Tiny-DNN

2019 - 2020

Ando Ki, Ph.D.
adki@future-ds.com

Table of contents

Neural Net in C/C++

- Tiny-Dnn: https://github.com/tiny-dnn/tiny-dnn
  - Tiny-Dnn is a header only, dependency free deep learning library written in C++.
  - BSD 3-Clause license
- Darknet: https://pjreddie.com/darknet/
  - Darknet is an open source neural network framework written in C and CUDA.
  - YOLO license – do whatever you want with it; do not emailing me about it.
- FANN: http://leenissen.dk/fann/wp/
  - Fast Artificial Neural Network Library is a free open source neural network library, which implements multilayer artificial neural networks in C with support for both fully connected and sparsely connected networks.
  - GNU LGPL v2.1
- OpenNN: http://www.opennn.net/
  - Open Neural Networks libaray is an open source class library written in C++ programming language which implements neural networks, a main area of machine learning research.
  - GNU LGPL
- Neural Network Library: https://nnabla.org/
  - An open source software to make research, development and implementation of neural network.
  - Apache License
- lwneuralnet: http://lwneuralnet.sourceforge.net/
  - Lightweight backpropagation neural network in C.
  - GNU LGPL v2.0

What is Tiny-Dnn

- **tiny-dnn** is a header only, dependency free deep learning library written in C++ with C++14 features. (by Taiga Nomi)
  - https://github.com/tiny-dnn/tiny-dnn
  - All you need is a C++14 compiler (gcc 4.9+, clang 3.6+ or VS 2015+).
- Features
  - reasonably fast, without GPU
    - with TBB (threading building block) threading and SSE/AVX (advanced vector extensions) vectorization
    - 98.8% accuracy on MNIST in 13 minutes training (@Core i7-3520M)
  - portable & header-only
    - Run anywhere as long as you have a compiler which supports C++14
    - *Just include tiny_dnn.h and write your model in C++. There is nothing to install.*
  - easy to integrate with real applications
    - no output to stdout/stderr
    - a constant throughput (simple parallelization model, no garbage collection)
    - work without throwing an exception
    - can import caffe's model
  - simply implemented
    - be a good library for learning neural networks
  - GitHub : https://github.com/tiny-dnn/tiny-dnn
  - Documentation : http://tiny-dnn.readthedocs.io/en/latest/

Feature comparison

	Prerequisites	Modeling	Training	Execution	GPU Support	Installing	Windows Support	Pre-Trained Model
tiny-dnn	**Nothing**(Optional:TBB, OpenMP)	C++	C++	C++	No	Unnecessary	Yes	Yes(via caffe-converter)
Caffe	BLAS,Boost,protobuf,glog,gflags,hdf5,(Optional:CUDA,OpenCV,lmdb,leveldbetc)	Config File	C++, Python	C++, Python	Yes	Necessary	Yes	Yes
Theano	Numpy,Scipy,BLAS,(optional:nose,Sphinx,CUDAetc)	Python Code	Python	Python	Yes	Necessary	Yes	No
TensorFlow	numpy,six,protobuf,(optional:CUDA,Bazel)	Python Code	Python	Python, C++	Yes	Necessary	Yes	No[1]
Mxnet	BLAS(optional:CUDA)	C++, Python, R, Julia ...	C++, Python, R, Julia ...	C++, Python, R, Julia ...	Yes	Necessary	Yes	Yes(via caffe-converter)

- Tiny-dnn: (https://github.com/tiny-dnn/tiny-dnn)
- Caffe: (https://github.com/BVLC/caffe)
- Theano: (https://github.com/Theano/Theano)
- TensorFlow: (https://www.tensorflow.org/)
- Mxnet: (http://mxnet.io/)

https://github.com/tiny-dnn/tiny-dnn/wiki/Comparison-with-other-libraries

Installing GCC 4.9 or later

- Tiny-dnn requires C++14 feature.
- Need GNU G++ Version 4.9 or later
- Check current GCC (gcc or g++)

```
$ gcc --version
```

