels, 50, 20);
```

with callback

```
...
adadelta optimizer;

// test&save for each epoch
int epoch = 0;
nn.train<cross_entropy>(optimizer, train_images, train_labels, 50, 20, [](){},
         [&](){
           result res = nn.test(test_images, test_labels);
           cout << res.num_success << "/" << res.num_total << endl;
           ofstream ofs (("epoch_"+to_string(epoch++)).c_str());
           ofs << nn;
         });
```

"freeze" layers

You can use `layer::set_trainable` to exclude a layer from updating its weights.

```
network<sequential> net = make_mlp({10,20,10});

net[1]->set_trainable(false); // freeze 2nd layer
```

use/evaluate trained model

predict a value

```
network<sequential> net;
// train network

vec_t in = {1.0, 2.0, 3.0};
vec_t result = net.predict(in);
```

```
double in[] = {1.0, 2.0, 3.0};
result = net.predict(in);
```

`predict` caclulates output vector for given input. You can use `vec_t`, `std::vector<float>`, `double[]` and any other **range** as input.

We also provide `predict_label` and `predict_max_value` for classification task.

```
void predict_mnist(network<sequential>& net, const vec_t& in) {
    std::cout << "result:" << net.predict_label(in) << std::endl;
    std::cout << "similarity:" << net.predict_max_value(in) << std::endl;
}
```

evaluate accuracy

calculate the loss

```
std::vector<vec_t> test_data;
std::vector<vec_t> test_target_values;

network<sequential> net;

// the lower, the better
double loss = net.get_loss<mse>(test_data, test_target_values);
```

You must specify loss-function by template parameter. We recommend you to use the same loss-function to training.

```
net.fit<cross_entropy>(...);
net.get_loss<mse>(...); // not recommended
net.get_loss<cross_entropy>(...); // ok :)
```

visualize the model

visualize graph networks

We can get graph structure in dot language format.

```
input_layer in1(shape3d(3,1,1));
input_layer in2(shape3d(3,1,1));
add added(2, 3);
```

```
linear_layer linear(3);
relu_layer relu();

(in1, in2) << added << linear << relu;
network<graph> net;

construct_graph(net, { &in1, &in2 }, { &linear } );

// generate graph model in dot language
std::ofstream ofs("graph_net_example.txt");
graph_visualizer viz(net, "graph");
viz.generate(ofs);
```

Once we get dot language model, we can easily get an image by graphviz:

```
dot -Tgif graph_net_example.txt -o graph.gif
```

Then you can get:

visualize each layer activations

```
network<sequential> nn;

nn << convolutional_layer(32, 32, 5, 3, 6) << tanh()
    << max_pooling_layer(28, 28, 6, 2) << tanh()
    << fully_connected_layer(14 * 14 * 6, 10) << tanh();
...
image img = nn[0]->output_to_image(); // visualize activations of recent input
img.write("layer0.bmp");
```

visualize convolution kernels

```
network<sequential> nn;

nn << conv(32, 32, 5, 3, 6) << tanh()
    << max_pool(28, 28, 6, 2) << tanh()
    << fc(14 * 14 * 6, 10) << tanh();
...
image img = nn.at<conv>(0).weight_to_image();
img.write("kernel0.bmp");
```

io

save and load the model

You can use `network::save` and `network::load` to save/load your model:

```
network<sequential> nn;

nn << convolutional_layer(32, 32, 5, 3, 6) << tanh()
    << max_pooling_layer(28, 28, 6, 2) << tanh()
    << fully_connected_layer(14 * 14 * 6, 10) << tanh();
```

```
...

nn.save("my-network");

network<sequential> nn2;
nn2.load("my-network");
```

The generated binary file will contain:

- the architecture of model
- the weights of the model

You can also select file format, and what you want to save:

```
// save the weights of the model in binary format
nn.save("my-weights", content_type::weights, file_format::binary);
nn.load("my-weights", content_type::weights, file_format::, file_format::json););

// save the architecture of the model in json format
nn.save("my-architecture", content_type::model, file_format::json);
nn.load("my-architecture", content_type::model, file_format::json);

// save both the architecture and the weights in binary format
// these are equivalent to nn.save("my-network") and nn.load("my-network")
nn.save("my-network", content_type::weights_and_model, file_format::binary);
nn.load("my-network", content_type::weights_and_model, file_format::binary);
```

If you want the architecture model in `string` format, you can use `to_json` and `from_json`.

```
std::string json = nn.to_json();

cout << json;

nn.from_json(json);
```

Note: `operator <<` and `operator >>` APIs before tiny-dnn v0.1.1 are deprecated.

import caffe's model

Import Caffe Model to tiny-dnn

reading data

from MNIST idx format

```
vector<vec_t> images;
vector<label_t> labels;
parse_mnist_images("train-images.idx3-ubyte", &images, -1.0, 1.0, 2, 2);
parse_mnist_labels("train-labels.idx1-ubyte", &labels);
```

from cifar-10 binary format

```
vector<vec_t> images;
vector<label_t> labels;
parse_cifar10("data_batch1.bin", &images, &labels, -1.0, 1.0, 0, 0);
```

reading images

You can use a simple `tiny_dnn::image` class to handle your images. JPEG (baseline & progressive), PNG (1/2/4/8 bit per channel), BMP (non-1bp, non-RLE), GIF are supported reading formats. Note that it's memory layout differs from OpenCV - it's layout is KHW (K:channels, H:height, W:width).

```
// default underlying type is uint8_t, and memory layout is KHW
// consider following 2x2 RGB image:
//
// R = [R0, R1,   G = [G0, G1,   B = [B0, B1,
//      R2, R3]        G2, G3]        B2, B3]
//
// memory layout of tiny_dnn::image is KHW, and order of channels K depends on its␣
↪image_type:
//
// gray_img = { gray(R0,G0,B0), gray(R1,G1,B1), gray(R2,G2,B2), gray(R3,G3,B3) }
// rgb_img = { R0, R1, R2, R3, G0, G1, G2, G3, B0, B1, B2, B3 }
// bgr_img = { B0, B1, B2, B3, G0, G1, G2, G3, R0, R1, R2, R3 }
//
// gray(r,g,b) = 0.300r + 0.586g + 0.113b
image<> gray_img("your-image.bmp", image_type::grayscale);
image<> rgb_img("your-image.bmp", image_type::rgb);
image<> bgr_img("your-image.bmp", image_type::bgr);

// convert into tiny-dnn's interface type
vec_t vec = img.to_vec();

