登录 | 注册

网络资源是无限的

目录视图 摘要视图 RSS 订阅

个人资料

fengbingchun

访问： 2252588次
积分： 25003
等级： BLOC 7
排名： 第202名

原创： 341篇 转载： 144篇
译文： 0篇 评论： 1434条

CSDN学院招募微信小程序讲师啦 程序员简历优化指南！ 【观点】移动原生App开发 PK HTML 5开发 云端应用征文大赛，秀绝招，赢无人机！

卷积神经网络(CNN)的简单实现(MNIST)

2016-03-06 19:20 7538人阅读 评论(24) 收藏 举报

本文章已收录于： 深度学习知识库

分类： Caffe（19） Deep Learning（8） Neural Network（12）

版权声明：本文为博主原创文章，未经博主允许不得转载。

卷积神经网络(CNN)的基础介绍见http://blog.csdn.net/fengbingchun/article/details/50529500，这里主要以代码实现为主。

CNN是一个多层的神经网络，每层由多个二维平面组成，而每个平面由多个独立神经元组成。

以MNIST作为数据库，仿照LeNet-5和tiny-cnn(http://blog.csdn.net/fengbingchun/article/details/50573841) 设计一个简单的7层CNN结构如下：

输入层Input：神经元数量32*32=1024；

C1层：卷积窗大小5*5，输出特征图数量6，卷积窗种类6，输出特征图大小28*28，可训练参数(权值+阈值(偏置))5*5*6+6=150+6，神经元数量28*28*6=4704；

S2层：卷积窗大小2*2，输出下采样图数量6，卷积窗种类6，输出下采样图大小14*14，可训练参数1*6+6=6+6，神经元数量14*14*6=1176；

C3层：卷积窗大小5*5，输出特征图数量16，卷积窗种类6*16=96，输出特征图大小10*10，可训练参数5*5*(6*16)+16=2400+16，神经元数量10*10*16=1600；

S4层：卷积窗大小2*2，输出下采样图数量16，卷积窗种类16，输出下采样图大小5*5，可训练参数1*16+16=16+16，神经元数量5*5*16=400；

C5层：卷积窗大小5*5，输出特征图数量120，卷积窗种类16*120=1920，输出特征图大小1*1，可训练参数5*5*(16*120)+120=48000+120，神经元数量1*1*120=120；

输出层Output：卷积窗大小1*1，输出特征图数量10，卷积窗种类120*10=1200，输出特征图大小1*1，可训练参数1*(120*10)+10=1200+10，神经元数量1*1*10=10。

下面对实现执行过程进行描述说明：

1. 从MNIST数据库中分别获取训练样本和测试样本数据：

(1)、原有MNIST库中图像大小为28*28，这里缩放为32*32，数据值范围为[-1,1]，扩充值均取-1；总共60000个32*32训练样本，10000个32*32测试样本；

(2)、输出层有10个输出节点，在训练阶段，对应位置的节点值设为0.8，其它节点设为-0.8.

2. 初始化权值和阈值(偏置)：权值就是卷积图像，每一个特征图上的神经元共享相同的权值和阈值，特征图的数量等于阈值的个数

HTML (3)
Image Recognition (8)
Image Processing (18)
Image Registration (13)
ImageMagick (3)
Java (5)
Linux (20)
Log (2)
Makefile (2)
Mathematical Knowledge (6)
Multi-thread (4)
Matlab (33)
MFC (8)
MinGW (3)
Mac (1)
Neural Network (13)
OCR (9)
Office (2)
OpenCL (2)
OpenSSL (7)
OpenCV (86)
OpenGL (2)
OpenGL ES (3)
OpenMP (3)
Photoshop (1)
Python (4)
Qt (1)
SIMD (14)
Software Development (4)
System architecture (2)
Skia (1)
SVN (1)
Software Testing (4)
Shell (2)
Socket (3)
Target Detection (2)
Target Tracking (2)
VC6 (6)
VS2008 (16)
VS2010 (4)
VS2013 (3)
vigra (2)
VLC (5)
VLFeat (1)
wxWidgets (1)
Watermark (4)
Windows7 (6)
Windows Core Programming (9)
XML (2)

Free Codes

pudn
freecode
Peter's Functions
CodeProject
SourceCodeOnline
Computer Vision Source Code
Codesoso
Digital Watermarking
SourceForge
HackChina
oschina

(1)、权值采用uniform rand的方法初始化；

(2)、阈值均初始化为0.

3.　前向传播：根据权值和阈值，主要计算每层神经元的值

(1)、输入层：每次输入一个32*32数据。

(2)、C1层：分别用每一个5*5的卷积图像去乘以32*32的图像，获得一个28*28的图像，即对应位置相加再求和，stride长度为1；一共6个5*5的卷积图像，然后对每一个神经元加上一个阈值，最后再通过tanh激活函数对每一神经元进行运算得到最终每一个神经元的结果。

(3)、S2层：对C1中6个28*28的特征图生成6个14*14的下采样图，相邻四个神经元分别进行相加求和，然后乘以一个权值，再求均值即除以4，然后再加上一个阈值，最后再通过tanh激活函数对每一神经元进行运算得到最终每一个神经元的结果。

(4)、C3层：由S2中的6个14*14下采样图生成16个10*10特征图，对于生成的每一个10*10的特征图，是由6个5*5的卷积图像去乘以6个14*14的下采样图，然后对应位置相加求和，然后对每一个神经元加上一个阈值，最后再通过tanh激活函数对每一神经元进行运算得到最终每一个神经元的结果。

(5)、S4层：由C3中16个10*10的特征图生成16个5*5下采样图，相邻四个神经元分别进行相加求和，然后乘以一个权值，再求均值即除以4，然后再加上一个阈值，最后再通过tanh激活函数对每一神经元进行运算得到最终每一个神经元的结果。

(6)、C5层：由S4中16个5*5下采样图生成120个1*1特征图，对于生成的每一个1*1的特征图，是由16个5*5的卷积图像去乘以16个5*5的下采用图，然后相加求和，然后对每一个神经元加上一个阈值，最后再通过tanh激活函数对每一神经元进行运算得到最终每一个神经元的结果。

(7)、输出层：即全连接层，输出层中的每一个神经元均是由C5层中的120个神经元乘以相对应的权值，然后相加求和；然后对每一个神经元加上一个阈值，最后再通过tanh激活函数对每一神经元进行运算得到最终每一个神经元的结果。

4.　反向传播：主要计算每层神经元、权值和阈值的误差，以用来更新权值和阈值

(1)、输出层：计算输出层神经元误差；通过mse损失函数的导数函数和tanh激活函数的导数函数来计算输出层神经元误差。

(2)、C5层：计算C5层神经元误差、输出层权值误差、输出层阈值误差；通过输出层神经元误差乘以输出层权值，求和，结果再乘以C5层神经元的tanh激活函数的导数，获得C5层每一个神经元误差；通过输出层神经元误差乘以C5层神经元获得输出层权值误差；输出层误差即为输出层阈值误差。

(3)、S4层：计算S4层神经元误差、C5层权值误差、C5层阈值误差；通过C5层权值乘以C5层神经元误差，求和，结果再乘以S4层神经元的tanh激活函数的导数，获得S4层每一个神经元误差；通过S4层神经元乘以C5层神经元误差，求和，获得C5层权值误差；C5层神经元误差即为C5层阈值误差。

(4)、C3层：计算C3层神经元误差、S4层权值误差、S4层阈值误差；

(5)、S2层：计算S2层神经元误差、C3层权值误差、C3层阈值误差；

(6)、C1层：计算C1层神经元误差、S2层权值误差、S2层阈值误差；

(7)、输入层：计算C1层权值误差、C1层阈值误差.

代码文件：

CNN.hpp：

[cpp]

```cpp
#ifndef _CNN_HPP_
#define _CNN_HPP_

#include <vector>
#include <unordered_map>

namespace ANN {

```

关闭

- libsvm
- joys99
- CodeForge
- cvchina
- tesseract-ocr
- sift
- TiRG
- imgSeek
- OpenSURF

Friendly Link

- OpenCL
- Python
- poesia-filter
- TortoiseSVN
- imgSeek
- Notepad
- Beyond Compare
- CMake
- VIGRA
- CodeGuru
- vchome
- aforgenet
- CVLAB
- Doxygen
- Coursera
- OpenMP

Technical Forum

- Matlab China
- OpenCV China
- The CImg Library
- Open Computer Vision Library
- CxImage
- ImageMagick
- ImageMagick China
- OpenCV_China
- Subversion China


```
#define width_image_input_CNN       32 //归一化图像宽
#define height_image_input_CNN      32 //归一化图像高
#define width_image_C1_CNN      28
#define height_image_C1_CNN     28
#define width_image_S2_CNN      14
#define height_image_S2_CNN     14
#define width_image_C3_CNN      10
#define height_image_C3_CNN     10
#define width_image_S4_CNN      5
#define height_image_S4_CNN     5
#define width_image_C5_CNN      1
#define height_image_C5_CNN     1
#define width_image_output_CNN      1
#define height_image_output_CNN     1

#define width_kernel_conv_CNN       5 //卷积核大小
#define height_kernel_conv_CNN      5
#define width_kernel_pooling_CNN    2
#define height_kernel_pooling_CNN   2
#define size_pooling_CNN        2

#define num_map_input_CNN       1 //输入层map个数
#define num_map_C1_CNN          6 //C1层map个数
#define num_map_S2_CNN          6 //S2层map个数
#define num_map_C3_CNN          16 //C3层map个数
#define num_map_S4_CNN          16 //S4层map个数
#define num_map_C5_CNN          120 //C5层map个数
#define num_map_output_CNN      10 //输出层map个数

#define num_patterns_train_CNN      60000 //训练模式对数(总数)
#define num_patterns_test_CNN       10000 //测试模式对数(总数)
#define num_epochs_CNN          100 //最大迭代次数
#define accuracy_rate_CNN       0.985 //要求达到的准确率
#define learning_rate_CNN       0.01 //学习率
#define eps_CNN             1e-8

#define len_weight_C1_CNN       150 //C1层权值数，5*5*6*1=150
#define len_bias_C1_CNN         6 //C1层阈值数，6
#define len_weight_S2_CNN       6 //S2层权值数,1*6=6
#define len_bias_S2_CNN         6 //S2层阈值数,6
#define len_weight_C3_CNN       2400 //C3层权值数，5*5*16*6=2400
#define len_bias_C3_CNN         16 //C3层阈值数,16
#define len_weight_S4_CNN       16 //S4层权值数，1*16=16
#define len_bias_S4_CNN         16 //S4层阈值数，16
#define len_weight_C5_CNN       48000 //C5层权值数，5*5*16*120=48000
#define len_bias_C5_CNN         120 //C5层阈值数，120
#define len_weight_output_CNN       1200 //输出层权值数，120*10=1200
#define len_bias_output_CNN     10 //输出层阈值数，10

#define num_neuron_input_CNN        1024 //输入层神经元数，32*32=1024
#define num_neuron_C1_CNN       4704 //C1层神经元数，28*28*6=4704
#define num_neuron_S2_CNN       1176 //S2层神经元数，14*14*6=1176
#define num_neuron_C3_CNN       1600 //C3层神经元数，10*10*16=1600
#define num_neuron_S4_CNN       400 //S4层神经元数，5*5*16=400
#define num_neuron_C5_CNN       120 //C5层神经元数，1*120=120
#define num_neuron_output_CNN       10 //输出层神经元数，1*10=10

class CNN {
public:
    CNN();
    ~CNN();

    void init(); //初始化，分配空间
    bool train(); //训练
    int predict(const unsigned char* data, int width, int height); //预测
    bool readModelFile(const char* name); //读取已训练

protected:
    typedef std::vector<std::pair<int, int> > wi_connections;
    typedef std::vector<std::pair<int, int> > wo_connections;
    typedef std::vector<std::pair<int, int> > io_connections;

    void release(); //释放申请的空间
    bool saveModelFile(const char* name); //将训练好的model保存起来，包括各层的节点数，权值和阈值
    bool initWeightThreshold(); //初始化，产生[-1, 1]之间的随机小数
    bool getSrcData(); //读取MNIST数据
    double test(); //训练完一次计算一次准确率
    double activation_function_tanh(double x); //激活函数:tanh
```

