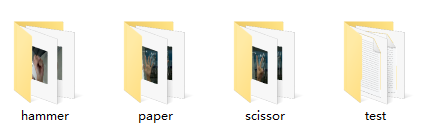
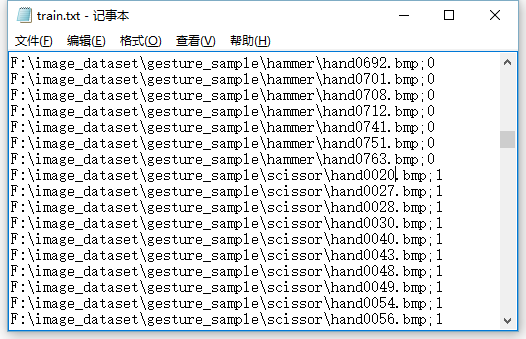
# 使用Tiny-dnn训练和分类

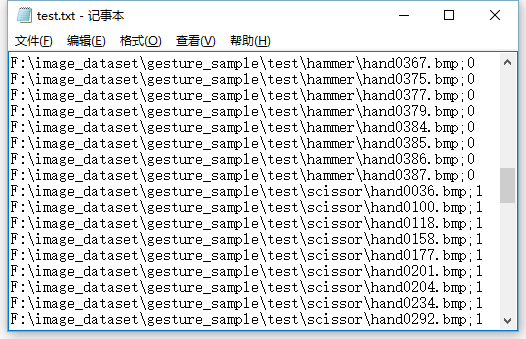
Tiny-dnn自带的例子使用的是MNIST手写字符数据库，下面演示使用自己的图像进行训练和测试，关于卷积神经网络的原理及Tiny-dnn的代码理解暂且放一边，只给出以下训练测试实例。

准备训练和测试数据：

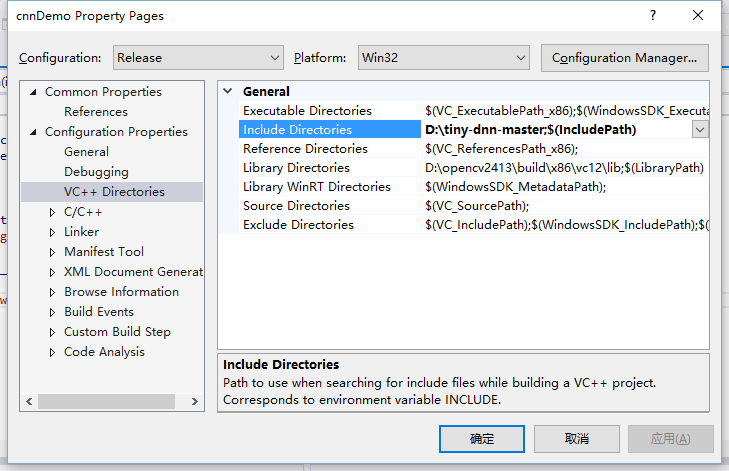


将各训练数据和测试数据路径以文本的形式保存，并标注其分类标签：

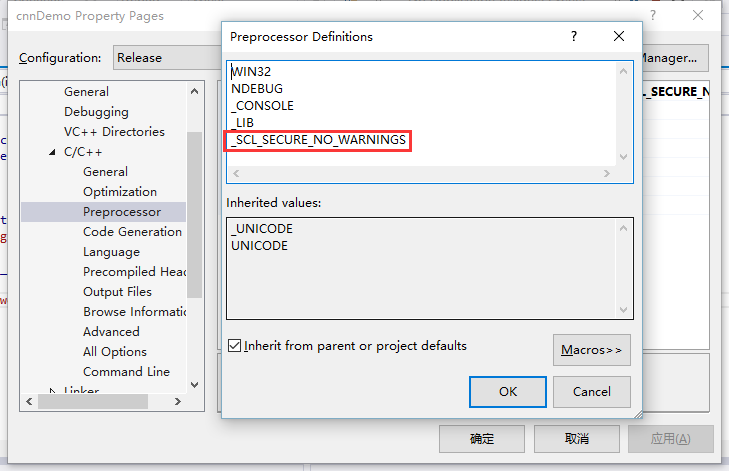




创建工程，添加tiny-dnn-master的路径到包含目录：



添加预处理命令\_SCL\_SECURE\_NO\_WARNINGS：



添加以下代码：

#include "stdafx.h"