// convert into HWK format with RGB order like:
// { R0, G0, B0, R1, G1, B1, R2, G2, B2, R3, G3, B3 }
std::vector<uint8_t> rgb = rgb_img.to_rgb();

// load data from HWK ordered array
rgb_img.from_rgb(rgb.begin(), rgb.end());

// resize image
image<> resized = resize_image(rgb_img, 256, 256);

// get the mean values (per channel)
image<float_t> mean = mean_image(resized);

// subtract mean values from image
image<float_t> subtracted = subtract_scalar(resized, mean);

// png,bmp are supported as saving types
subtracted.save("subtracted.png");
```

get/set the properties

traverse layers

```
// (1) get layers by operator[]
network<sequential> net;
net << conv(...)
    << fc(...);
```

```
layer* conv = net[0];
layer* fully_connected = net[1];
```

```
// (2) get layers using range-based for
for (layer* l : net) {
    std::cout << l->layer_type() << std::endl;
}
```

```
// (3) get layers using at<T> method
//     you can get derived class,

// throw nn_error if n-th layer can't be trated as T
conv* conv = net.at<conv>(0);
fc* fully_connected = net.at<fc>(1);
```

```
// (4) get layers and edges(tensors) using traverse method
graph_traverse(net[0],
    [](const layer& l) { // called for each node
        std::cout << l.layer_type() << std::endl;
    },
    [](const edge& e) { // called for each edge
        std::cout << e.vtype() << std::endl;
    });
```

get layer types

You can access each layer by operator[] after construction.

```
...
network<sequential> nn;

nn << convolutional_layer(32, 32, 5, 3, 6) << tanh()
    << max_pooling_layer(28, 28, 6, 2) << tanh()
    << fully_connected_layer(14 * 14 * 6, 10) << tanh();

for (int i = 0; i < nn.depth(); i++) {
    cout << "#layer:" << i << "\n";
    cout << "layer type:" << nn[i]->layer_type() << "\n";
    cout << "input:" << nn[i]->in_data_size() << "(" << nn[i]->in_data_shape() << ")\n
↪";
    cout << "output:" << nn[i]->out_data_size() << "(" << nn[i]->out_data_shape() <<
↪")\n";
}
```

output:

```
#layer:0
layer type:conv
input:3072([[32x32x3]])
output:4704([[28x28x6]])
num of parameters:456
#layer:1
layer type:max-pool
input:4704([[28x28x6]])
output:1176([[14x14x6]])
```

```
num of parameters:0
#layer:2
layer type:fully-connected
input:1176([[1176x1x1]])
output:10([[10x1x1]])
num of parameters:11770
```

get weight vector

```
std::vector<vec_t*> weights = nn[i]->weights();
```

Number of elements differs by layer types and settings. For example, in fully-connected layer with bias term, weights[0] represents weight matrix and weights[1] represents bias vector.

change the weight initialization

In neural network training, initial value of weight/bias can affect training speed and accuracy. In tiny-dnn, the weight is appropriately scaled by xavier algorithm1 and the bias is filled with 0.

To change initialization method (or weight-filler) and scaling factor, use `weight_init()` and `bias_init()` function of network and layer class.

- xavier ... automatic scaling using sqrt(scale / (fan-in + fan-out))
- lecun ... automatic scaling using scale / sqrt(fan-in)
- constant ... fill constant value

```
int num_units [] = { 100, 400, 100 };
auto nn = make_mlp<tanh>(num_units, num_units + 3);

// change all layers at once
nn.weight_init(weight_init::lecun());
nn.bias_init(weight_init::xavier(2.0));

// change specific layer
nn[0]->weight_init(weight_init::xavier(4.0));
nn[0]->bias_init(weight_init::constant(1.0));
```

change the seed value

You can change the seed value for the random value generator.

```
set_random_seed(3);
```

Note: Random value generator is shared among thread.

tune the performance

profile

```
timer t; // start the timer
//...
double elapsed_ms = t.elapsed();
t.reset();
```

change the number of threads while training

`CNN_TASK_SIZE` macro defines the number of threads for parallel training. Change it to smaller value will reduce memory footprint. This change affects execution time of training the network, but no affects on prediction.

```
// in config.h
#define CNN_TASK_SIZE 8
```

handle errors

When some error occurs, tiny-dnn doesn't print any message on stdout. Instead of `printf`, tiny-dnn throws exception. This behaviour is suitable when you integrate tiny-dnn into your application (especially embedded systems).

catch application exceptions

tiny-dnn may throw one of the following types:

- `tiny_dnn::nn_error`
- `tiny_dnn::not_implemented_error`
- `std::bad_alloc`

`not_implemented_error` is derived from `nn_error`, and they have `what()` method to provide detail message about the error.

```
try {
   network<sequential> nn;
   ...
} catch (const nn_error& e) {
   cout << e.what();
}
```

Integrate with your application

Because tiny-dnn is header-only, integrating it with your application is extremely easy. We explain how to do it step-by-step.

Step1/3: Include tiny_dnn.h in your application

Just add the following line:

```
#include "tiny_dnn/tiny_dnn.h"
```

Step2/3: Enable C++11 options

tiny-dnn uses C++11's core features and libraries. You must use the c++11 compliant compiler and compile with c++11-mode.

Visual Studio(2013-)

C++11 features are enabled by default, you have nothing to do about it.

gcc(4.8-)/clang(3.3-)

Use `-std=c++11` option to enable c++11-mode.

> From gcc 6.0, the default compile mode for c++ is -std=gnu++14, so you don't need to add this option.

Step3/3: Add include path of tiny-dnn to your build system

Tell your build system where tiny-dnn exists. In gcc:

```
g++ -std=c++11 -Iyour-downloaded-path -O3 your-app.cpp -o your-app
```

> Another solution: place tiny-dnn's header files under your project root

Train network with your original dataset

Here are some examples.

1. using opencv (image file => vec_t)

```
#include <opencv2/imgcodecs.hpp>
#include <opencv2/imgproc.hpp>
#include <boost/foreach.hpp>
#include <boost/filesystem.hpp>
using namespace boost::filesystem;

// convert image to vec_t
void convert_image(const std::string& imagefilename,
                   double scale,
                   int w,
                   int h,
                   std::vector<vec_t>& data)
{
    auto img = cv::imread(imagefilename, cv::IMREAD_GRAYSCALE);
    if (img.data == nullptr) return; // cannot open, or it's not an image

    cv::Mat_<uint8_t> resized;
    cv::resize(img, resized, cv::Size(w, h));
    vec_t d;

    std::transform(resized.begin(), resized.end(), std::back_inserter(d),
                   [=](uint8_t c) { return c * scale; });
```

```
    data.push_back(d);
}

// convert all images found in directory to vec_t
void convert_images(const std::string& directory,
                    double scale,
                    int w,
                    int h,
                    std::vector<vec_t>& data)
{
    path dpath(directory);

    BOOST_FOREACH(const path& p,
                  std::make_pair(directory_iterator(dpath), directory_iterator())) {
        if (is_directory(p)) continue;
        convert_image(p.string(), scale, w, h, data);
    }
}
```

Another example can be found in issue#16, which can treat color channels.

2. using mnisten (image file => idx format)

mnisten is a library to convert image files to idx format.