- Install New GCC on Ubuntu if your GCC is not a later than version 4.9

```
$ sudo add-apt-repository ppa:ubuntu-toolchain-r/test
$ sudo apt-get update
$ sudo apt-get install g++-4.9
$ which g++-4.9
  /usr/bin/g++-4.9
```

ubuntu

- User New GCC

```
$ g++-4.9 -std=c++14 test.cpp
```

Installing GCC 4.9 or later

- Install New GCC on CentOS

```
$ cd; mkdir tmp; cd tmp
$ wget https://ftp.gnu.org/gnu/gcc/gcc-4.9.0/gcc-4.9.0.tar.gz
$ tar xzf gcc-4.9.0.tar.gz
$ cd gcc-4.9.0
$ ./contrib/download_prerequisites
$ cd ..
$ mkdir objdir; cd objdir
$ ../gcc-4.9.0/configure --prefix=$HOME/gcc-4.9.0 --program-suffix=-4.9\
                         --enable-languages=c,c++ --disable-multilib
$ make
$ make install
$ ls $HOME/gcc-4.9.0/bin/g++4.9
```

  - Now add following to "$(HOME)/.bashrc"
    - export PATH=$HOME/gcc-4.9.0/bin:$PATH
    - export LD_LIBRARY_PATH=$HOME/gcc-4.9.0/lib:$HOME/gcc.c-4.9.0/lib64:$LD_LIBRARY_PATH
- User New GCC

```
$ g++-4.9 -std=c++14 test.cpp
```


Get Tiny-Dnn

- Go to project directory and make a directory

```
$ cd ~/work
```

- Get a copy of Tiny-Dnn package at '$(PROJECT)/codes'

```
$ git clone https://github.com/tiny-dnn/tiny-dnn.git
```

  - or visit following site and get a copy of it
    - check directory hierarchy and its name; modify if necessary

```
https://github.com/tiny-dnn/tiny-dnn
```

- There is no need to compile or installation, just copy the package where you want.
- Note ${HOME}/work/tiny_dnn/tiny_dnn/config.h' has some macros that user can define or undefined in order to use features.
  - 'CNN_USE_DOUBLE': user 'double' instead of 'float' for 'float_t' type
  - 'DNN_USE_IMAGE_API': define in order to image related API

Test Tiny-DNN

- Go to benchmark directory

```
$ cd ~/work/codes/tiny-dnn/examples/benchmarks
```

- Compile using G++

```
$ g++ main.cpp -I../.. -std=c++14 -pthread -O2
```

- Run the result

```
$ ./a.out
Elapsed time(s): 6.43359
```

- Add following two lines at the end of ${HOME}/.bashrc

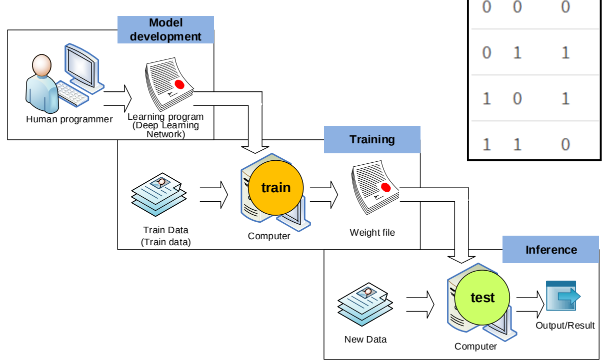
```
export TINYDNN_HOME=${HOME}/work/tiny-dnn
export TINYDNN_ROOT=${HOME}/work/tiny-dnn
```


Getting started: XOR problem

- Use tiny-dnn to solve XOR problem.

- The solution consists of two programs.
  - train.cpp: for training, i.e., learning
    - It fits input against output and save trained network into binary file.
  - test.cpp: for inference
    - It reads the trained network binary file and use it to yield result

X	Y	X ⊕ Y
0	0	0
0	1	1
1	0	1
1	1	0

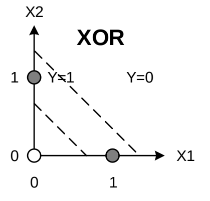


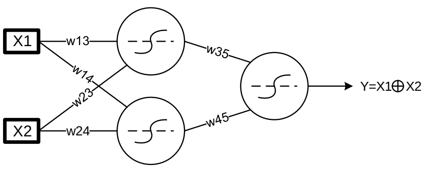
http://linerocks.blogspot.kr/2017/02/solving-xor-problem-using-tiny-dnn_89.html

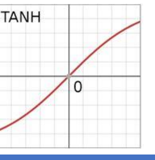
Design MLP

- As XOR problem cannot be solved using linear classifier, it uses MLP (Multi-Layer Perceptron) using fully-connected layers.
- Why single hidden layer?
- Why two nodes in the hidden layer?
- Why tanh (hyperbolic tangent) as activation function?