关闭

Technical Blog

邹宇华
深之JohnChen
HUNNISH
周伟明
superdont
carson2005
OpenHero
Netman(Linux)
wqvbjhc
yang_xian521
gnuhpc
gnuhpc
千里8848
CVART
tornadomeet
gotosuc
onezeros
hellogv
abcjennifer
crzy_sparrow

评论排行

Windows7 32位机上，O (120)
tiny-cnn开源库的使用(MI (93)
Ubuntu 14.04 64位机上才 (89)
tesseract-ocr3.02字符识 (63)
Windows7上使用VS201 (47)
tesseract-ocr (42)
图像配准算法 (41)
Windows 7 64位机上Ope (36)
OpenCV中resize函数五种 (34)
小波矩特征提取matlab代 (30)

最新评论

Tesseract-OCR 3.04在Windows
fengbingchun: @ilikede:没有密码，那个commit只是提示是从哪个commit fork过来的，无需管那个

Tesseract-OCR 3.04在Windows
ilikede: 问一下，你第一句中的commit的那个密码，怎么用啊

卷积神经网络(CNN)的简单实现(
fengbingchun: @hugl950123:是需要opencv的支持，你在本地opencv的环境配好了吗，配好了就应该没...

卷积神经网络(CNN)的简单实现(
hugl950123: @fengbingchun:博主请问一下，test_CNN_predict()函数是不是需要open...

卷积神经网络(CNN)的简单实现(
hugl950123: @fengbingchun:博主请问一下，test_CNN_predict()函数是不是需要open...

卷积神经网络(CNN)的简单实现(
hugl950123: @fengbingchun:谢谢，能够成功运行了现在

卷积神经网络(CNN)的简单实现(
fengbingchun: @hugl950123:NN中一共有四个工程，它们之间没有任何关系，都是独立的，如果要运行这篇文章的...

```
    double activation_function_tanh_derivative(double x); //激活函数tanh的导数
    double activation_function_identity(double x);
    double activation_function_identity_derivative(double x);
    double loss_function_mse(double y, double t); //损失函数:mean squared error
    double loss_function_mse_derivative(double y, double t);
    void loss_function_gradient(const double* y, const double* t, double* dst, int len);
    double dot_product(const double* s1, const double* s2, int len); //点乘
    bool muladd(const double* src, double c, int len, double* dst); //dst[i] += c * src[i
    void init_variable(double* val, double c, int len);
    bool uniform_rand(double* src, int len, double min, double max);
    double uniform_rand(double min, double max);
    int get_index(int x, int y, int channel, int width, int height, int depth);
    void calc_out2wi(int width_in, int height_in, int width_out, int height_out, int deptl
    void calc_out2bias(int width, int height, int depth, std::vector<int>& out2bias);
    void calc_in2wo(int width_in, int height_in, int width_out, int height_out, int depth_
    void calc_weight2io(int width_in, int height_in, int width_out, int height_out, int de
    void calc_bias2out(int width_in, int height_in, int width_out, int height_out, int de

    bool Forward_C1(); //前向传播
    bool Forward_S2();
    bool Forward_C3();
    bool Forward_S4();
    bool Forward_C5();
    bool Forward_output();
    bool Backward_output();
    bool Backward_C5(); //反向传播
    bool Backward_S4();
    bool Backward_C3();
    bool Backward_S2();
    bool Backward_C1();
    bool Backward_input();
    bool UpdateWeights(); //更新权值、阈值
    void update_weights_bias(const double* delta, double* e_weight, double* weight, int le

private:
    double* data_input_train; //原始标准输入数据，训练,范围：[-1, 1]
    double* data_output_train; //原始标准期望结果，训练,取值：-0.8/0.8
    double* data_input_test; //原始标准输入数据，测试,范围：[-1, 1]
    double* data_output_test; //原始标准期望结果，测试,取值：-0.8/0.8
    double* data_single_image;
    double* data_single_label;

    double weight_C1[len_weight_C1_CNN];
    double bias_C1[len_bias_C1_CNN];
    double weight_S2[len_weight_S2_CNN];
    double bias_S2[len_bias_S2_CNN];
    double weight_C3[len_weight_C3_CNN];
    double bias_C3[len_bias_C3_CNN];
    double weight_S4[len_weight_S4_CNN];
    double bias_S4[len_bias_S4_CNN];
    double weight_C5[len_weight_C5_CNN];
    double bias_C5[len_bias_C5_CNN];
    double weight_output[len_weight_output_CNN];
    double bias_output[len_bias_output_CNN];

    double E_weight_C1[len_weight_C1_CNN];
    double E_bias_C1[len_bias_C1_CNN];
    double E_weight_S2[len_weight_S2_CNN];
    double E_bias_S2[len_bias_S2_CNN];
    double E_weight_C3[len_weight_C3_CNN];
    double E_bias_C3[len_bias_C3_CNN];
    double E_weight_S4[len_weight_S4_CNN];
    double E_bias_S4[len_bias_S4_CNN];
    double* E_weight_C5;
    double* E_bias_C5;
    double* E_weight_output;
    double* E_bias_output;

    double neuron_input[num_neuron_input_CNN]; //data_single_image
    double neuron_C1[num_neuron_C1_CNN];
    double neuron_S2[num_neuron_S2_CNN];
    double neuron_C3[num_neuron_C3_CNN];
    double neuron_S4[num_neuron_S4_CNN];
    double neuron_C5[num_neuron_C5_CNN];
    double neuron_output[num_neuron_output_CNN];

    double delta_neuron_output[num_neuron_output_CNN]; //神经元误差
    double delta_neuron_C5[num_neuron_C5_CNN];
    double delta_neuron_S4[num_neuron_S4_CNN];
```

关闭

卷积神经网络(CNN)的简单实现(
hugl950123: @fengbingchun:下的是新的，我在CNN.cpp文件中每个函数都设置了断点，还是没有变化=...
卷积神经网络(CNN)的简单实现(
fengbingchun: @hugl950123:你用的是GitHub上最新的吗？既然能编译过，在Debug下设断点，应该很快...
卷积神经网络(CNN)的简单实现(
hugl950123: 博主，请问我按照您的代码成功编译后执行结果窗口一闪而过，并且里面什么内容也没有，应该如何解决，能不能...

阅读排行

C#中OpenFileDialog的使 (47141)
tesseract-ocr3.02字符识 (34575)
举例说明使用MATLAB C (25987)
OpenCV中resize函数五种 (24317)
利用cvMinAreaRect2求取 (24277)
Windows 7 64位机上搭建 (22586)
opencv 检测直线、线段、 (20776)
OpenCV运动检测跟踪(bl (20475)
图像配准算法 (19237)
有效的rtsp流媒体测试地: (19143)

文章存档

2017年01月 (18)
2016年12月 (11)
2016年11月 (8)
2016年10月 (7)
2016年09月 (16)
展开


```
    double delta_neuron_C3[num_neuron_C3_CNN];
    double delta_neuron_S2[num_neuron_S2_CNN];
    double delta_neuron_C1[num_neuron_C1_CNN];
    double delta_neuron_input[num_neuron_input_CNN];

    double delta_weight_C1[len_weight_C1_CNN]; //权值、阈值误差
    double delta_bias_C1[len_bias_C1_CNN];
    double delta_weight_S2[len_weight_S2_CNN];
    double delta_bias_S2[len_bias_S2_CNN];
    double delta_weight_C3[len_weight_C3_CNN];
    double delta_bias_C3[len_bias_C3_CNN];
    double delta_weight_S4[len_weight_S4_CNN];
    double delta_bias_S4[len_bias_S4_CNN];
    double delta_weight_C5[len_weight_C5_CNN];
    double delta_bias_C5[len_bias_C5_CNN];
    double delta_weight_output[len_weight_output_CNN];
    double delta_bias_output[len_bias_output_CNN];

    std::vector<wi_connections> out2wi_S2; // out_id -> [(weight_id, in_id)]
    std::vector<int> out2bias_S2;
    std::vector<wi_connections> out2wi_S4;
    std::vector<int> out2bias_S4;
    std::vector<wo_connections> in2wo_C3; // in_id -> [(weight_id, out_id)]
    std::vector<io_connections> weight2io_C3; // weight_id -> [(in_id, out_id
    std::vector<std::vector<int> > bias2out_C3;
    std::vector<wo_connections> in2wo_C1;
    std::vector<io_connections> weight2io_C1;
    std::vector<std::vector<int> > bias2out_C1;
};

}

#endif //_CNN_HPP_
```