```
mnisten -d my_image_files_directory_name -o my_prefix -s 32x32
```

After generating idx files, you can use parse_mnist_images / parse_mnist_labels utilities in mnist_parser.h

3. from levelDB (caffe style => [vec_t, label_t])

Caffe supports levelDB data format. Following code can convert levelDB created by Caffe into data/label arrays.

```
#include "leveldb/db.h"

void convert_leveldb(const std::string& dbname,
                     double scale,
                     std::vector<vec_t>& data,
                     std::vector<label_t>& label)
{
    leveldb::DB *db;
    leveldb::Options options;
    options.create_if_missing = false;
    auto status = leveldb::DB::Open(options, dbname, &db);

    leveldb::Iterator* it = db->NewIterator(leveldb::ReadOptions());
    for (it->SeekToFirst(); it->Valid(); it->Next()) {
        const char* src = it->value().data();
        size_t sz = it->value().size();
        vec_t d;
        std::transform(src, src + sz - 1, std::back_inserter(d),
                       [=](char c){ return c * scale; });
        data.push_back(d);
        label.push_back(src[sz - 1]);
    }
```

```
    delete it;
    delete db;
}
```

Layers:

Layers

[source]

elementwise_add_layer

element-wise add N vectors `y_i = x0_i + x1_i + ... + xnum_i`

Constructors

```
    elementwise_add_layer(size_t num_args, size_t dim)
```

- **dim** number of elements for each input
- **num_args** number of inputs

[source]

average_pooling_layer

average pooling with trainable weights

Constructors

```
    average_pooling_layer(size_t in_width,
                          size_t in_height,
                          size_t in_channels,
                          size_t pool_size)
```

- **in_height** height of input image
- **in_channels** the number of input image channels(depth)
- **in_width** width of input image
- **pool_size** factor by which to downscale

```
    average_pooling_layer(size_t in_width,
                          size_t in_height,
                          size_t in_channels,
                          size_t pool_size,
                          size_t stride)
```

- **in_height** height of input image
- **stride** interval at which to apply the filters to the input

- **in_channels** the number of input image channels(depth)
- **in_width** width of input image
- **pool_size** factor by which to downscale

```
average_pooling_layer(size_t     in_width,
                      size_t     in_height,
                      size_t     in_channels,
                      size_t     pool_size_x,
                      size_t     pool_size_y,
                      size_t     stride_x,
                      size_t     stride_y,
                      padding        pad_type = padding::valid)
```

- **in_height** height of input image
- **pad_type** padding mode(same/valid)
- **in_channels** the number of input image channels(depth)
- **pool_size_x** factor by which to downscale
- **pool_size_y** factor by which to downscale
- **in_width** width of input image
- **stride_x** interval at which to apply the filters to the input
- **stride_y** interval at which to apply the filters to the input

[source]

average_unpooling_layer

average pooling with trainable weights

Constructors

```
average_unpooling_layer(size_t in_width,
                        size_t in_height,
                        size_t in_channels,
                        size_t pooling_size)
```

- **in_height** height of input image
- **in_channels** the number of input image channels(depth)
- **in_width** width of input image
- **pooling_size** factor by which to upscale

```
average_unpooling_layer(size_t in_width,
                        size_t in_height,
                        size_t in_channels,
                        size_t pooling_size,
                        size_t stride)
```

- **in_height** height of input image
- **stride** interval at which to apply the filters to the input

- **in_channels** the number of input image channels(depth)
- **in_width** width of input image
- **pooling_size** factor by which to upscale

[source]

batch_normalization_layer

Batch Normalization

Normalize the activations of the previous layer at each batch

Constructors

```
batch_normalization_layer(const layer& prev_layer,
                          float_t epsilon = 1e-5,
                          float_t momentum = 0.999,
                          net_phase phase = net_phase::train)
```

- **phase** specify the current context (train/test)
- **epsilon** small positive value to avoid zero-division
- **prev_layer** previous layer to be connected with this layer
- **momentum** momentum in the computation of the exponential average of the mean/stddev of the data

```
batch_normalization_layer(size_t in_spatial_size,
                          size_t in_channels,
                          float_t epsilon = 1e-5,
                          float_t momentum = 0.999,
                          net_phase phase = net_phase::train)
```

- **phase** specify the current context (train/test)
- **in_channels** channels of the input data
- **in_spatial_size** spatial size (WxH) of the input data
- **momentum** momentum in the computation of the exponential average of the mean/stddev of the data
- **epsilon** small positive value to avoid zero-division

[source]

concat_layer

concat N layers along depth

```
// in: [3,1,1],[3,1,1] out: [3,1,2] (in W,H,K order)
concat_layer l1(2,3);

// in: [3,2,2],[3,2,5] out: [3,2,7] (in W,H,K order)
concat_layer l2({shape3d(3,2,2),shape3d(3,2,5)});
```

Constructors

```
concat_layer(const std::vector<shape3d>& in_shapes)
```

- **in_shapes** shapes of input tensors

```
concat_layer(size_t num_args, size_t ndim)
```

- **ndim** number of elements for each input
- **num_args** number of input tensors

[source]

convolutional_layer

2D convolution layer

take input as two-dimensional *image* and applying filtering operation.

Constructors