X	Y	X ⊕ Y
0	0	0
0	1	1
1	0	1
1	1	0







train.cpp for training

$(PROJECT) stands for '${HOME}/work'.

$(PROJECT)/codes/tiny-dnn-projects/xor-two

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

using namespace std;
using namespace tiny_dnn;

int main(int argc, char *argv[]) {
    network<sequential> net("XOR net");
    net << layers::fc(2,2) << activation::tanh()
        << layers::fc(2,1) << activation::tanh();

    vector<vec_t> input_data  { {0,0}, {0,1}, {1,0}, {1,1} };
    vector<vec_t> desired_out { { 0}, { 1}, { 1}, { 0} };

    size_t        batch_size = 1;
    size_t        epochs     = 3000; // num of iteration
    gradient_descent opt; // optimizer algorithm

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

    net.save("xor_net");
    net.save("xor_net_json", content_type::weights_and_model, file_format::json);

    return 0;
}
```

Tiny-Dnn header

Building network

Training data set

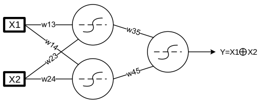
Expected result

Number of iteration

Training

Save network weights

Save network weights & model in JavaScript Object Notation



test.cpp for inference

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

using namespace std;
using namespace tiny_dnn;

int main(int argc, char *argv[]) {

    network<sequential> net("XOR");
    net.load("xor_net");

    vec_t in = {(float_t)atof(argv[1]), (float_t)atof(argv[2])};

    vec_t result = net.predict(in);

    cout << result.at(0) << " ==> "
       << ((result.at(0)>0.05) ? "1" :"0")
       << endl;

    return 0;
}
```

Tiny-Dnn header

Load weight

Get testing data

Inference

Print results

```
//tiny-dnn/config.h
#ifdef CNN_USE_DOUBLE
typedef double float_t;
#else
typedef float float_t;
#endif
```

```
//tiny-dnn/util/util.h
typedef std::vector<float_t, aligned_allocator<float_t, 64>> vec_t;
```


Compilation and running

```
$ g++-4.9 –o train train.cpp –I${TINYDNN_HOME} -std=c++14 -pthread

$ g++-4.9 –o test test.cpp –I${TINYDNN_HOME} -std=c++14 -pthread

$ ./train -e 3000
   mse: 0.00690048

$ ./test 0 0
0.0053472 ==> 0

$ ./test 0 1
0.941501 ==> 1

$ ./test 1 0
0.941791 ==> 1

$ ./test 1 1
0.00784345 ==> 0
```

X	Y	X ⊕ Y
0	0	0
0	1	1
1	0	1
1	1	0

It will be 'g++' instead of 'g++-4.9' if your GCC is a higher than 4.9'.

Note that Raspberry Pi stops when compiling 'test.cpp'. To deal with it increase swap size.

Need Pthread

User C++14 feature

Where Tiny-Dnn locates

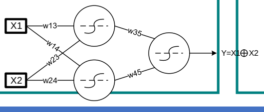
Source program

Executable program

This generates 'xor_net' and 'xor_net_json' files

Model and weight in JSON format

```
{
    "nodes": [
        {
            "type": "fully_connected",
            "in_size": 2,
            "out_size": 2,
            "has_bias": true
        },
        {
            "type": "tanh",
            "in_size": {
                "width": 2,
                "height": 1,
                "depth": 1
            }
        },
        {
            "type": "fully_connected",
            "in_size": 2,
            "out_size": 1,
            "has_bias": true
        },
```

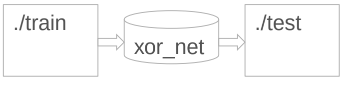



```
        {
            "type": "tanh",
            "in_size": {
                "width": 1,
                "height": 1,
                "depth": 1
            }
        }
    ],
    "value0": {
        "type": "fully_connected",
        "value0": [
            -1.9977405071258545,
            1.0795886516571045,
            -1.8817317485809326,
            1.0510303974151611
        ],
        "value1": [
            0.5363711186981201,
            -1.5420776605606079
        ]
    },
```

```
    "value1": {
        "type": "tanh"
    },
    "value2": {
        "type": "fully_connected",
        "value0": [
            -1.9583592414855957,
            -2.0197112560272217
        ],
        "value1": [
            -0.8775179386138916
        ]
    },
    "value3": {
        "type": "tanh"
    }
}
```


Running XOR example

- This example shows how to use compile Tiny-Dnn program
    - Step 1: go to your project directory
        - [user@host] cd $(PROJECT)/codes/tiny-dnn-projects/xor-two
    - Step 2: see the codes
    - Step 3: compile
        - [user@host] make
    - Step 4: training
        - [user@host] make learn
    - Step 5: see how network has been built
        - [user@host] vi xor_net_json
    - Step 6: inference
        - [user@host] make inference



It takes a long time since Tiny-DNN is compiled.