CNN.cpp：

[cpp]

```
#include <CNN.hpp>
#include <assert.h>
#include <time.h>
#include <iostream>
#include <fstream>
#include <numeric>
#include <windows.h>
#include <random>
#include <algorithm>
#include <string>

namespace ANN {

CNN::CNN()
{
    data_input_train = NULL;
    data_output_train = NULL;
    data_input_test = NULL;
    data_output_test = NULL;
    data_single_image = NULL;
    data_single_label = NULL;
    E_weight_C5 = NULL;
    E_bias_C5 = NULL;
    E_weight_output = NULL;
    E_bias_output = NULL;
}

CNN::~CNN()
{
    release();
}

void CNN::release()
{
    if (data_input_train) {
        delete[] data_input_train;
        data_input_train = NULL;
    }
    if (data_output_train) {
        delete[] data_output_train;
        data_output_train = NULL;
```

关闭

```
    }
    if (data_input_test) {
        delete[] data_input_test;
        data_input_test = NULL;
    }
    if (data_output_test) {
        delete[] data_output_test;
        data_output_test = NULL;
    }

    if (E_weight_C5) {
        delete[] E_weight_C5;
        E_weight_C5 = NULL;
    }
    if (E_bias_C5) {
        delete[] E_bias_C5;
        E_bias_C5 = NULL;
    }
    if (E_weight_output) {
        delete[] E_weight_output;
        E_weight_output = NULL;
    }
    if (E_bias_output) {
        delete[] E_bias_output;
        E_bias_output = NULL;
    }
}

// connection table [Y.Lecun, 1998 Table.1]
#define O true
#define X false
static const bool tbl[6][16] = {
    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

void CNN::init_variable(double* val, double c, int len)
{
    for (int i = 0; i < len; i++) {
        val[i] = c;
    }
}

void CNN::init()
{
    int len1 = width_image_input_CNN * height_image_input_CNN * num_patterns_train_CNN;
    data_input_train = new double[len1];
    init_variable(data_input_train, -1.0, len1);

    int len2 = num_map_output_CNN * num_patterns_train_CNN;
    data_output_train = new double[len2];
    init_variable(data_output_train, -0.8, len2);

    int len3 = width_image_input_CNN * height_image_input_CNN * num_patterns_test_CNN;
    data_input_test = new double[len3];
    init_variable(data_input_test, -1.0, len3);

    int len4 = num_map_output_CNN * num_patterns_test_CNN;
    data_output_test = new double[len4];
    init_variable(data_output_test, -0.8, len4);

    std::fill(E_weight_C1, E_weight_C1 + len_weight_C1_CNN, 0.0);
    std::fill(E_bias_C1, E_bias_C1 + len_bias_C1_CNN, 0.0);
    std::fill(E_weight_S2, E_weight_S2 + len_weight_S2_CNN, 0.0);
    std::fill(E_bias_S2, E_bias_S2 + len_bias_S2_CNN, 0.0);
    std::fill(E_weight_C3, E_weight_C3 + len_weight_C3_CNN, 0.0);
    std::fill(E_bias_C3, E_bias_C3 + len_bias_C3_CNN, 0.0);
    std::fill(E_weight_S4, E_weight_S4 + len_weight_S4_CNN, 0.0);
    std::fill(E_bias_S4, E_bias_S4 + len_bias_S4_CNN, 0.0);
    E_weight_C5 = new double[len_weight_C5_CNN];
    std::fill(E_weight_C5, E_weight_C5 + len_weight_C5_CNN, 0.0);
    E_bias_C5 = new double[len_bias_C5_CNN];
    std::fill(E_bias_C5, E_bias_C5 + len_bias_C5_CNN, 0.0);
```

关闭

```
    E_weight_output = new double[len_weight_output_CNN];
    std::fill(E_weight_output, E_weight_output + len_weight_output_CNN, 0.0);
    E_bias_output = new double[len_bias_output_CNN];
    std::fill(E_bias_output, E_bias_output + len_bias_output_CNN, 0.0);

    initWeightThreshold();
    getSrcData();
}

double CNN::uniform_rand(double min, double max)
{
    static std::mt19937 gen(1);
    std::uniform_real_distribution<double> dst(min, max);
    return dst(gen);
}

bool CNN::uniform_rand(double* src, int len, double min, double max)
{
    for (int i = 0; i < len; i++) {
        src[i] = uniform_rand(min, max);
    }

    return true;
}

bool CNN::initWeightThreshold()
{
    srand(time(0) + rand());
    const double scale = 6.0;

    double min_ = -std::sqrt(scale / (25.0 + 150.0));
    double max_ = std::sqrt(scale / (25.0 + 150.0));
    uniform_rand(weight_C1, len_weight_C1_CNN, min_, max_);
    for (int i = 0; i < len_bias_C1_CNN; i++) {
        bias_C1[i] = 0.0;
    }

    min_ = -std::sqrt(scale / (4.0 + 1.0));
    max_ = std::sqrt(scale / (4.0 + 1.0));
    uniform_rand(weight_S2, len_weight_S2_CNN, min_, max_);
    for (int i = 0; i < len_bias_S2_CNN; i++) {
        bias_S2[i] = 0.0;
    }

    min_ = -std::sqrt(scale / (150.0 + 400.0));
    max_ = std::sqrt(scale / (150.0 + 400.0));
    uniform_rand(weight_C3, len_weight_C3_CNN, min_, max_);
    for (int i = 0; i < len_bias_C3_CNN; i++) {
        bias_C3[i] = 0.0;
    }

    min_ = -std::sqrt(scale / (4.0 + 1.0));
    max_ = std::sqrt(scale / (4.0 + 1.0));
    uniform_rand(weight_S4, len_weight_S4_CNN, min_, max_);
    for (int i = 0; i < len_bias_S4_CNN; i++) {
        bias_S4[i] = 0.0;
    }

    min_ = -std::sqrt(scale / (400.0 + 3000.0));
    max_ = std::sqrt(scale / (400.0 + 3000.0));
    uniform_rand(weight_C5, len_weight_C5_CNN, min_, max_);
    for (int i = 0; i < len_bias_C5_CNN; i++) {
        bias_C5[i] = 0.0;
    }

    min_ = -std::sqrt(scale / (120.0 + 10.0));
    max_ = std::sqrt(scale / (120.0 + 10.0));
    uniform_rand(weight_output, len_weight_output_CNN, min_, max_);
    for (int i = 0; i < len_bias_output_CNN; i++) {
        bias_output[i] = 0.0;
    }

    return true;
}

static int reverseInt(int i)
{
    unsigned char ch1, ch2, ch3, ch4;
    ch1 = i & 255;
```

关闭

```
    ch2 = (i >> 8) & 255;
    ch3 = (i >> 16) & 255;
    ch4 = (i >> 24) & 255;
    return((int)ch1 << 24) + ((int)ch2 << 16) + ((int)ch3 << 8) + ch4;
}

static void readMnistImages(std::string filename, double* data_dst, int num_image)
{
    const int width_src_image = 28;
    const int height_src_image = 28;
    const int x_padding = 2;
    const int y_padding = 2;
    const double scale_min = -1;
    const double scale_max = 1;

    std::ifstream file(filename, std::ios::binary);
    assert(file.is_open());

    int magic_number = 0;
    int number_of_images = 0;
    int n_rows = 0;
    int n_cols = 0;
    file.read((char*)&magic_number, sizeof(magic_number));
    magic_number = reverseInt(magic_number);
    file.read((char*)&number_of_images, sizeof(number_of_images));
    number_of_images = reverseInt(number_of_images);
    assert(number_of_images == num_image);
    file.read((char*)&n_rows, sizeof(n_rows));
    n_rows = reverseInt(n_rows);
    file.read((char*)&n_cols, sizeof(n_cols));
    n_cols = reverseInt(n_cols);
    assert(n_rows == height_src_image && n_cols == width_src_image);

    int size_single_image = width_image_input_CNN * height_image_input_CNN;

    for (int i = 0; i < number_of_images; ++i) {
        int addr = size_single_image * i;

        for (int r = 0; r < n_rows; ++r) {
            for (int c = 0; c < n_cols; ++c) {
                unsigned char temp = 0;
                file.read((char*)&temp, sizeof(temp));
                data_dst[addr + width_image_input_CNN * (r + y_padding) + c + x_padding] =
            }
        }
    }
}

static void readMnistLabels(std::string filename, double* data_dst, int num_image)
{
    const double scale_max = 0.8;

    std::ifstream file(filename, std::ios::binary);
    assert(file.is_open());

    int magic_number = 0;
    int number_of_images = 0;
    file.read((char*)&magic_number, sizeof(magic_number));
    magic_number = reverseInt(magic_number);
    file.read((char*)&number_of_images, sizeof(number_of_images));
    number_of_images = reverseInt(number_of_images);
    assert(number_of_images == num_image);

    for (int i = 0; i < number_of_images; ++i) {
        unsigned char temp = 0;
        file.read((char*)&temp, sizeof(temp));
        data_dst[i * num_map_output_CNN + temp] = scale_max;
    }
}

bool CNN::getSrcData()
{
    assert(data_input_train && data_output_train && data_input_test && data_output_test);

    std::string filename_train_images = "E:/GitCode/NN_Test/data/train-images.idx3-
ubyte";
    std::string filename_train_labels = "E:/GitCode/NN_Test/data/train-labels.idx1-
ubyte";
    readMnistImages(filename_train_images, data_input_train, num_patterns_train_CNN);
```