```
convolutional_layer(size_t in_width,
                    size_t in_height,
                    size_t window_size,
                    size_t in_channels,
                    size_t out_channels,
                    padding    pad_type = padding::valid,
                    bool       has_bias = true,
                    size_t w_stride = 1,
                    size_t h_stride = 1,
                    backend_t  backend_type = core::default_engine())
```

- **in_height** input image height
- **h_stride** specify the vertical interval at which to apply the filters to the input
- **window_size** window(kernel) size of convolution
- **has_bias** whether to add a bias vector to the filter outputs
- **out_channels** output image channels
- **w_stride** specify the horizontal interval at which to apply the filters to the input
- **padding** rounding strategy
  - valid: use valid pixels of input only. `output-size = (in-width - window_width + 1) * (in-height - window_height + 1) * out_channels`
  - same: add zero-padding to keep same width/height. `output-size = in-width * in-height * out_channels`
- **in_channels** input image channels (grayscale=1, rgb=3)
- **backend_type** specify backend engine you use
- **in_width** input image width

```
convolutional_layer(size_t in_width,
                    size_t in_height,
                    size_t window_width,
                    size_t window_height,
                    size_t in_channels,
                    size_t out_channels,
                    padding    pad_type = padding::valid,
                    bool       has_bias = true,
                    size_t w_stride = 1,
                    size_t h_stride = 1,
                    backend_t  backend_type = core::default_engine())
```

- **in_height** input image height
- **h_stride** specify the vertical interval at which to apply the filters to the input
- **backend_type** specify backend engine you use
- **has_bias** whether to add a bias vector to the filter outputs
- **out_channels** output image channels
- **w_stride** specify the horizontal interval at which to apply the filters to the input
- **window_height** window_height(kernel) size of convolution
- **window_width** window_width(kernel) size of convolution
- **padding** rounding strategy
  - valid: use valid pixels of input only. `output-size = (in-width - window_width + 1) * (in-height - window_height + 1) * out_channels`
  - same: add zero-padding to keep same width/height. `output-size = in-width * in-height * out_channels`
- **in_channels** input image channels (grayscale=1, rgb=3)
- **in_width** input image width

```
convolutional_layer(size_t                in_width,
                    size_t                in_height,
                    size_t                window_size,
                    size_t                in_channels,
                    size_t                out_channels,
                    const connection_table& connection_table,
                    padding                   pad_type = padding::valid,
                    bool                      has_bias = true,
                    size_t                w_stride = 1,
                    size_t                h_stride = 1,
                    backend_t      backend_type = core::default_engine())
```

- **in_height** input image height
- **window_size** window(kernel) size of convolution
- **has_bias** whether to add a bias vector to the filter outputs
- **connection_table** definition of connections between in-channels and out-channels
- **out_channels** output image channels
- **w_stride** specify the horizontal interval at which to apply the filters to the input

- **h_stride** specify the vertical interval at which to apply the filters to the input
- **pad_type** rounding strategy
  - valid: use valid pixels of input only. `output-size = (in-width - window_width + 1) * (in-height - window_height + 1) * out_channels`
  - same: add zero-padding to keep same width/height. `output-size = in-width * in-height * out_channels`
- **in_channels** input image channels (grayscale=1, rgb=3)
- **backend_type** specify backend engine you use
- **in_width** input image width

```
convolutional_layer(size_t                 in_width,
                    size_t                 in_height,
                    size_t                 window_width,
                    size_t                 window_height,
                    size_t                 in_channels,
                    size_t                 out_channels,
                    const connection_table& connection_table,
                    padding                    pad_type = padding::valid,
                    bool                       has_bias = true,
                    size_t                 w_stride = 1,
                    size_t                 h_stride = 1,
                    backend_t      backend_type = core::default_engine())
```

- **in_height** input image height
- **backend_type** specify backend engine you use
- **has_bias** whether to add a bias vector to the filter outputs
- **connection_table** definition of connections between in-channels and out-channels
- **out_channels** output image channels
- **w_stride** specify the horizontal interval at which to apply the filters to the input
- **window_height** window_height(kernel) size of convolution
- **window_width** window_width(kernel) size of convolution
- **h_stride** specify the vertical interval at which to apply the filters to the input
- **pad_type** rounding strategy
  - valid: use valid pixels of input only. `output-size = (in-width - window_width + 1) * (in-height - window_height + 1) * out_channels`
  - same: add zero-padding to keep same width/height. `output-size = in-width * in-height * out_channels`
- **in_channels** input image channels (grayscale=1, rgb=3)
- **in_width** input image width

[source]

deconvolutional_layer

2D deconvolution layer

take input as two-dimensional *image* and applying filtering operation.

Constructors

```
deconvolutional_layer(size_t     in_width,
                      size_t     in_height,
                      size_t     window_size,
                      size_t     in_channels,
                      size_t     out_channels,
                      padding        pad_type = padding::valid,
                      bool           has_bias = true,
                      size_t     w_stride = 1,
                      size_t     h_stride = 1,
                      backend_t      backend_type = core::default_engine())
```

- **in_height** input image height
- **h_stride** specify the vertical interval at which to apply the filters to the input
- **window_size** window(kernel) size of convolution
- **has_bias** whether to add a bias vector to the filter outputs
- **out_channels** output image channels
- **w_stride** specify the horizontal interval at which to apply the filters to the input
- **padding** rounding strategy valid: use valid pixels of input only. output-size = (in-width - window_size + 1) * (in-height - window_size + 1) * out_channels same: add zero-padding to keep same width/height. output-size = in-width * in-height * out_channels
- **in_channels** input image channels (grayscale=1, rgb=3)
- **in_width** input image width

```
deconvolutional_layer(size_t     in_width,
                      size_t     in_height,
                      size_t     window_width,
                      size_t     window_height,
                      size_t     in_channels,
                      size_t     out_channels,
                      padding        pad_type = padding::valid,
                      bool           has_bias = true,
                      size_t     w_stride = 1,
                      size_t     h_stride = 1,
                      backend_t      backend_type = core::default_engine())
```

- **in_height** input image height
- **h_stride** specify the vertical interval at which to apply the filters to the input
- **has_bias** whether to add a bias vector to the filter outputs
- **out_channels** output image channels
- **w_stride** specify the horizontal interval at which to apply the filters to the input
- **window_height** window_height(kernel) size of convolution
- **window_width** window_width(kernel) size of convolution

- **padding** rounding strategy valid: use valid pixels of input only. output-size = (in-width - window_width + 1) * (in-height - window_height + 1) * out_channels same: add zero-padding to keep same width/height. output-size = in-width * in-height * out_channels
- **in_channels** input image channels (grayscale=1, rgb=3)
- **in_width** input image width