$(PROJECT) stands for '${HOME}/work'.

```
[user@host] cd $(PROJECT)/codes/tiny-dnn-projects/xor-two
[user@host] make
[user@host] make learn
[user@host] make inference
```

More things

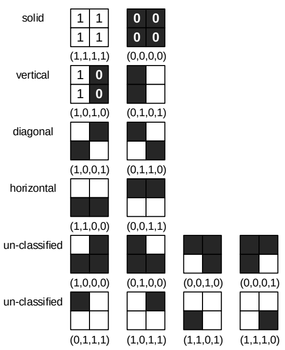
- Let's look what happens when the XOR network is trained not sufficiently.
  - Say, use '500' or '1000' epochs (i.e., iteration) while training
    - Need to check loss value.
      - Training loss
- Let's try to use more neurons.
  - Say, three neurons on the hidden lay.
- Let's try to use other activation functions.
  - Sigmoid: 0 ~ 1 (expecting error, since non negative value)
  - ReLU: 0 ~ linear (what affects with non-negative)
  - Leaky ReLU

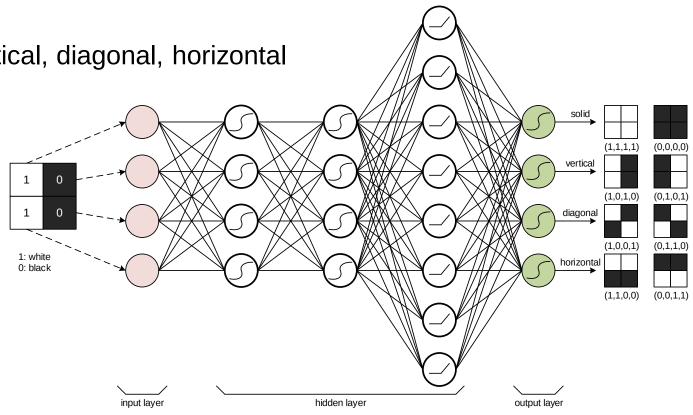
Table of contents

Four pixels project

- Four pixels classification
  - Each pixel can be 0 (black) or 1 (white)
  - There are 16 cases
  - It is trained only 8 cases: solid, vertical, diagonal, horizontal

$(PROJECT)/codes/tiny-dnn-projects/four-pixels





Inspired by Brandon Rohrer's YouTube "How Deep Neural Network Work": https://www.youtube.com/watch?v=ILsA4nyG7I0

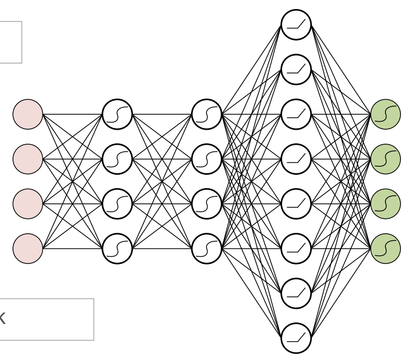
Four pixels: train.cpp

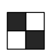
```
… …
#include "tiny_dnn/tiny_dnn.h"

using namespace tiny_dnn;
using namespace std;

int main(int argc, char *argv[]) {
   network<sequential> net("Four Pixels");
   net << layers::fc(4,4) << activation::sigmoid()
       << layers::fc(4,4) << activation::sigmoid()
       << layers::fc(4,8) << activation::relu()
       << layers::fc(8,4) << activation::tanh();

 vector<vec_t> input_data  { {0,0, 0,0}, // solid
                        {1,1, 1,1}, // solid
                        {1,0, 1,0}, // vertical
                        {0,1, 0,1}, // vertical
                        {1,0, 0,1}, // diagonal
                        {0,1, 1,0}, // diagonal
                        {1,1, 0,0}, // horizontal
                        {0,0, 1,1}};// horizontal
```