关闭

```
    readMnistLabels(filename_train_labels, data_output_train, num_patterns_train_CNN);

    std::string filename_test_images = "E:/GitCode/NN_Test/data/t10k-images.idx3-
ubyte";
    std::string filename_test_labels = "E:/GitCode/NN_Test/data/t10k-labels.idx1-
ubyte";
    readMnistImages(filename_test_images, data_input_test, num_patterns_test_CNN);
    readMnistLabels(filename_test_labels, data_output_test, num_patterns_test_CNN);

    return true;
}

bool CNN::train()
{
    out2wi_S2.clear();
    out2bias_S2.clear();
    out2wi_S4.clear();
    out2bias_S4.clear();
    in2wo_C3.clear();
    weight2io_C3.clear();
    bias2out_C3.clear();
    in2wo_C1.clear();
    weight2io_C1.clear();
    bias2out_C1.clear();

    calc_out2wi(width_image_C1_CNN, height_image_C1_CNN, width_image_S2_CNN, height_image_
    calc_out2bias(width_image_S2_CNN, height_image_S2_CNN, num_map_S2_CNN, out2bias_S2);
    calc_out2wi(width_image_C3_CNN, height_image_C3_CNN, width_image_S4_CNN, height_image_
    calc_out2bias(width_image_S4_CNN, height_image_S4_CNN, num_map_S4_CNN, out2bias_S4);
    calc_in2wo(width_image_C3_CNN, height_image_C3_CNN, width_image_S4_CNN, height_image_S
    calc_weight2io(width_image_C3_CNN, height_image_C3_CNN, width_image_S4_CNN, height_ima
    calc_bias2out(width_image_C3_CNN, height_image_C3_CNN, width_image_S4_CNN, height_imag
    calc_in2wo(width_image_C1_CNN, height_image_C1_CNN, width_image_S2_CNN, height_image_S
    calc_weight2io(width_image_C1_CNN, height_image_C1_CNN, width_image_S2_CNN, height_ima
    calc_bias2out(width_image_C1_CNN, height_image_C1_CNN, width_image_S2_CNN, height_imag

    int iter = 0;
    for (iter = 0; iter < num_epochs_CNN; iter++) {
        std::cout << "epoch: " << iter + 1;

        for (int i = 0; i < num_patterns_train_CNN; i++) {
            data_single_image = data_input_train + i * num_neuron_input_CNN;
            data_single_label = data_output_train + i * num_neuron_output_CNN;

            Forward_C1();
            Forward_S2();
            Forward_C3();
            Forward_S4();
            Forward_C5();
            Forward_output();

            Backward_output();
            Backward_C5();
            Backward_S4();
            Backward_C3();
            Backward_S2();
            Backward_C1();
            Backward_input();

            UpdateWeights();
        }

        double accuracyRate = test();
        std::cout << ",    accuray rate: " << accuracyRate << std::endl;
        if (accuracyRate > accuracy_rate_CNN) {
            saveModelFile("E:/GitCode/NN_Test/data/cn
            std::cout << "generate cnn model" << std::endl;
            break;
        }
    }

    if (iter == num_epochs_CNN) {
        saveModelFile("E:/GitCode/NN_Test/data/cnn.model");
        std::cout << "generate cnn model" << std::endl;
    }

    return true;
}

```

关闭

```
double CNN::activation_function_tanh(double x)
{
    double ep = std::exp(x);
    double em = std::exp(-x);

    return (ep - em) / (ep + em);
}

double CNN::activation_function_tanh_derivative(double x)
{
    return (1.0 - x * x);
}

double CNN::activation_function_identity(double x)
{
    return x;
}

double CNN::activation_function_identity_derivative(double x)
{
    return 1;
}

double CNN::loss_function_mse(double y, double t)
{
    return (y - t) * (y - t) / 2;
}

double CNN::loss_function_mse_derivative(double y, double t)
{
    return (y - t);
}

void CNN::loss_function_gradient(const double* y, const double* t, double* dst, int len)
{
    for (int i = 0; i < len; i++) {
        dst[i] = loss_function_mse_derivative(y[i], t[i]);
    }
}

double CNN::dot_product(const double* s1, const double* s2, int len)
{
    double result = 0.0;

    for (int i = 0; i < len; i++) {
        result += s1[i] * s2[i];
    }

    return result;
}

bool CNN::muladd(const double* src, double c, int len, double* dst)
{
    for (int i = 0; i < len; i++) {
        dst[i] += (src[i] * c);
    }

    return true;
}

int CNN::get_index(int x, int y, int channel, int width, int height, int depth)
{
    assert(x >= 0 && x < width);
    assert(y >= 0 && y < height);
    assert(channel >= 0 && channel < depth);
    return (height * channel + y) * width + x;
}

void CNN::calc_out2wi(int width_in, int height_in, int width_out, int height_out, int dept
{
    for (int i = 0; i < depth_out; i++) {
        int block = width_in * height_in * i;

        for (int y = 0; y < height_out; y++) {
            for (int x = 0; x < width_out; x++) {
                int rows = y * width_kernel_pooling_CNN;
                int cols = x * height_kernel_pooling_CNN;

                wi_connections wi_connections_;
```

关闭

```
                std::pair<int, int> pair_;

                for (int m = 0; m < width_kernel_pooling_CNN; m++) {
                    for (int n = 0; n < height_kernel_pooling_CNN; n++) {
                        pair_.first = i;
                        pair_.second = (rows + m) * width_in + cols + n + block;
                        wi_connections_.push_back(pair_);
                    }
                }
                out2wi.push_back(wi_connections_);
            }
        }
    }
}

void CNN::calc_out2bias(int width, int height, int depth, std::vector<int>& out2bias)
{
    for (int i = 0; i < depth; i++) {
        for (int y = 0; y < height; y++) {
            for (int x = 0; x < width; x++) {
                out2bias.push_back(i);
            }
        }
    }
}

void CNN::calc_in2wo(int width_in, int height_in, int width_out, int height_out, int deptl
{
    int len = width_in * height_in * depth_in;
    in2wo.resize(len);

    for (int c = 0; c < depth_in; c++) {
        for (int y = 0; y < height_in; y += height_kernel_pooling_CNN) {
            for (int x = 0; x < width_in; x += width_kernel_pooling_CNN) {
                int dymax = min(size_pooling_CNN, height_in - y);
                int dxmax = min(size_pooling_CNN, width_in - x);
                int dstx = x / width_kernel_pooling_CNN;
                int dsty = y / height_kernel_pooling_CNN;

                for (int dy = 0; dy < dymax; dy++) {
                    for (int dx = 0; dx < dxmax; dx++) {
                        int index_in = get_index(x + dx, y + dy, c, width_in, height_in, (
                        int index_out = get_index(dstx, dsty, c, width_out, height_out, de

                        wo_connections wo_connections_;
                        std::pair<int, int> pair_;
                        pair_.first = c;
                        pair_.second = index_out;
                        wo_connections_.push_back(pair_);

                        in2wo[index_in] = wo_connections_;
                    }
                }
            }
        }
    }
}

void CNN::calc_weight2io(int width_in, int height_in, int width_out, int height_out, int (
{
    int len = depth_in;
    weight2io.resize(len);

    for (int c = 0; c < depth_in; c++) {
        for (int y = 0; y < height_in; y += height_kernel_pooling_CNN) {
            for (int x = 0; x < width_in; x += width_l
                int dymax = min(size_pooling_CNN, height_in - y);
                int dxmax = min(size_pooling_CNN, width_in - x);
                int dstx = x / width_kernel_pooling_CNN;
                int dsty = y / height_kernel_pooling_CNN;

                for (int dy = 0; dy < dymax; dy++) {
                    for (int dx = 0; dx < dxmax; dx++) {
                        int index_in = get_index(x + dx, y + dy, c, width_in, height_in, (
                        int index_out = get_index(dstx, dsty, c, width_out, height_out, de

                        std::pair<int, int> pair_;
                        pair_.first = index_in;
                        pair_.second = index_out;
```

关闭

```
                        weight2io[c].push_back(pair_);
                    }
                }
            }
        }
    }
}

void CNN::calc_bias2out(int width_in, int height_in, int width_out, int height_out, int de
{
    int len = depth_in;
    bias2out.resize(len);

    for (int c = 0; c < depth_in; c++) {
        for (int y = 0; y < height_out; y++) {
            for (int x = 0; x < width_out; x++) {
                int index_out = get_index(x, y, c, width_out, height_out, depth_out);
                bias2out[c].push_back(index_out);
            }
        }
    }
}

bool CNN::Forward_C1()
{
    init_variable(neuron_C1, 0.0, num_neuron_C1_CNN);

    for (int o = 0; o < num_map_C1_CNN; o++) {
        for (int inc = 0; inc < num_map_input_CNN; inc++) {
            int addr1 = get_index(0, 0, num_map_input_CNN * o + inc, width_kernel_conv_CNN
            int addr2 = get_index(0, 0, inc, width_image_input_CNN, height_image_input_CNN
            int addr3 = get_index(0, 0, o, width_image_C1_CNN, height_image_C1_CNN, num_ma

            const double* pw = &weight_C1[0] + addr1;
            const double* pi = data_single_image + addr2;
            double* pa = &neuron_C1[0] + addr3;

            for (int y = 0; y < height_image_C1_CNN; y++) {
                for (int x = 0; x < width_image_C1_CNN; x++) {
                    const double* ppw = pw;
                    const double* ppi = pi + y * width_image_input_CNN + x;
                    double sum = 0.0;

                    for (int wy = 0; wy < height_kernel_conv_CNN; wy++) {
                        for (int wx = 0; wx < width_kernel_conv_CNN; wx++) {
                            sum += *ppw++ * ppi[wy * width_image_input_CNN + wx];
                        }
                    }

                    pa[y * width_image_C1_CNN + x] += sum;
                }
            }
        }

        int addr3 = get_index(0, 0, o, width_image_C1_CNN, height_image_C1_CNN, num_map_C1
        double* pa = &neuron_C1[0] + addr3;
        double b = bias_C1[o];
        for (int y = 0; y < height_image_C1_CNN; y++) {
            for (int x = 0; x < width_image_C1_CNN; x++) {
                pa[y * width_image_C1_CNN + x] += b;
            }
        }
    }

    for (int i = 0; i < num_neuron_C1_CNN; i++) {
        neuron_C1[i] = activation_function_tanh(neuron_C1[i]);
    }

    return true;
}

bool CNN::Forward_S2()
{
    init_variable(neuron_S2, 0.0, num_neuron_S2_CNN);
    double scale_factor = 1.0 / (width_kernel_pooling_CNN * height_kernel_pooling_CNN);

    assert(out2wi_S2.size() == num_neuron_S2_CNN);
    assert(out2bias_S2.size() == num_neuron_S2_CNN);
```