```
    deconvolutional_layer(size_t                in_width,
                          size_t                in_height,
                          size_t                window_size,
                          size_t                in_channels,
                          size_t                out_channels,
                          const connection_table& connection_table,
                          padding                   pad_type = padding::valid,
                          bool                      has_bias = true,
                          size_t                w_stride = 1,
                          size_t                h_stride = 1,
                          backend_t                 backend_type = core::default_
↪engine())
```

- **in_height** input image height
- **window_size** window(kernel) size of convolution
- **has_bias** whether to add a bias vector to the filter outputs
- **connection_table** definition of connections between in-channels and out-channels
- **out_channels** output image channels
- **w_stride** specify the horizontal interval at which to apply the filters to the input
- **h_stride** specify the vertical interval at which to apply the filters to the input
- **pad_type** rounding strategy valid: use valid pixels of input only. output-size = (in-width - window_size + 1) * (in-height - window_size + 1) * out_channels same: add zero-padding to keep same width/height. output-size = in-width * in-height * out_channels
- **in_channels** input image channels (grayscale=1, rgb=3)
- **in_width** input image width

```
    deconvolutional_layer(size_t                in_width,
                          size_t                in_height,
                          size_t                window_width,
                          size_t                window_height,
                          size_t                in_channels,
                          size_t                out_channels,
                          const connection_table& connection_table,
                          padding                   pad_type = padding::valid,
                          bool                      has_bias = true,
                          size_t                w_stride = 1,
                          size_t                h_stride = 1,
                          backend_t                 backend_type = core::default_
↪engine())
```

- **in_height** input image height
- **has_bias** whether to add a bias vector to the filter outputs
- **connection_table** definition of connections between in-channels and out-channels

- **out_channels** output image channels
- **w_stride** specify the horizontal interval at which to apply the filters to the input
- **window_height** window_height(kernel) size of convolution
- **window_width** window_width(kernel) size of convolution
- **h_stride** specify the vertical interval at which to apply the filters to the input
- **pad_type** rounding strategy valid: use valid pixels of input only. output-size = (in-width - window_size + 1) * (in-height - window_size + 1) * out_channels same: add zero-padding to keep same width/height. output-size = in-width * in-height * out_channels
- **in_channels** input image channels (grayscale=1, rgb=3)
- **in_width** input image width

[source]

dropout_layer

applies dropout to the input

Constructors

```
    dropout_layer(size_t in_dim, float_t dropout_rate, net_phase phase = net_
↪phase::train)
```

- **phase** initial state of the dropout
- **dropout_rate** (0-1) fraction of the input units to be dropped
- **in_dim** number of elements of the input

[source]

feedforward_layer

single-input, single-output network with activation function

Constructors

[source]

fully_connected_layer

compute fully-connected(matmul) operation

Constructors

```
    fully_connected_layer(size_t in_dim,
                          size_t out_dim,
                          bool      has_bias = true,
                          backend_t  backend_type = core::default_engine())
```

```
```

- **out_dim** number of elements of the output
- **has_bias** whether to include additional bias to the layer
- **in_dim** number of elements of the input

[source]

input_layer

Constructors

[source]

linear_layer

element-wise operation: `f(x) = h(scale*x+bias)`

Constructors

```
linear_layer(size_t dim, float_t scale = float_t(1)
```

- **dim** number of elements
- **scale** factor by which to multiply
- **bias** bias term

[source]

lrn_layer

local response normalization

Constructors

```
lrn_layer(layer*      prev,
          size_t      local_size,
          float_t     alpha = 1.0,
          float_t     beta  = 5.0,
          norm_region region = norm_region::across_channels)
```

- **beta** the scaling parameter (same to caffe's LRN)
- **alpha** the scaling parameter (same to caffe's LRN)
- **layer** the previous layer connected to this
- **in_channels** the number of channels of input data
- **local_size** the number of channels(depths) to sum over

```
lrn_layer(size_t  in_width,
          size_t  in_height,
          size_t  local_size,
          size_t  in_channels,
          float_t     alpha = 1.0,
          float_t     beta  = 5.0,
          norm_region region = norm_region::across_channels)
```

- **in_height** the height of input data
- **local_size** the number of channels(depths) to sum over
- **beta** the scaling parameter (same to caffe's LRN)
- **in_channels** the number of channels of input data
- **alpha** the scaling parameter (same to caffe's LRN)
- **in_width** the width of input data

[source]

max_pooling_layer

Constructors

```
max_pooling_layer(size_t in_width,
                  size_t in_height,
                  size_t in_channels,
                  size_t pooling_size,
                  backend_t  backend_type = core::default_engine())
```

- **in_height** height of input image
- **in_channels** the number of input image channels(depth)
- **in_width** width of input image
- **pooling_size** factor by which to downscale

```
max_pooling_layer(size_t in_width,
                  size_t in_height,
                  size_t in_channels,
                  size_t pooling_size_x,
                  size_t pooling_size_y,
                  size_t stride_x,
                  size_t stride_y,
                  padding    pad_type = padding::valid,
                  backend_t  backend_type = core::default_engine())
```

- **in_height** height of input image
- **stride** interval at which to apply the filters to the input
- **in_channels** the number of input image channels(depth)
- **in_width** width of input image
- **pooling_size** factor by which to downscale

[source]

max_unpooling_layer

Constructors

```
max_unpooling_layer(size_t in_width,
                    size_t in_height,
                    size_t in_channels,
                    size_t unpooling_size)
```

- **in_height** height of input image
- **in_channels** the number of input image channels(depth)
- **in_width** width of input image
- **unpooling_size** factor by which to upscale

```
max_unpooling_layer(size_t in_width,
                    size_t in_height,
                    size_t in_channels,
                    size_t unpooling_size,
                    size_t stride)
```

- **in_height** height of input image
- **stride** interval at which to apply the filters to the input
- **in_channels** the number of input image channels(depth)
- **in_width** width of input image
- **unpooling_size** factor by which to upscale

[source]

partial_connected_layer

Constructors

[source]

power_layer

element-wise pow: `y = scale*x^factor`

Constructors

```
power_layer(const shape3d& in_shape, float_t factor, float_t scale=1.0f)
```

- **factor** floating-point number that specifies a power
- **scale** scale factor for additional multiply
- **in_shape** shape of input tensor

```
power_layer(const layer& prev_layer, float_t factor, float_t scale=1.0f)
```

- **prev_layer** previous layer to be connected
- **scale** scale factor for additional multiply
- **factor** floating-point number that specifies a power

[source]

quantized_convolutional_layer

2D convolution layer

take input as two-dimensional *image* and applying filtering operation.

Constructors