Tiny-Dnn header

Building network

Training data set




(0,1,1,0)

Four pixels: train.cpp

```
vector<vec_t> desired_out { {1,0,0,0}, // solid
                            {1,0,0,0}, // solid
                            {0,1,0,0}, // vertical
                            {0,1,0,0}, // vertical
                            {0,0,1,0}, // diagonal
                            {0,0,1,0}, // diagonal
                            {0,0,0,1}, // horizontal
                            {0,0,0,1}};// horizontal
  size_t         batch_size = 1;
  size_t         epochs     = 10000; // num of iteration
  gradient_descent opt; // optimizer algorithm

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

  // Figure out how good the net is.
  float_t loss = net.get_loss<mse>(input_data, desired_out);
  cout << "mse : " << loss << endl;

  // Save our result as 'pixel_net'.
  net.save("pixel_net");
  net.save("pixel_net_json", content_type::weights_and_model, file_format::json);
  return 0;
}
```

Expected result

Training

Save network weights

Save network weights & model in JavaScript Object Notation

Four pixels: test.cpp

```
//#include <cstdlib>
#include "tiny_dnn/tiny_dnn.h"
using namespace std;
using namespace tiny_dnn;

int main(int argc, char *argv[]) {

  network<sequential> net("Four Pixels");
  net.load("pixel_net");

  vec_t in = {(float)atof(argv[1]),
              (float)atof(argv[2]),
              (float)atof(argv[3]),
              (float)atof(argv[4])};

  vec_t result = net.predict(in);

  string val[] = { "solid", "vertical", "diagonal", "horizontal"};
  for (int i=0; i<4; i++) {
     cout << "[" << val[i] << "] " << result[i] << endl;
  }
  return 0;
}
```

Tiny-Dnn header

Load parameters

Test data from command arguments

Run inference

Print result

Four pixels: Compilation and running

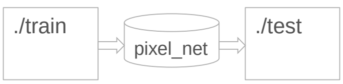
$(PROJECT)/codes/tiny-dnn-projects/four-pixels

```
$ make
g++ -c train.cpp –I${TINYDNN_HOME} -std=c++14 -pthread -O2
g++ -o train train.o  -lpthread 2>&1 | tee -a compile.log.train
g++ -c test.cpp –I${TIYDNN_HOME} -std=c++14 -pthread -O2
g++ -o test test.o  -lpthread
$ make learn
./train -e 15000
mse : 0.388477
$ make inference
./test 0 0 0 0
[solid] 0.91082 [vertical] -0.0607263 [diagonal] 0.0153693 [horizontal] 0.0228204
./test 1 1 1 1
[solid] 0.868702 [vertical] -0.00437486 [diagonal] 0.0354705 [horizontal] 0.0180183
./test 1 0 1 0
[solid] -0.00548971 [vertical] 0.51708 [diagonal] 0.290092 [horizontal] 0.169049
...
[solid] 0.135655 [vertical] 0.133207 [diagonal] -0.117472 [horizontal] 0.680154
./test 1 0 0 1
[solid] 0.0423046 [vertical] 0.365283 [diagonal] 0.504739 [horizontal] -0.0512018
sum: 0.861126
./test 0 1 1 0
[solid] 0.131284 [vertical] 0.0592648 [diagonal] 0.739264 [horizontal] 0.0594117
$
```

This generates 'pixel_net' file

Running four pixels example

- This example shows how to use compile Tiny-Dnn program
  - Step 1: go to your project directory
    - [user@host] cd $(PROJECT)/codes/tiny-dnn-projects/four-pixels
  - Step 2: see the codes
  - Step 3: compile
    - [user@host] make
  - Step 4: training
    - [user@host] make learn
  - Step 5: see how network has been built
    - [user@host] vi pixel_net_json
  - Step 6: inference
    - [user@host] make inference




```
[user@host] cd $(PROJECT)/codes/tiny-dnn-projects/four-pixels
[user@host] make
[user@host] make learn
[user@host] make inference
```

Four pixels project revisited

- Four pixels classification
  - Each pixel can be 0 (black) or 1 (white)
  - There are 16 cases
  - It is trained only 8 cases: solid, vertical, diagonal, horizontal
  - Let assign label to the result, and use ‘train’ instead of ‘fit’ for training function