关闭

```

    for (int i = 0; i < num_neuron_S2_CNN; i++) {
        const wi_connections& connections = out2wi_S2[i];
        neuron_S2[i] = 0;

        for (int index = 0; index < connections.size(); index++) {
            neuron_S2[i] += weight_S2[connections[index].first] * neuron_C1[connections[in
        }

        neuron_S2[i] *= scale_factor;
        neuron_S2[i] += bias_S2[out2bias_S2[i]];
    }

    for (int i = 0; i < num_neuron_S2_CNN; i++) {
        neuron_S2[i] = activation_function_tanh(neuron_S2[i]);
    }

    return true;
}

bool CNN::Forward_C3()
{
    init_variable(neuron_C3, 0.0, num_neuron_C3_CNN);

    for (int o = 0; o < num_map_C3_CNN; o++) {
        for (int inc = 0; inc < num_map_S2_CNN; inc++) {
            if (!tbl[inc][o]) continue;

            int addr1 = get_index(0, 0, num_map_S2_CNN * o + inc, width_kernel_conv_CNN, h
            int addr2 = get_index(0, 0, inc, width_image_S2_CNN, height_image_S2_CNN, num_
            int addr3 = get_index(0, 0, o, width_image_C3_CNN, height_image_C3_CNN, num_ma

            const double* pw = &weight_C3[0] + addr1;
            const double* pi = &neuron_S2[0] + addr2;
            double* pa = &neuron_C3[0] + addr3;

            for (int y = 0; y < height_image_C3_CNN; y++) {
                for (int x = 0; x < width_image_C3_CNN; x++) {
                    const double* ppw = pw;
                    const double* ppi = pi + y * width_image_S2_CNN + x;
                    double sum = 0.0;

                    for (int wy = 0; wy < height_kernel_conv_CNN; wy++) {
                        for (int wx = 0; wx < width_kernel_conv_CNN; wx++) {
                            sum += *ppw++ * ppi[wy * width_image_S2_CNN + wx];
                        }
                    }

                    pa[y * width_image_C3_CNN + x] += sum;
                }
            }
        }

        int addr3 = get_index(0, 0, o, width_image_C3_CNN, height_image_C3_CNN, num_map_C3
        double* pa = &neuron_C3[0] + addr3;
        double b = bias_C3[o];
        for (int y = 0; y < height_image_C3_CNN; y++) {
            for (int x = 0; x < width_image_C3_CNN; x++) {
                pa[y * width_image_C3_CNN + x] += b;
            }
        }
    }

    for (int i = 0; i < num_neuron_C3_CNN; i++) {
        neuron_C3[i] = activation_function_tanh(neuron_C3[i]);
    }

    return true;
}

bool CNN::Forward_S4()
{
    double scale_factor = 1.0 / (width_kernel_pooling_CNN * height_kernel_pooling_CNN);
    init_variable(neuron_S4, 0.0, num_neuron_S4_CNN);

    assert(out2wi_S4.size() == num_neuron_S4_CNN);
    assert(out2bias_S4.size() == num_neuron_S4_CNN);

    for (int i = 0; i < num_neuron_S4_CNN; i++) {
```

关闭

```
        const wi_connections& connections = out2wi_S4[i];
        neuron_S4[i] = 0.0;

        for (int index = 0; index < connections.size(); index++) {
            neuron_S4[i] += weight_S4[connections[index].first] * neuron_C3[connections[ir
        }

        neuron_S4[i] *= scale_factor;
        neuron_S4[i] += bias_S4[out2bias_S4[i]];
    }

    for (int i = 0; i < num_neuron_S4_CNN; i++) {
        neuron_S4[i] = activation_function_tanh(neuron_S4[i]);
    }

    return true;
}

bool CNN::Forward_C5()
{
    init_variable(neuron_C5, 0.0, num_neuron_C5_CNN);

    for (int o = 0; o < num_map_C5_CNN; o++) {
        for (int inc = 0; inc < num_map_S4_CNN; inc++) {
            int addr1 = get_index(0, 0, num_map_S4_CNN * o + inc, width_kernel_conv_CNN, h
            int addr2 = get_index(0, 0, inc, width_image_S4_CNN, height_image_S4_CNN, num_
            int addr3 = get_index(0, 0, o, width_image_C5_CNN, height_image_C5_CNN, num_ma

            const double *pw = &weight_C5[0] + addr1;
            const double *pi = &neuron_S4[0] + addr2;
            double *pa = &neuron_C5[0] + addr3;

            for (int y = 0; y < height_image_C5_CNN; y++) {
                for (int x = 0; x < width_image_C5_CNN; x++) {
                    const double *ppw = pw;
                    const double *ppi = pi + y * width_image_S4_CNN + x;
                    double sum = 0.0;

                    for (int wy = 0; wy < height_kernel_conv_CNN; wy++) {
                        for (int wx = 0; wx < width_kernel_conv_CNN; wx++) {
                            sum += *ppw++ * ppi[wy * width_image_S4_CNN + wx];
                        }
                    }

                    pa[y * width_image_C5_CNN + x] += sum;
                }
            }
        }

        int addr3 = get_index(0, 0, o, width_image_C5_CNN, height_image_C5_CNN, num_map_C5
        double *pa = &neuron_C5[0] + addr3;
        double b = bias_C5[o];
        for (int y = 0; y < height_image_C5_CNN; y++) {
            for (int x = 0; x < width_image_C5_CNN; x++) {
                pa[y * width_image_C5_CNN + x] += b;
            }
        }
    }

    for (int i = 0; i < num_neuron_C5_CNN; i++) {
        neuron_C5[i] = activation_function_tanh(neuron_C5[i]);
    }

    return true;
}

bool CNN::Forward_output()
{
    init_variable(neuron_output, 0.0, num_neuron_output_CNN);

    for (int i = 0; i < num_neuron_output_CNN; i++) {
        neuron_output[i] = 0.0;

        for (int c = 0; c < num_neuron_C5_CNN; c++) {
            neuron_output[i] += weight_output[c * num_neuron_output_CNN + i] * neuron_C5[c
        }

        neuron_output[i] += bias_output[i];
    }
```

关闭

```

    for (int i = 0; i < num_neuron_output_CNN; i++) {
        neuron_output[i] = activation_function_tanh(neuron_output[i]);
    }

    return true;
}

bool CNN::Backward_output()
{
    init_variable(delta_neuron_output, 0.0, num_neuron_output_CNN);

    double dE_dy[num_neuron_output_CNN];
    init_variable(dE_dy, 0.0, num_neuron_output_CNN);
    loss_function_gradient(neuron_output, data_single_label, dE_dy, num_neuron_output_CNN
失函数: mean squared error(均方差)

    // delta = dE/da = (dE/dy) * (dy/da)
    for (int i = 0; i < num_neuron_output_CNN; i++) {
        double dy_da[num_neuron_output_CNN];
        init_variable(dy_da, 0.0, num_neuron_output_CNN);

        dy_da[i] = activation_function_tanh_derivative(neuron_output[i]);
        delta_neuron_output[i] = dot_product(dE_dy, dy_da, num_neuron_output_
    }

    return true;
}

bool CNN::Backward_C5()
{
    init_variable(delta_neuron_C5, 0.0, num_neuron_C5_CNN);
    init_variable(delta_weight_output, 0.0, len_weight_output_CNN);
    init_variable(delta_bias_output, 0.0, len_bias_output_CNN);

    for (int c = 0; c < num_neuron_C5_CNN; c++) {
        // propagate delta to previous layer
        // prev_delta[c] += current_delta[r] * W_[c * out_size_ + r]
        delta_neuron_C5[c] = dot_product(&
delta_neuron_output[0], &weight_output[c * num_neuron_output_CNN], num_neuron_output_CNN)
        delta_neuron_C5[c] *= activation_function_tanh_derivative(neuron_C5[c]);
    }

    // accumulate weight-step using delta
    // dW[c * out_size + i] += current_delta[i] * prev_out[c]
    for (int c = 0; c < num_neuron_C5_CNN; c++) {
        muladd(&delta_neuron_output[0], neuron_C5[c], num_neuron_output_CNN, &delta_weigh
    }

    for (int i = 0; i < len_bias_output_CNN; i++) {
        delta_bias_output[i] += delta_neuron_output[i];
    }

    return true;
}

bool CNN::Backward_S4()
{
    init_variable(delta_neuron_S4, 0.0, num_neuron_S4_CNN);
    init_variable(delta_weight_C5, 0.0, len_weight_C5_CNN);
    init_variable(delta_bias_C5, 0.0, len_bias_C5_CNN);

    // propagate delta to previous layer
    for (int inc = 0; inc < num_map_S4_CNN; inc++) {
        for (int outc = 0; outc < num_map_C5_CNN; outc++) {
            int addr1 = get_index(0, 0, num_map_S4_CN
            int addr2 = get_index(0, 0, outc, width_image_C5_CNN, height_image_C5_CNN, nu
            int addr3 = get_index(0, 0, inc, width_image_S4_CNN, height_image_S4_CNN, num_

            const double* pw = &weight_C5[0] + addr1;
            const double* pdelta_src = &delta_neuron_C5[0] + addr2;
            double* pdelta_dst = &delta_neuron_S4[0] + addr3;

            for (int y = 0; y < height_image_C5_CNN; y++) {
                for (int x = 0; x < width_image_C5_CNN; x++) {
                    const double* ppw = pw;
                    const double ppdelta_src = pdelta_src[y * width_image_C5_CNN + x];
                    double* ppdelta_dst = pdelta_dst + y * width_image_S4_CNN + x;