```
quantized_convolutional_layer(size_t      in_width,
                              size_t      in_height,
                              size_t      window_size,
                              size_t      in_channels,
                              size_t      out_channels,
                              padding         pad_type = padding::valid,
                              bool            has_bias = true,
                              size_t      w_stride = 1,
                              size_t      h_stride = 1,
                              backend_t       backend_type = core::backend_
↪t::internal)
```

- **in_height** input image height
- **h_stride** specify the vertical interval at which to apply the filters to the input
- **window_size** window(kernel) size of convolution
- **has_bias** whether to add a bias vector to the filter outputs
- **out_channels** output image channels
- **w_stride** specify the horizontal interval at which to apply the filters to the input
- **padding** rounding strategy valid: use valid pixels of input only. output-size = (in-width - window_size + 1) * (in-height - window_size + 1) * out_channels same: add zero-padding to keep same width/height. output-size = in-width * in-height * out_channels
- **in_channels** input image channels (grayscale=1, rgb=3)
- **in_width** input image width

```
quantized_convolutional_layer(size_t      in_width,
                              size_t      in_height,
                              size_t      window_width,
                              size_t      window_height,
                              size_t      in_channels,
                              size_t      out_channels,
                              padding         pad_type = padding::valid,
                              bool            has_bias = true,
                              size_t      w_stride = 1,
                              size_t      h_stride = 1,
                              backend_t       backend_type = core::backend_
↪t::internal)
```

- **in_height** input image height
- **h_stride** specify the vertical interval at which to apply the filters to the input
- **has_bias** whether to add a bias vector to the filter outputs
- **out_channels** output image channels
- **w_stride** specify the horizontal interval at which to apply the filters to the input
- **window_height** window_height(kernel) size of convolution
- **window_width** window_width(kernel) size of convolution
- **padding** rounding strategy valid: use valid pixels of input only. output-size = (in-width - window_width + 1) * (in-height - window_height + 1) * out_channels same: add zero-padding to keep same width/height. output-size = in-width * in-height * out_channels
- **in_channels** input image channels (grayscale=1, rgb=3)
- **in_width** input image width

```
quantized_convolutional_layer(size_t                  in_width,
                              size_t                  in_height,
                              size_t                  window_size,
                              size_t                  in_channels,
                              size_t                  out_channels,
                              const connection_table& connection_table,
                              padding                     pad_type = padding::valid,
                              bool                        has_bias = true,
                              size_t                  w_stride = 1,
                              size_t                  h_stride = 1,
                              backend_t backend_type = core::backend_t::internal)
```

- **in_height** input image height
- **window_size** window(kernel) size of convolution
- **has_bias** whether to add a bias vector to the filter outputs
- **connection_table** definition of connections between in-channels and out-channels
- **out_channels** output image channels
- **w_stride** specify the horizontal interval at which to apply the filters to the input
- **h_stride** specify the vertical interval at which to apply the filters to the input
- **pad_type** rounding strategy valid: use valid pixels of input only. output-size = (in-width - window_size + 1) * (in-height - window_size + 1) * out_channels same: add zero-padding to keep same width/height. output-size = in-width * in-height * out_channels
- **in_channels** input image channels (grayscale=1, rgb=3)
- **in_width** input image width

```
quantized_convolutional_layer(size_t                  in_width,
                              size_t                  in_height,
                              size_t                  window_width,
                              size_t                  window_height,
                              size_t                  in_channels,
                              size_t                  out_channels,
                              const connection_table& connection_table,
                              padding                     pad_type = padding::valid,
                              bool                        has_bias = true,
```

```
                                                size_t              w_stride = 1,
                                                size_t              h_stride = 1,
                                                backend_t      backend_type = core::backend_
↪t::internal)
```

- **in_height** input image height
- **has_bias** whether to add a bias vector to the filter outputs
- **connection_table** definition of connections between in-channels and out-channels
- **out_channels** output image channels
- **w_stride** specify the horizontal interval at which to apply the filters to the input
- **window_height** window_height(kernel) size of convolution
- **window_width** window_width(kernel) size of convolution
- **h_stride** specify the vertical interval at which to apply the filters to the input
- **pad_type** rounding strategy valid: use valid pixels of input only. output-size = (in-width - window_size + 1) * (in-height - window_size + 1) * out_channels same: add zero-padding to keep same width/height. output-size = in-width * in-height * out_channels
- **in_channels** input image channels (grayscale=1, rgb=3)
- **in_width** input image width

[source]

quantized_deconvolutional_layer

2D deconvolution layer

take input as two-dimensional *image* and applying filtering operation.

Constructors

```
    quantized_deconvolutional_layer(size_t     in_width,
                                    size_t     in_height,
                                    size_t     window_size,
                                    size_t     in_channels,
                                    size_t     out_channels,
                                    padding        pad_type = padding::valid,
                                    bool           has_bias = true,
                                    size_t     w_stride = 1,
                                    size_t     h_stride = 1,
                                    backend_t      backend_type = core::backend_
↪t::internal)
```

- **in_height** input image height
- **h_stride** specify the vertical interval at which to apply the filters to the input
- **window_size** window(kernel) size of convolution
- **has_bias** whether to add a bias vector to the filter outputs
- **out_channels** output image channels

- **w_stride** specify the horizontal interval at which to apply the filters to the input
- **padding** rounding strategy valid: use valid pixels of input only. output-size = (in-width - window_size + 1) * (in-height - window_size + 1) * out_channels same: add zero-padding to keep same width/height. output-size = in-width * in-height * out_channels
- **in_channels** input image channels (grayscale=1, rgb=3)
- **in_width** input image width

```
quantized_deconvolutional_layer(size_t     in_width,
                                size_t     in_height,
                                size_t     window_width,
                                size_t     window_height,
                                size_t     in_channels,
                                size_t     out_channels,
                                padding        pad_type = padding::valid,
                                bool           has_bias = true,
                                size_t     w_stride = 1,
                                size_t     h_stride = 1,
                                backend_t      backend_type = core::backend_
↪t::internal)
```

- **in_height** input image height
- **h_stride** specify the vertical interval at which to apply the filters to the input
- **has_bias** whether to add a bias vector to the filter outputs
- **out_channels** output image channels
- **w_stride** specify the horizontal interval at which to apply the filters to the input
- **window_height** window_height(kernel) size of convolution
- **window_width** window_width(kernel) size of convolution
- **padding** rounding strategy valid: use valid pixels of input only. output-size = (in-width - window_width + 1) * (in-height - window_height + 1) * out_channels same: add zero-padding to keep same width/height. output-size = in-width * in-height * out_channels
- **in_channels** input image channels (grayscale=1, rgb=3)
- **in_width** input image width