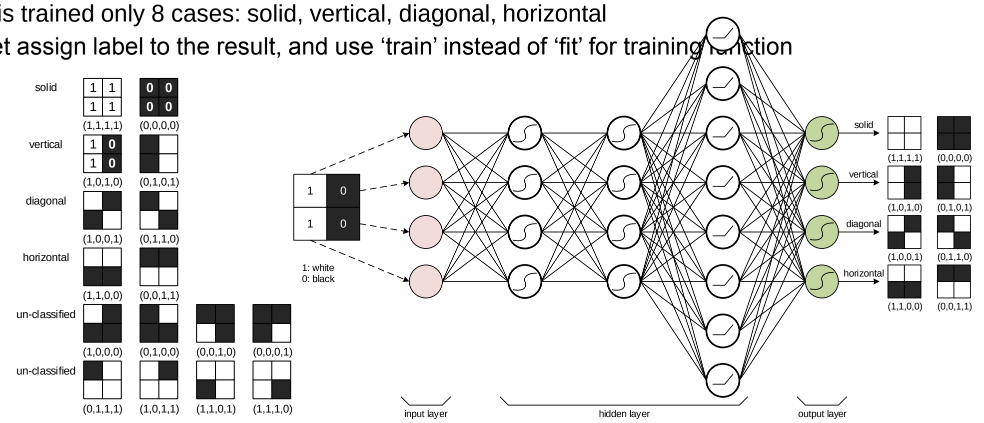
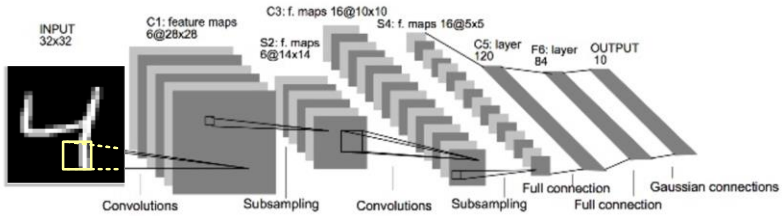


Table of contents

LeNet project with MNIST

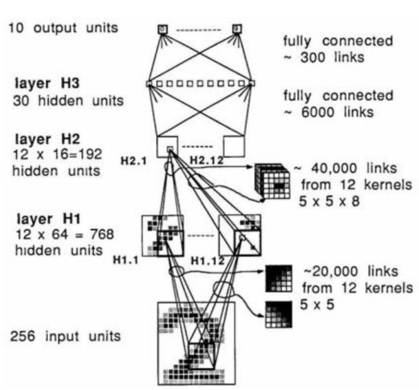
- MNIST (Modified National Institute of Standards and Technology) database
  - Handwritten digits image written by high-school student and employees of US Census Bureau
  - 28x28 pixels Black and White – 28x28 centered, [0,255] value
  - 10 classes (0, 1, …, 9)
  - 60K training image, 10K testing image
- LeNet-5: Yann LeCun, 1998
  - 28x28 pixels + 2-pixels boards ➔ 32x32 pixels; [0,255] ➔ [-1.0, 1.0] scaling
  - 0.95% error

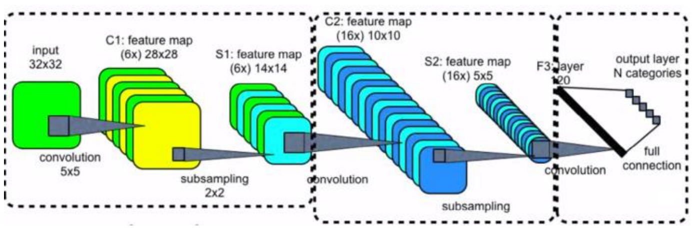
$(PROJECT)/codes/tiny-dnn-projects/lenet-mnist

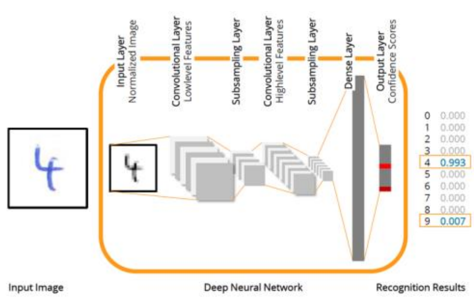


https://github.com/tiny-dnn/tiny-dnn/tree/master/e›
https://github.com/tiny-dnn/tiny-dnn/wiki/MNIST-Example