```

关闭

```
                        for (int wy = 0; wy < height_kernel_conv_CNN; wy++) {
                            for (int wx = 0; wx < width_kernel_conv_CNN; wx++) {
                                ppdelta_dst[wy * width_image_S4_CNN + wx] += *ppw++ * ppdelta_
                            }
                        }
                    }
                }
            }
        }

        for (int i = 0; i < num_neuron_S4_CNN; i++) {
            delta_neuron_S4[i] *= activation_function_tanh_derivative(neuron_S4[i]);
        }

        // accumulate dw
        for (int inc = 0; inc < num_map_S4_CNN; inc++) {
            for (int outc = 0; outc < num_map_C5_CNN; outc++) {
                for (int wy = 0; wy < height_kernel_conv_CNN; wy++) {
                    for (int wx = 0; wx < width_kernel_conv_CNN; wx++) {
                        int addr1 = get_index(wx, wy, inc, width_image_S4_CNN, height_image_S4
                        int addr2 = get_index(0, 0, outc, width_image_C5_CNN, height_image_C5_
                        int addr3 = get_index(wx, wy, num_map_S4_CNN * outc + inc, width_kerne

                        double dst = 0.0;
                        const double* prevo = &neuron_S4[0] + addr1;
                        const double* delta = &delta_neuron_C5[0] + addr2;

                        for (int y = 0; y < height_image_C5_CNN; y++) {
                            dst += dot_product(prevo + y * width_image_S4_CNN, delta + y * wic
                        }

                        delta_weight_C5[addr3] += dst;
                    }
                }
            }
        }

        // accumulate db
        for (int outc = 0; outc < num_map_C5_CNN; outc++) {
            int addr2 = get_index(0, 0, outc, width_image_C5_CNN, height_image_C5_CNN, num_maj
            const double* delta = &delta_neuron_C5[0] + addr2;

            for (int y = 0; y < height_image_C5_CNN; y++) {
                for (int x = 0; x < width_image_C5_CNN; x++) {
                    delta_bias_C5[outc] += delta[y * width_image_C5_CNN + x];
                }
            }
        }

        return true;
    }

    bool CNN::Backward_C3()
    {
        init_variable(delta_neuron_C3, 0.0, num_neuron_C3_CNN);
        init_variable(delta_weight_S4, 0.0, len_weight_S4_CNN);
        init_variable(delta_bias_S4, 0.0, len_bias_S4_CNN);

        double scale_factor = 1.0 / (width_kernel_pooling_CNN * height_kernel_pooling_CNN);

        assert(in2wo_C3.size() == num_neuron_C3_CNN);
        assert(weight2io_C3.size() == len_weight_S4_CNN);
        assert(bias2out_C3.size() == len_bias_S4_CNN);

        for (int i = 0; i < num_neuron_C3_CNN; i++) {
            const wo_connections& connections = in2wo_C3[:
            double delta = 0.0;

            for (int j = 0; j < connections.size(); j++) {
                delta += weight_S4[connections[j].first] * delta_neuron_S4[connections[j].secc
            }

            delta_neuron_C3[i] = delta * scale_factor * activation_function_tanh_derivative(ne
        }

        for (int i = 0; i < len_weight_S4_CNN; i++) {
            const io_connections& connections = weight2io_C3[i];
            double diff = 0;

```

关闭

```
        for (int j = 0; j < connections.size(); j++) {
            diff += neuron_C3[connections[j].first] * delta_neuron_S4[connections[j].secon
        }

        delta_weight_S4[i] += diff * scale_factor;
    }

    for (int i = 0; i < len_bias_S4_CNN; i++) {
        const std::vector<int>& outs = bias2out_C3[i];
        double diff = 0;

        for (int o = 0; o < outs.size(); o++) {
            diff += delta_neuron_S4[outs[o]];
        }

        delta_bias_S4[i] += diff;
    }

    return true;
}

bool CNN::Backward_S2()
{
    init_variable(delta_neuron_S2, 0.0, num_neuron_S2_CNN);
    init_variable(delta_weight_C3, 0.0, len_weight_C3_CNN);
    init_variable(delta_bias_C3, 0.0, len_bias_C3_CNN);

    // propagate delta to previous layer
    for (int inc = 0; inc < num_map_S2_CNN; inc++) {
        for (int outc = 0; outc < num_map_C3_CNN; outc++) {
            if (!tbl[inc][outc]) continue;

            int addr1 = get_index(0, 0, num_map_S2_CNN * outc + inc, width_kernel_conv_CNN
            int addr2 = get_index(0, 0, outc, width_image_C3_CNN, height_image_C3_CNN, nur
            int addr3 = get_index(0, 0, inc, width_image_S2_CNN, height_image_S2_CNN, num_

            const double *pw = &weight_C3[0] + addr1;
            const double *pdelta_src = &delta_neuron_C3[0] + addr2;;
            double* pdelta_dst = &delta_neuron_S2[0] + addr3;

            for (int y = 0; y < height_image_C3_CNN; y++) {
                for (int x = 0; x < width_image_C3_CNN; x++) {
                    const double* ppw = pw;
                    const double ppdelta_src = pdelta_src[y * width_image_C3_CNN + x];
                    double* ppdelta_dst = pdelta_dst + y * width_image_S2_CNN + x;

                    for (int wy = 0; wy < height_kernel_conv_CNN; wy++) {
                        for (int wx = 0; wx < width_kernel_conv_CNN; wx++) {
                            ppdelta_dst[wy * width_image_S2_CNN + wx] += *ppw++ * ppdelta_
                        }
                    }
                }
            }
        }
    }

    for (int i = 0; i < num_neuron_S2_CNN; i++) {
        delta_neuron_S2[i] *= activation_function_tanh_derivative(neuron_S2[i]);
    }

    // accumulate dw
    for (int inc = 0; inc < num_map_S2_CNN; inc++) {
        for (int outc = 0; outc < num_map_C3_CNN; outc++) {
            if (!tbl[inc][outc]) continue;

            for (int wy = 0; wy < height_kernel_conv_(
                for (int wx = 0; wx < width_kernel_conv_CNN; wx++) {
                    int addr1 = get_index(wx, wy, inc, width_image_S2_CNN, height_image_S2
                    int addr2 = get_index(0, 0, outc, width_image_C3_CNN, height_image_C3_
                    int addr3 = get_index(wx, wy, num_map_S2_CNN * outc + inc, width_kerne

                    double dst = 0.0;
                    const double* prevo = &neuron_S2[0] + addr1;
                    const double* delta = &delta_neuron_C3[0] + addr2;

                    for (int y = 0; y < height_image_C3_CNN; y++) {
                        dst += dot_product(prevo + y * width_image_S2_CNN, delta + y * wid
                    }

```

关闭

```
                    delta_weight_C3[addr3] += dst;
                }
            }
        }
    }

    // accumulate db
    for (int outc = 0; outc < len_bias_C3_CNN; outc++) {
        int addr1 = get_index(0, 0, outc, width_image_C3_CNN, height_image_C3_CNN, num_map
        const double* delta = &delta_neuron_C3[0] + addr1;

        for (int y = 0; y < height_image_C3_CNN; y++) {
            for (int x = 0; x < width_image_C3_CNN; x++) {
                delta_bias_C3[outc] += delta[y * width_image_C3_CNN + x];
            }
        }
    }

    return true;
}

bool CNN::Backward_C1()
{
    init_variable(delta_neuron_C1, 0.0, num_neuron_C1_CNN);
    init_variable(delta_weight_S2, 0.0, len_weight_S2_CNN);
    init_variable(delta_bias_S2, 0.0, len_bias_S2_CNN);

    double scale_factor = 1.0 / (width_kernel_pooling_CNN * height_kernel_pooling_CNN);

    assert(in2wo_C1.size() == num_neuron_C1_CNN);
    assert(weight2io_C1.size() == len_weight_S2_CNN);
    assert(bias2out_C1.size() == len_bias_S2_CNN);

    for (int i = 0; i < num_neuron_C1_CNN; i++) {
        const wo_connections& connections = in2wo_C1[i];
        double delta = 0.0;

        for (int j = 0; j < connections.size(); j++) {
            delta += weight_S2[connections[j].first] * delta_neuron_S2[connections[j].seco
        }

        delta_neuron_C1[i] = delta * scale_factor * activation_function_tanh_derivative(ne
    }

    for (int i = 0; i < len_weight_S2_CNN; i++) {
        const io_connections& connections = weight2io_C1[i];
        double diff = 0.0;

        for (int j = 0; j < connections.size(); j++) {
            diff += neuron_C1[connections[j].first] * delta_neuron_S2[connections[j].seco
        }

        delta_weight_S2[i] += diff * scale_factor;
    }

    for (int i = 0; i < len_bias_S2_CNN; i++) {
        const std::vector<int>& outs = bias2out_C1[i];
        double diff = 0;

        for (int o = 0; o < outs.size(); o++) {
            diff += delta_neuron_S2[outs[o]];
        }

        delta_bias_S2[i] += diff;
    }

    return true;
}

bool CNN::Backward_input()
{
    init_variable(delta_neuron_input, 0.0, num_neuron_input_CNN);
    init_variable(delta_weight_C1, 0.0, len_weight_C1_CNN);
    init_variable(delta_bias_C1, 0.0, len_bias_C1_CNN);

    // propagate delta to previous layer
    for (int inc = 0; inc < num_map_input_CNN; inc++) {
        for (int outc = 0; outc < num_map_C1_CNN; outc++) {
            int addr1 = get_index(0, 0, num_map_input_CNN * outc + inc, width_kernel_conv_
```

关闭

```
                int addr2 = get_index(0, 0, outc, width_image_C1_CNN, height_image_C1_CNN, nur
                int addr3 = get_index(0, 0, inc, width_image_input_CNN, height_image_input_CNI

                const double* pw = &weight_C1[0] + addr1;
                const double* pdelta_src = &delta_neuron_C1[0] + addr2;
                double* pdelta_dst = &delta_neuron_input[0] + addr3;

                for (int y = 0; y < height_image_C1_CNN; y++) {
                    for (int x = 0; x < width_image_C1_CNN; x++) {
                        const double* ppw = pw;
                        const double ppdelta_src = pdelta_src[y * width_image_C1_CNN + x];
                        double* ppdelta_dst = pdelta_dst + y * width_image_input_CNN + x;

                        for (int wy = 0; wy < height_kernel_conv_CNN; wy++) {
                            for (int wx = 0; wx < width_kernel_conv_CNN; wx++) {
                                ppdelta_dst[wy * width_image_input_CNN + wx] += *ppw++ * ppde]
                            }
                        }
                    }
                }
            }
        }

        for (int i = 0; i < num_neuron_input_CNN; i++) {
            delta_neuron_input[i] *= activation_function_identity_derivative(data_single_image
        }

        // accumulate dw
        for (int inc = 0; inc < num_map_input_CNN; inc++) {
            for (int outc = 0; outc < num_map_C1_CNN; outc++) {
                for (int wy = 0; wy < height_kernel_conv_CNN; wy++) {
                    for (int wx = 0; wx < width_kernel_conv_CNN; wx++) {
                        int addr1 = get_index(wx, wy, inc, width_image_input_CNN, height_image
                        int addr2 = get_index(0, 0, outc, width_image_C1_CNN, height_image_C1_
                        int addr3 = get_index(wx, wy, num_map_input_CNN * outc + inc, width_ke

                        double dst = 0.0;
                        const double* prevo = data_single_image + addr1;//&neuron_input[0]
                        const double* delta = &delta_neuron_C1[0] + addr2;

                        for (int y = 0; y < height_image_C1_CNN; y++) {
                            dst += dot_product(prevo + y * width_image_input_CNN, delta + y *
                        }

                        delta_weight_C1[addr3] += dst;
                    }
                }
            }
        }

        // accumulate db
        for (int outc = 0; outc < len_bias_C1_CNN; outc++) {
            int addr1 = get_index(0, 0, outc, width_image_C1_CNN, height_image_C1_CNN, num_ma
            const double* delta = &delta_neuron_C1[0] + addr1;

            for (int y = 0; y < height_image_C1_CNN; y++) {
                for (int x = 0; x < width_image_C1_CNN; x++) {
                    delta_bias_C1[outc] += delta[y * width_image_C1_CNN + x];
                }
            }
        }

        return true;
    }

    void CNN::update_weights_bias(const double* delta, do
    {
        for (int i = 0; i < len; i++) {
            e_weight[i] += delta[i] * delta[i];
            weight[i] -= learning_rate_CNN * delta[i] / (std::sqrt(e_weight[i]) + eps_CNN);
        }
    }

    bool CNN::UpdateWeights()
    {
        update_weights_bias(delta_weight_C1, E_weight_C1, weight_C1, len_weight_C1_CNN);
        update_weights_bias(delta_bias_C1, E_bias_C1, bias_C1, len_bias_C1_CNN);

        update_weights_bias(delta_weight_S2, E_weight_S2, weight_S2, len_weight_S2_CNN);
```