#include <opencv2\opencv.hpp>

#include <tiny\_dnn\tiny\_dnn.h>

#include <iostream>

#include <fstream>

#include <memory>

using namespace *tiny\_dnn*;

using namespace *tiny\_dnn*::activation;

using namespace *tiny\_dnn*::*layers*;

using namespace *std*;

*std*::*vector*<*label\_t*> train\_labels, test\_labels;

*std*::*vector*<*vec\_t*> train\_images, test\_images;

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;

*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);

}

void readTrainData()

{

*fstream* file("train.txt", *fstream*::*in*);

*string* line;

while (*getline*(file, line))

{

*string* path, label;

char sep = ';';

*stringstream* ss(line);

*getline*(ss, path, sep);

*getline*(ss, label);

convert\_image(path, 1.0, 32, 32, train\_images);

train\_labels.*push\_back*(*atoi*(label.*c\_str*()));

}

file.*close*();

}

void readTestData()

{

*fstream* file("test.txt", *fstream*::*in*);

*string* line;

while (*getline*(file, line))

{

*string* path, label;

char sep = ';';

*stringstream* ss(line);

*getline*(ss, path, sep);

*getline*(ss, label);

convert\_image(path, 1.0, 32, 32, test\_images);

test\_labels.*push\_back*(*atoi*(label.*c\_str*()));

}

file.*close*();

}

int *main*(int argc, char\*\* argv) {

network<sequential> nn;

adagrad optimizer;

// connection table [Y.Lecun, 1998 Table.1]

#define O true

#define X false

static const bool connection[] = {

O, X, X, X, O, O, O, X, X, O, O, O, O, X, O, O,

O, O, X, X, X, O, O, O, X, X, O, O, O, O, X, O,

O, O, O, X, X, X, O, O, O, X, X, O, X, O, O, O,

X, O, O, O, X, X, O, O, O, O, X, X, O, X, O, O,

X, X, O, O, O, X, X, O, O, O, O, X, O, O, X, O,

X, X, X, O, O, O, X, X, O, O, O, O, X, O, O, O

};

#undef O

#undef X

nn << convolutional\_layer<tan\_h>(32, 32, 5, 1, 6) // 32x32 in, 5x5 kernel, 1-6 fmaps conv

<< average\_pooling\_layer<tan\_h>(28, 28, 6, 2) // 28x28 in, 6 fmaps, 2x2 subsampling

<< convolutional\_layer<tan\_h>(14, 14, 5, 6, 16,

connection\_table(connection, 6, 16)) // with connection-table

<< average\_pooling\_layer<tan\_h>(10, 10, 16, 2)

<< convolutional\_layer<tan\_h>(5, 5, 5, 16, 120)

<< fully\_connected\_layer<tan\_h>(120, 10);

*std*::*cout* << "load models..." << *std*::*endl*;

readTrainData();

readTestData();

*std*::*cout* << "start learning" << *std*::*endl*;

progress\_display disp(train\_images.*size*());

*timer* t;

int minibatch\_size = 10;

optimizer.*alpha* \*= *std*::*sqrt*(minibatch\_size);

// create callback

auto on\_enumerate\_epoch = [&](){

*std*::*cout* << t.elapsed() << "s elapsed." << *std*::*endl*;

*tiny\_dnn*::*result* res = nn.*test*(test\_images, test\_labels);

*std*::*cout* << res.num\_success << "/" << res.num\_total << *std*::*endl*;

disp.*restart*(train\_images.*size*());

t.*restart*();

};

auto on\_enumerate\_minibatch = [&](){

disp += minibatch\_size;

};

// training

nn.*train*<mse>(optimizer, train\_images, train\_labels, minibatch\_size, 20, on\_enumerate\_minibatch, on\_enumerate\_epoch);

*std*::*cout* << "end training." << *std*::*endl*;

// test and show results

nn.*test*(test\_images, test\_labels).print\_detail(*std*::*cout*);

// save networks

*std*::*ofstream* ofs("LeNet-weights");

ofs << nn;

*cin*.*get*();

}

运行结果：