LeNet project with MNIST

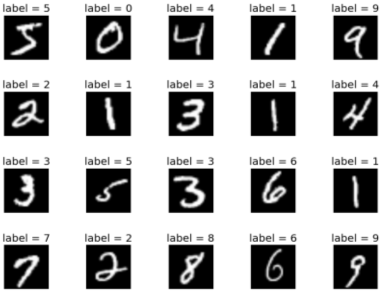






LeNet project with MNIST

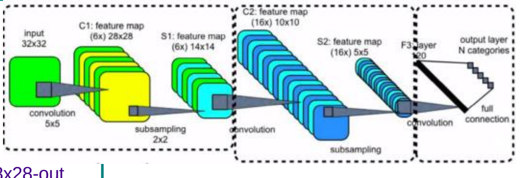
- Data set from ‘http://yann.lecun.com/exdb/mnist/’
  - see ‘dataset’ directory
  - to check run as follows
    - $ python mnist_save_png.py
- Training data
  - train-images-idx3-ubyte: 55K training set images,
  - train-labels-idx1-ubyte: training set labels
- Testing data
  - t10k-images-idx3-ubyte: 10K test set images
  - t10k-labels-idx1-ubyte: test set labels



29

LeNet: train.cpp

```
// construct nets
//
// C : convolution
// S : sub-sampling
// F : fully connected

nn << conv(32, 32, 5, 1, 6,                          // C1, 1@32x32-in, 6@28x28-out
        padding::valid, true, 1, 1, backend_type)
   << tanh()
   << ave_pool(28, 28, 6, 2)                         // S2, 6@28x28-in, 6@14x14-out
   << tanh()
   << conv(14, 14, 5, 6, 16,                         // C3, 6@14x14-in, 16@10x10-out
        connection_table(tbl, 6, 16),
        padding::valid, true, 1, 1, backend_type)
   << tanh()
   << ave_pool(10, 10, 16, 2)                        // S4, 16@10x10-in, 16@5x5-out
   << tanh()
   << conv(5, 5, 5, 16, 120,                         // C5, 16@5x5-in, 120@1x1-out
        padding::valid, true, 1, 1, backend_type)
   << tanh()
   << fc(120, 10, true, backend_type)                // F6, 120-in, 10-out
   << tanh();
```



$(PROJECT)/codes/tiny-dnn-projects/lenet-mnist

30

LeNet: Compilation and running

Do not forget to define 'TINYDNN_HOME' environment variable.

$ export TINYDNN_HOME=${HOME}/work/tiny-dnn
$ export TINYDNN_ROOT=${HOME}/work/tiny-dnn

$(PROJECT)/codes/tiny-dnn-projects/lenet-mnist

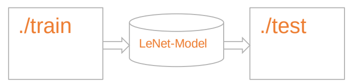
$ make

g++ -c train.cpp –I${*TINYDNN_HOME*} –DCNN_USE_DOUBLE –DDNN_USE_IMAGE_API\
-std=c++14 -pthread -O2
g++ -o train train.o -lpthread 2>&1 | tee -a compile.log.train
g++ -c test.cpp –I${TINYDNN_HOME} –DCNN_USE_DOUBLE –DDNN_USE_IMAGE_API\
-DVERBOSE -std=c++14 -pthread -O2
g++ -o test test.o -lpthread

$ make learn

./train --epochs 30 --data_path dataset
…..

This generates 'LeNet-model' file

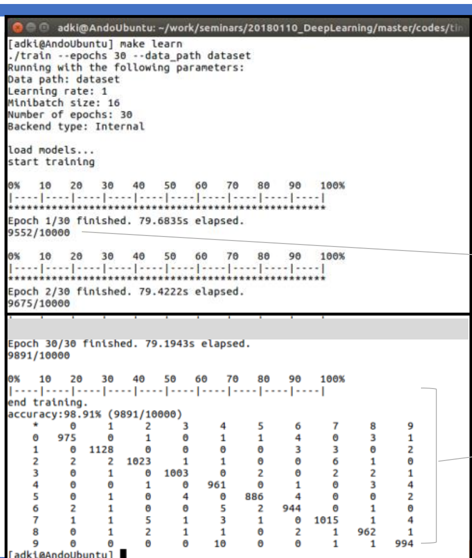


$ make inference

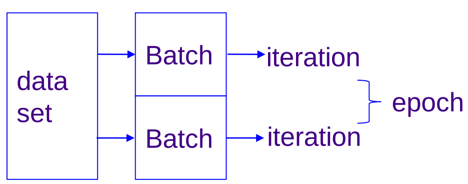
./test **testset/4.bmp**
4,105.18
9,66.6988
7,55.9366

This generates a number of PNG files, which are graphical images of weights and intermediate results.