关闭

```
    update_weights_bias(delta_bias_S2, E_bias_S2, bias_S2, len_bias_S2_CNN);

    update_weights_bias(delta_weight_C3, E_weight_C3, weight_C3, len_weight_C3_CNN);
    update_weights_bias(delta_bias_C3, E_bias_C3, bias_C3, len_bias_C3_CNN);

    update_weights_bias(delta_weight_S4, E_weight_S4, weight_S4, len_weight_S4_CNN);
    update_weights_bias(delta_bias_S4, E_bias_S4, bias_S4, len_bias_S4_CNN);

    update_weights_bias(delta_weight_C5, E_weight_C5, weight_C5, len_weight_C5_CNN);
    update_weights_bias(delta_bias_C5, E_bias_C5, bias_C5, len_bias_C5_CNN);

    update_weights_bias(delta_weight_output, E_weight_output, weight_output, len_weight_ou
    update_weights_bias(delta_bias_output, E_bias_output, bias_output, len_bias_output_CNN

    return true;
}

int CNN::predict(const unsigned char* data, int width, int height)
{
    assert(data && width == width_image_input_CNN && height == height_image_input_CNN);

    const double scale_min = -1;
    const double scale_max = 1;

    double tmp[width_image_input_CNN * height_image_input_CNN];
    for (int y = 0; y < height; y++) {
        for (int x = 0; x < width; x++) {
            tmp[y * width + x] = (data[y * width + x] / 255.0) * (scale_max - scale_min) -
        }
    }

    data_single_image = &tmp[0];

    Forward_C1();
    Forward_S2();
    Forward_C3();
    Forward_S4();
    Forward_C5();
    Forward_output();

    int pos = -1;
    double max_value = -9999.0;

    for (int i = 0; i < num_neuron_output_CNN; i++) {
        if (neuron_output[i] > max_value) {
            max_value = neuron_output[i];
            pos = i;
        }
    }

    return pos;
}

bool CNN::readModelFile(const char* name)
{
    FILE* fp = fopen(name, "rb");
    if (fp == NULL) {
        return false;
    }

    int width_image_input =0;
    int height_image_input = 0;
    int width_image_C1 = 0;
    int height_image_C1 = 0;
    int width_image_S2 = 0;
    int height_image_S2 = 0;
    int width_image_C3 = 0;
    int height_image_C3 = 0;
    int width_image_S4 = 0;
    int height_image_S4 = 0;
    int width_image_C5 = 0;
    int height_image_C5 = 0;
    int width_image_output = 0;
    int height_image_output = 0;

    int width_kernel_conv = 0;
    int height_kernel_conv = 0;
    int width_kernel_pooling = 0;
    int height_kernel_pooling = 0;
```

关闭

```

    int num_map_input = 0;
    int num_map_C1 = 0;
    int num_map_S2 = 0;
    int num_map_C3 = 0;
    int num_map_S4 = 0;
    int num_map_C5 = 0;
    int num_map_output = 0;

    int len_weight_C1 = 0;
    int len_bias_C1 = 0;
    int len_weight_S2 = 0;
    int len_bias_S2 = 0;
    int len_weight_C3 = 0;
    int len_bias_C3 = 0;
    int len_weight_S4 = 0;
    int len_bias_S4 = 0;
    int len_weight_C5 = 0;
    int len_bias_C5 = 0;
    int len_weight_output = 0;
    int len_bias_output = 0;

    int num_neuron_input = 0;
    int num_neuron_C1 = 0;
    int num_neuron_S2 = 0;
    int num_neuron_C3 = 0;
    int num_neuron_S4 = 0;
    int num_neuron_C5 = 0;
    int num_neuron_output = 0;

    fread(&width_image_input, sizeof(int), 1, fp);
    fread(&height_image_input, sizeof(int), 1, fp);
    fread(&width_image_C1, sizeof(int), 1, fp);
    fread(&height_image_C1, sizeof(int), 1, fp);
    fread(&width_image_S2, sizeof(int), 1, fp);
    fread(&height_image_S2, sizeof(int), 1, fp);
    fread(&width_image_C3, sizeof(int), 1, fp);
    fread(&height_image_C3, sizeof(int), 1, fp);
    fread(&width_image_S4, sizeof(int), 1, fp);
    fread(&height_image_S4, sizeof(int), 1, fp);
    fread(&width_image_C5, sizeof(int), 1, fp);
    fread(&height_image_C5, sizeof(int), 1, fp);
    fread(&width_image_output, sizeof(int), 1, fp);
    fread(&height_image_output, sizeof(int), 1, fp);

    fread(&width_kernel_conv, sizeof(int), 1, fp);
    fread(&height_kernel_conv, sizeof(int), 1, fp);
    fread(&width_kernel_pooling, sizeof(int), 1, fp);
    fread(&height_kernel_pooling, sizeof(int), 1, fp);

    fread(&num_map_input, sizeof(int), 1, fp);
    fread(&num_map_C1, sizeof(int), 1, fp);
    fread(&num_map_S2, sizeof(int), 1, fp);
    fread(&num_map_C3, sizeof(int), 1, fp);
    fread(&num_map_S4, sizeof(int), 1, fp);
    fread(&num_map_C5, sizeof(int), 1, fp);
    fread(&num_map_output, sizeof(int), 1, fp);

    fread(&len_weight_C1, sizeof(int), 1, fp);
    fread(&len_bias_C1, sizeof(int), 1, fp);
    fread(&len_weight_S2, sizeof(int), 1, fp);
    fread(&len_bias_S2, sizeof(int), 1, fp);
    fread(&len_weight_C3, sizeof(int), 1, fp);
    fread(&len_bias_C3, sizeof(int), 1, fp);
    fread(&len_weight_S4, sizeof(int), 1, fp);
    fread(&len_bias_S4, sizeof(int), 1, fp);
    fread(&len_weight_C5, sizeof(int), 1, fp);
    fread(&len_bias_C5, sizeof(int), 1, fp);
    fread(&len_weight_output, sizeof(int), 1, fp);
    fread(&len_bias_output, sizeof(int), 1, fp);

    fread(&num_neuron_input, sizeof(int), 1, fp);
    fread(&num_neuron_C1, sizeof(int), 1, fp);
    fread(&num_neuron_S2, sizeof(int), 1, fp);
    fread(&num_neuron_C3, sizeof(int), 1, fp);
    fread(&num_neuron_S4, sizeof(int), 1, fp);
    fread(&num_neuron_C5, sizeof(int), 1, fp);
    fread(&num_neuron_output, sizeof(int), 1, fp);

```

关闭

```
    fread(weight_C1, sizeof(weight_C1), 1, fp);
    fread(bias_C1, sizeof(bias_C1), 1, fp);
    fread(weight_S2, sizeof(weight_S2), 1, fp);
    fread(bias_S2, sizeof(bias_S2), 1, fp);
    fread(weight_C3, sizeof(weight_C3), 1, fp);
    fread(bias_C3, sizeof(bias_C3), 1, fp);
    fread(weight_S4, sizeof(weight_S4), 1, fp);
    fread(bias_S4, sizeof(bias_S4), 1, fp);
    fread(weight_C5, sizeof(weight_C5), 1, fp);
    fread(bias_C5, sizeof(bias_C5), 1, fp);
    fread(weight_output, sizeof(weight_output), 1, fp);
    fread(bias_output, sizeof(bias_output), 1, fp);

    fflush(fp);
    fclose(fp);

    out2wi_S2.clear();
    out2bias_S2.clear();
    out2wi_S4.clear();
    out2bias_S4.clear();

    calc_out2wi(width_image_C1_CNN, height_image_C1_CNN, width_image_S2_CNN, height_image_
    calc_out2bias(width_image_S2_CNN, height_image_S2_CNN, num_map_S2_CNN, out2bias_S2);
    calc_out2wi(width_image_C3_CNN, height_image_C3_CNN, width_image_S4_CNN,
    calc_out2bias(width_image_S4_CNN, height_image_S4_CNN, num_map_S4_CNN, out2bias_S4);

    return true;
}

bool CNN::saveModelFile(const char* name)
{
    FILE* fp = fopen(name, "wb");
    if (fp == NULL) {
        return false;
    }

    int width_image_input = width_image_input_CNN;
    int height_image_input = height_image_input_CNN;
    int width_image_C1 = width_image_C1_CNN;
    int height_image_C1 = height_image_C1_CNN;
    int width_image_S2 = width_image_S2_CNN;
    int height_image_S2 = height_image_S2_CNN;
    int width_image_C3 = width_image_C3_CNN;
    int height_image_C3 = height_image_C3_CNN;
    int width_image_S4 = width_image_S4_CNN;
    int height_image_S4 = height_image_S4_CNN;
    int width_image_C5 = width_image_C5_CNN;
    int height_image_C5 = height_image_C5_CNN;
    int width_image_output = width_image_output_CNN;
    int height_image_output = height_image_output_CNN;

    int width_kernel_conv = width_kernel_conv_CNN;
    int height_kernel_conv = height_kernel_conv_CNN;
    int width_kernel_pooling = width_kernel_pooling_CNN;
    int height_kernel_pooling = height_kernel_pooling_CNN;

    int num_map_input = num_map_input_CNN;
    int num_map_C1 = num_map_C1_CNN;
    int num_map_S2 = num_map_S2_CNN;
    int num_map_C3 = num_map_C3_CNN;
    int num_map_S4 = num_map_S4_CNN;
    int num_map_C5 = num_map_C5_CNN;
    int num_map_output = num_map_output_CNN;

    int len_weight_C1 = len_weight_C1_CNN;
    int len_bias_C1 = len_bias_C1_CNN;
    int len_weight_S2 = len_weight_S2_CNN;
    int len_bias_S2 = len_bias_S2_CNN;
    int len_weight_C3 = len_weight_C3_CNN;
    int len_bias_C3 = len_bias_C3_CNN;
    int len_weight_S4 = len_weight_S4_CNN;
    int len_bias_S4 = len_bias_S4_CNN;
    int len_weight_C5 = len_weight_C5_CNN;
    int len_bias_C5 = len_bias_C5_CNN;
    int len_weight_output = len_weight_output_CNN;
    int len_bias_output = len_bias_output_CNN;

    int num_neuron_input = num_neuron_input_CNN;
    int num_neuron_C1 = num_neuron_C1_CNN;
```