```
quantized_deconvolutional_layer(size_t                 in_width,
                                size_t                 in_height,
                                size_t                 window_size,
                                size_t                 in_channels,
                                size_t                 out_channels,
                                const connection_table& connection_table,
                                padding                    pad_type = padding::valid,
                                bool                       has_bias = true,
                                size_t                 w_stride = 1,
                                size_t                 h_stride = 1,
                                backend_t                  backend_type =␣
↪core::backend_t::internal)
```

- **in_height** input image height
- **window_size** window(kernel) size of convolution
- **has_bias** whether to add a bias vector to the filter outputs

- **connection_table** definition of connections between in-channels and out-channels
- **out_channels** output image channels
- **w_stride** specify the horizontal interval at which to apply the filters to the input
- **h_stride** specify the vertical interval at which to apply the filters to the input
- **pad_type** rounding strategy valid: use valid pixels of input only. output-size = (in-width - window_size + 1) * (in-height - window_size + 1) * out_channels same: add zero-padding to keep same width/height. output-size = in-width * in-height * out_channels
- **in_channels** input image channels (grayscale=1, rgb=3)
- **in_width** input image width

```
quantized_deconvolutional_layer(size_t                    in_width,
                                size_t                    in_height,
                                size_t                    window_width,
                                size_t                    window_height,
                                size_t                    in_channels,
                                size_t                    out_channels,
                                const connection_table& connection_table,
                                padding                    pad_type = padding::valid,
                                bool                       has_bias = true,
                                size_t                    w_stride = 1,
                                size_t                    h_stride = 1,
                                backend_t                  backend_type =␣
↪core::backend_t::internal)
```

- **in_height** input image height
- **has_bias** whether to add a bias vector to the filter outputs
- **connection_table** definition of connections between in-channels and out-channels
- **out_channels** output image channels
- **w_stride** specify the horizontal interval at which to apply the filters to the input
- **window_height** window_height(kernel) size of convolution
- **window_width** window_width(kernel) size of convolution
- **h_stride** specify the vertical interval at which to apply the filters to the input
- **pad_type** rounding strategy valid: use valid pixels of input only. output-size = (in-width - window_size + 1) * (in-height - window_size + 1) * out_channels same: add zero-padding to keep same width/height. output-size = in-width * in-height * out_channels
- **in_channels** input image channels (grayscale=1, rgb=3)
- **in_width** input image width

[source]

quantized_fully_connected_layer

compute fully-connected(matmul) operation

Constructors

```
    quantized_fully_connected_layer(size_t in_dim,
                                    size_t out_dim,
                                    bool      has_bias = true,
                                    backend_t  backend_type = core::backend_
↪t::internal)
```

- **out_dim** number of elements of the output
- **has_bias** whether to include additional bias to the layer
- **in_dim** number of elements of the input

[source]

slice_layer

slice an input data into multiple outputs along a given slice dimension.

Constructors

```
    slice_layer(const shape3d& in_shape, slice_type slice_type, size_t num_outputs)
```

- **num_outputs** number of output layers

  example1: input: NxKxWxH = 4x3x2x2 (N:batch-size, K:channels, W:width, H:height) slice_type: slice_samples num_outputs: 3

  output[0]: 1x3x2x2 output[1]: 1x3x2x2 output[2]: 2x3x2x2 (mod data is assigned to the last output)

  example2: input: NxKxWxH = 4x6x2x2 slice_type: slice_channels num_outputs: 3

  output[0]: 4x2x2x2 output[1]: 4x2x2x2 output[2]: 4x2x2x2

- **slice_type** target axis of slicing

Examples(Link to github):

- MNIST image classification
- Cifar-10 image classification
- Deconvolutional Auto-encoder
- Importing Caffe's model into tiny-dnn

Update Logs:

Changing from v0.0.1

This section explains the API changes from v0.0.1.

How to specify the loss and the optimizer

In v0.0.1, the loss function and the optimization algorithm are treated as template parameter of `network`.

```
//v0.0.1
network<mse, adagrad> net;
net.train(x_data, y_label, n_batch, n_epoch);
```

From v0.1.0, these are treated as parameters of train/fit functions.

```
//v0.1.0
network<sequential> net;
adagrad opt;
net.fit<mse>(opt, x_data, y_label, n_batch, n_epoch);
```

Training API for regression

In v0.0.1, the regression and the classification have the same API:

```
//v0.0.1
net.train(x_data, y_data, n_batch, n_epoch); // regression
net.train(x_data, y_label, n_batch, n_epoch); // classification
```

From v0.1.0, these are separated into `fit` and `train`.

```
//v0.1.0
net.fit<mse>(opt, x_data, y_data, n_batch, n_epoch); // regression
net.train<mse>(opt, x_data, y_label, n_batch, n_epoch); // classification
```

The default mode of re-init weights

In v0.0.1, the default mode of weight-initialization in `trian` function is `reset_weights=true`.

```
//v0.0.1
std::ifstream is("model");
is >> net;
net.train(x_data, y_data, n_batch, n_epoch); // reset loaded weights automatically
net.train(x_data, y_data, n_bacth, n_epoch); // again we lost trained parameter we␣
↪got the last training
```

```
//v0.1.0
std::ifstream is("model");
is >> net;
net.trian<mse>(opt, x_data, y_data, n_batch, n_epoch); // hold loaded weights
net.train<mse>(opt, x_data, y_data, n_bacth, n_epoch); // continue training from the␣
↪last training

// weights are automatically initialized if we don't load models and train yet
net2.train<mse>(opt, x_data, y_data, n_batch, n_epoch);
```

Developer Guides:

Adding a new layer

This section describes how to create a new layer incorporated with tiny-dnn. Let's create simple fully-connected layer for example.

> Note: This document is old, and doesn't fit to current tiny-dnn. We need to update.

Declare class

Let's define your layer. All of layer operations in tiny-dnn are derived from `layer` class.

```
// calculate y = Wx + b
class fully_connected : public layer {
public:
    //todo
};
```

the `layer` class prepares input/output data for your calculation. To do this, you must tell `layer`'s constructor what you need.

```
layer::layer(const std::vector<vector_type>& in_type,
             const std::vector<vector_type>& out_type)
```

For example, consider calculating fully-connected operation: `y = Wx + b`. In this caluculation, Input (right hand of this eq) is data `x`, weight `W` and bias `b`. Output is, of course `y`. So it's constructor should pass {data,weight,bias} as input and {data} as output.