LeNet: $ make learn



```
adki@AndoUbuntu: ~/work/seminars/20180110_DeepLearning/master/codes/tin
[adki@AndoUbuntu] make learn
./train --epochs 30 --data_path dataset
Running with the following parameters:
Data path: dataset
Learning rate: 1
Minibatch size: 16
Number of epochs: 30
Backend type: Internal

load models...
start training

0%   10   20   30   40   50   60   70   80   90   100%
|----|----|----|----|----|----|----|----|----|----|
***************************************************
Epoch 1/30 finished. 79.6835s elapsed.
9552/10000

0%   10   20   30   40   50   60   70   80   90   100%
|----|----|----|----|----|----|----|----|----|----|
***************************************************
Epoch 2/30 finished. 79.4222s elapsed.
9675/10000

Epoch 30/30 finished. 79.1943s elapsed.
9891/10000

0%   10   20   30   40   50   60   70   80   90   100%
|----|----|----|----|----|----|----|----|----|----|
end training.
accuracy:98.91% (9891/10000)
    *     0     1     2     3     4     5     6     7     8     9
    0   975     0     1     0     1     1     4     0     3     1
    1     0  1128     0     0     0     0     3     3     0     2
    2     2     2  1023     1     1     0     0     6     1     0
    3     0     1     0  1003     0     2     0     2     2     1
    4     0     0     1     0   961     0     1     0     3     4
    5     0     1     0     4     0   886     4     0     0     2
    6     2     1     0     0     5     2   944     0     1     0
    7     1     1     5     1     3     1     0  1015     1     4
    8     0     1     2     1     1     0     2     1   962     1
    9     0     0     0     0    10     0     0     1     1   994
[adki@AndoUbuntu]
```



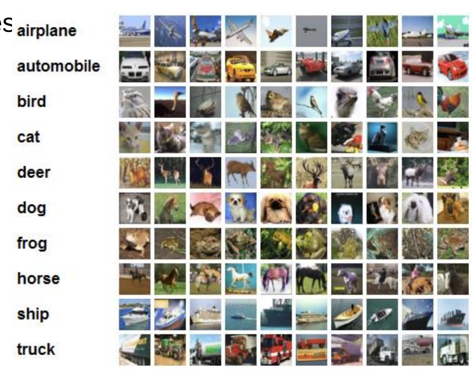
num_sussess / num_total

Testing result

Table of contents

CIFAR-10 project

- CIFA (Canadian Institute For Advanced Research) database
- CIFA-10
  - Object classification of 10 classes (airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck.)
  - image dataset consists of 60,000 (32x32-pixes h 6,000 images per class.
  - 32x32 pixels (color)
  - 50,000 training
  - 10,000 testing (evaluation)



Cifar-10: Compilation and running

$(PROJECT)/codes/tiny-dnn-projects/cifar-10

```
$ make
g++ -c train.cpp –I${TINYDNN_HOME} –DCNN_USE_DOUBLE –DDNN_USE_IMAGE_API\
                 -std=c++14 -pthread -O2
g++ -o train train.o  -lpthread 2>&1 | tee -a compile.log.train
g++ -c test.cpp –I${TINYDNN_HOME} –DCNN_USE_DOUBLE –DDNN_USE_IMAGE_API\
                -DVERBOSE -std=c++14 -pthread -O2
g++ -o test test.o  -lpthread

$ make dataset
$ make testset

$ make learn
…..

$ make inference
…
```

This generates 'cifar-weight' file

This generates a number of PNG files, whi ch are graphical images of weights and int ermediate results.

References

- https://github.com/tiny-dnn/tiny-dnn
- http://tiny-dnn.readthedocs.io

㈜퓨쳐디자인시스템
34051 대전광역시 유성구 문지로 193, KAIST 문지캠퍼스, F723호
(042) 864-0211~0212 / contact@future-ds.com / www.future-ds.com

Future Design Systems, Inc.
Faculty Wing F723, KAIST Munji Campus, 193 Munji-ro, Yuseong-gu, Daejeon 34051, Korea
+82-042-864-0211~0212 / contact@future-ds.com / www.future-ds.com

FUTURE
Design Systems