```
// calculate y = Wx + b
class fully_connected : public layer {
public:
    fully_connected(size_t x_size, size_t y_size)
    :layer({vector_type::data,vector_type::weight,vector_type::bias}, // x, W and b
           {vector_type::data}),
     x_size_(x_size),
     y_size_(y_size)
    {}

private:
    size_t x_size_; // number of input elements
    size_t y_size_; // number of output elements
};
```

the `vector_type::data` is some input data passed by previous layer, or output data consumed by next layer. `vector_type::weight` and `vector_type::bias` represents trainable parameters. The only difference between them is default initialization method: `weight` is initialized by random value, and `bias` is initialized by zero-vector (this behaviour can be changed by network::weight_init method). If you need another vector to calculate, `vector_type::aux` can be used.

Implement virtual method

There are 5 methods to implement. In most case 3 methods are written as one-liner and remaining 2 are essential:

- layer_type
- in_shape

- out_shape
- forward_propagation
- back_propagation

layer_type

Returns name of your layer.

```
std::string layer_type() const override {
    return "fully-connected";
}
```

in_shape/out_shape

Returns input/output shapes corresponding to inputs/outputs. Shapes is defined by [width, height, depth]. For example fully-connected layer treats input data as 1-dimensional array, so its shape is [N, 1, 1].

```
std::vector<shape3d> in_shape() const override {
    // return input shapes
    // order of shapes must be equal to argument of layer constructor
    return { shape3d(x_size_, 1, 1), // x
             shape3d(x_size_, y_size_, 1), // W
             shape3d(y_size_, 1, 1) }; // b
}

std::vector<shape3d> out_shape() const override {
    return { shape3d(y_size_, 1, 1) }; // y
}
```

forward_propagation

Execute forward calculation in this method.

```
void forward_propagation(size_t worker_index,
                         const std::vector<vec_t*>& in_data,
                         std::vector<vec_t*>& out_data) override {
    const vec_t& x = *in_data[0]; // it's size is in_shapes()[0] (=[x_size_,1,1])
    const vec_t& W = *in_data[1];
    const vec_t& b = *in_data[2];
    vec_t& y = *out_data[0];

    std::fill(y.begin(), y.end(), 0.0);

    // y = Wx+b
    for (size_t r = 0; r < y_size_; r++) {
        for (size_t c = 0; c < x_size_; c++)
            y[r] += W[r*x_size_+c]*x[c];
        y[r] += b[r];
    }
}
```

the `in_data/out_data` is array of input/output data, which is ordered as you told `layer`'s constructor. The implementation is simple and straightforward, isn't it?

`worker_index` is task-id. It is always zero if you run tiny-dnn in single thread. If some class member variables are updated while forward/backward pass, these members must be treated carefully to avoid data race. If their variables are task-independent, your class can hold just N variables and access them by worker_index (you can see this example in max_pooling_layer.h). input/output data managed by `layer` base class is *task-local*, so `in_data/out_data` is treated as if it is running on single thread.

back propagation

```
void back_propagation(size_t                index,
                      const std::vector<vec_t*>& in_data,
                      const std::vector<vec_t*>& out_data,
                      std::vector<vec_t*>&       out_grad,
                      std::vector<vec_t*>&       in_grad) override {
    const vec_t& curr_delta = *out_grad[0]; // dE/dy (already calculated in next␣
↪layer)
    const vec_t& x          = *in_data[0];
    const vec_t& W          = *in_data[1];
    vec_t&       prev_delta = *in_grad[0]; // dE/dx (passed into previous layer)
    vec_t&       dW         = *in_grad[1]; // dE/dW
    vec_t&       db         = *in_grad[2]; // dE/db

    // propagate delta to prev-layer
    for (size_t c = 0; c < x_size_; c++)
        for (size_t r = 0; r < y_size_; r++)
            prev_delta[c] += curr_delta[r] * W[r*x_size_+c];

    // accumulate weight difference
    for (size_t r = 0; r < y_size_; r++)
        for (size_t c = 0; c < x_size_; c++)
            dW[r*x_size_+c] += curr_delta[r] * x[c];

    // accumulate bias difference
    for (size_t r = 0; r < y_size_; r++)
        db[r] += curr_delta[r];
}
```

the `in_data/out_data` are just same as forward_propagation, and `in_grad/out_grad` are its gradient. Order of gradient values are same as `in_data/out_data`.

> Note: Gradient of weight/bias are collected over mini-batch and zero-cleared automatically, so you can't use assignment operator to these elements (layer will forget previous training data in mini-batch!). like this example, use `operator +=` instead. Gradient of data (`prev_delta` in the example) may already have meaningful values if two or more layers share this data, so you can't overwrite this value too.

Verify backward caluculation

It is always a good idea to check if your backward implementation is correct. `network` class provides `gradient_check` method for this purpose. Let's add following lines to test/test_network.h and execute test.

```
TEST(network, gradient_check_fully_connected) {
    network<sequential> net;
    net << fully_connected(2, 3)
        << fully_connected(3, 2);

    std::vector<tensor_t> in{ tensor_t{ 1, { 0.5, 1.0 } } };
```

```
    std::vector<std::vector<label_t>> t = { std::vector<label_t>(1, {1}) };

    EXPECT_TRUE(net.gradient_check<mse>(in, t, 1e-4, GRAD_CHECK_ALL));
}
```

Congratulations! Now you can use this new class as a tiny-dnn layer.

CHAPTER 2

External Links

Here is the list of apps/papers using tiny-dnn. I'm willing to update this list if your software use tiny-dnn. Please contact me at nyanpn (at) gmail (dot) com.

Applications

- zhangqianhui/CnnForAndroid - A Vehicle Recognition Project using Convolutional Neural Network(CNN) in Android platform
- edgarriba/opencv_contrib (in progress) - A new opencv's dnn module which use tiny-dnn as its backend

Papers

- S.S.Sarwar, S.Venkataramani, A.Raghunathan, and K.Roy, Multiplier-less Artificial Neurons Exploiting Error Resiliency for Energy-Efficient Neural Computing
- A.Viebke, Accelerated Deep Learning using Intel Xeon Phi
- Y.Xu, M.Berger, Q.Qian and E.O.Tuguldur, DAInamite - Team Description 2016
- K.Kim, H.Woo, Y.Han, K.Cho, H.Moon, D.Han, Diagnosis-Prescription System for Plant Disease using Mobile Device (Written in Korean)