关闭

```
    int num_neuron_S2 = num_neuron_S2_CNN;
    int num_neuron_C3 = num_neuron_C3_CNN;
    int num_neuron_S4 = num_neuron_S4_CNN;
    int num_neuron_C5 = num_neuron_C5_CNN;
    int num_neuron_output = num_neuron_output_CNN;

    fwrite(&width_image_input, sizeof(int), 1, fp);
    fwrite(&height_image_input, sizeof(int), 1, fp);
    fwrite(&width_image_C1, sizeof(int), 1, fp);
    fwrite(&height_image_C1, sizeof(int), 1, fp);
    fwrite(&width_image_S2, sizeof(int), 1, fp);
    fwrite(&height_image_S2, sizeof(int), 1, fp);
    fwrite(&width_image_C3, sizeof(int), 1, fp);
    fwrite(&height_image_C3, sizeof(int), 1, fp);
    fwrite(&width_image_S4, sizeof(int), 1, fp);
    fwrite(&height_image_S4, sizeof(int), 1, fp);
    fwrite(&width_image_C5, sizeof(int), 1, fp);
    fwrite(&height_image_C5, sizeof(int), 1, fp);
    fwrite(&width_image_output, sizeof(int), 1, fp);
    fwrite(&height_image_output, sizeof(int), 1, fp);

    fwrite(&width_kernel_conv, sizeof(int), 1, fp);
    fwrite(&height_kernel_conv, sizeof(int), 1, fp);
    fwrite(&width_kernel_pooling, sizeof(int), 1, fp);
    fwrite(&height_kernel_pooling, sizeof(int), 1, fp);

    fwrite(&num_map_input, sizeof(int), 1, fp);
    fwrite(&num_map_C1, sizeof(int), 1, fp);
    fwrite(&num_map_S2, sizeof(int), 1, fp);
    fwrite(&num_map_C3, sizeof(int), 1, fp);
    fwrite(&num_map_S4, sizeof(int), 1, fp);
    fwrite(&num_map_C5, sizeof(int), 1, fp);
    fwrite(&num_map_output, sizeof(int), 1, fp);

    fwrite(&len_weight_C1, sizeof(int), 1, fp);
    fwrite(&len_bias_C1, sizeof(int), 1, fp);
    fwrite(&len_weight_S2, sizeof(int), 1, fp);
    fwrite(&len_bias_S2, sizeof(int), 1, fp);
    fwrite(&len_weight_C3, sizeof(int), 1, fp);
    fwrite(&len_bias_C3, sizeof(int), 1, fp);
    fwrite(&len_weight_S4, sizeof(int), 1, fp);
    fwrite(&len_bias_S4, sizeof(int), 1, fp);
    fwrite(&len_weight_C5, sizeof(int), 1, fp);
    fwrite(&len_bias_C5, sizeof(int), 1, fp);
    fwrite(&len_weight_output, sizeof(int), 1, fp);
    fwrite(&len_bias_output, sizeof(int), 1, fp);

    fwrite(&num_neuron_input, sizeof(int), 1, fp);
    fwrite(&num_neuron_C1, sizeof(int), 1, fp);
    fwrite(&num_neuron_S2, sizeof(int), 1, fp);
    fwrite(&num_neuron_C3, sizeof(int), 1, fp);
    fwrite(&num_neuron_S4, sizeof(int), 1, fp);
    fwrite(&num_neuron_C5, sizeof(int), 1, fp);
    fwrite(&num_neuron_output, sizeof(int), 1, fp);

    fwrite(weight_C1, sizeof(weight_C1), 1, fp);
    fwrite(bias_C1, sizeof(bias_C1), 1, fp);
    fwrite(weight_S2, sizeof(weight_S2), 1, fp);
    fwrite(bias_S2, sizeof(bias_S2), 1, fp);
    fwrite(weight_C3, sizeof(weight_C3), 1, fp);
    fwrite(bias_C3, sizeof(bias_C3), 1, fp);
    fwrite(weight_S4, sizeof(weight_S4), 1, fp);
    fwrite(bias_S4, sizeof(bias_S4), 1, fp);
    fwrite(weight_C5, sizeof(weight_C5), 1, fp);
    fwrite(bias_C5, sizeof(bias_C5), 1, fp);
    fwrite(weight_output, sizeof(weight_output), 1, f|
    fwrite(bias_output, sizeof(bias_output), 1, fp);

    fflush(fp);
    fclose(fp);

    return true;
}

double CNN::test()
{
    int count_accuracy = 0;

    for (int num = 0; num < num_patterns_test_CNN; num++) {
```

关闭

```
            data_single_image = data_input_test + num * num_neuron_input_CNN;
            data_single_label = data_output_test + num * num_neuron_output_CNN;

            Forward_C1();
            Forward_S2();
            Forward_C3();
            Forward_S4();
            Forward_C5();
            Forward_output();

            int pos_t = -1;
            int pos_y = -2;
            double max_value_t = -9999.0;
            double max_value_y = -9999.0;

            for (int i = 0; i < num_neuron_output_CNN; i++) {
                if (neuron_output[i] > max_value_y) {
                    max_value_y = neuron_output[i];
                    pos_y = i;
                }

                if (data_single_label[i] > max_value_t) {
                    max_value_t = data_single_label[i];
                    pos_t = i;
                }
            }

            if (pos_y == pos_t) {
                ++count_accuracy;
            }

            Sleep(1);
        }

        return (count_accuracy * 1.0 / num_patterns_test_CNN);
    }

}
```

测试代码如下：

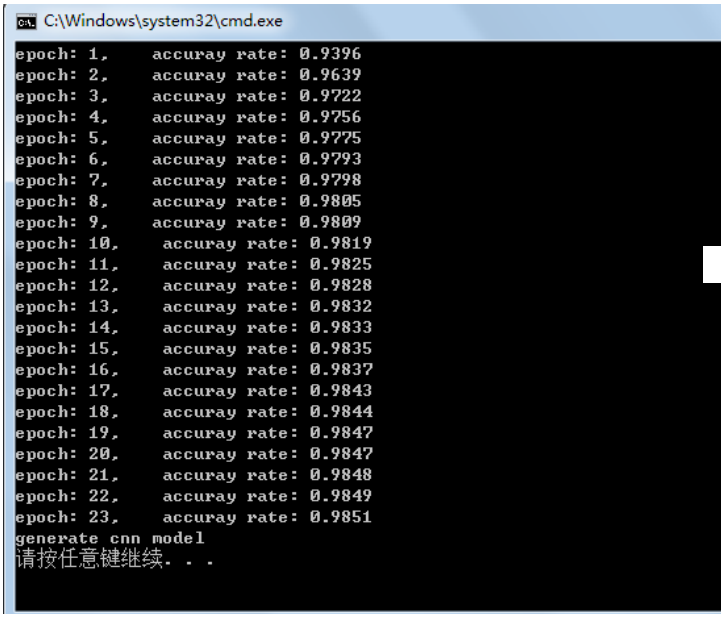
[cpp]

```
int test_CNN_train()
{
    ANN::CNN cnn1;
    cnn1.init();
    cnn1.train();

    return 0;
}

int test_CNN_predict()
{
    ANN::CNN cnn2;
    bool flag = cnn2.readModelFile("E:/GitCode/NN_Test/data/cnn.model");
    if (!flag) {
        std::cout << "read cnn model error" << std::endl;
        return -1;
    }

    int width{ 32 }, height{ 32 };
    std::vector<int> target{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
    std::string image_path{ "E:/GitCode/NN_Test/data/images/" };

    for (auto i : target) {
        std::string str = std::to_string(i);
        str += ".png";
        str = image_path + str;

        cv::Mat src = cv::imread(str, 0);
        if (src.data == nullptr) {
            fprintf(stderr, "read image error: %s\n", str.c_str());
            return -1;
        }

        cv::Mat tmp(src.rows, src.cols, CV_8UC1, cv::Scalar::all(255));
        cv::subtract(tmp, src, tmp);

```

关闭

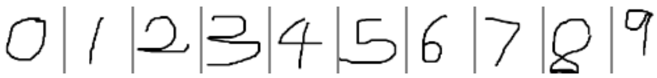
```
            cv::resize(tmp, tmp, cv::Size(width, height));

            auto ret = cnn2.predict(tmp.data, width, height);

            fprintf(stdout, "the actual digit is: %d, correct digit is: %d\n", ret, i);
        }

        return 0;
}
```

通过执行test_CNN_train()函数可生成cnn model文件，执行结果如下：

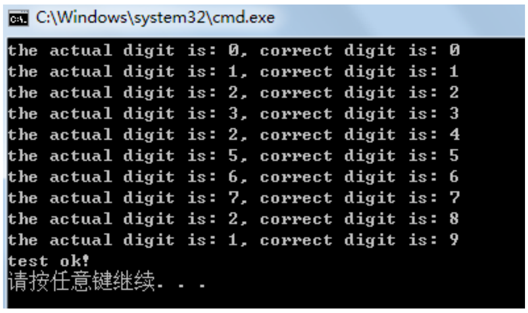


```
C:\Windows\system32\cmd.exe
epoch: 1,    accuray rate: 0.9396
epoch: 2,    accuray rate: 0.9639
epoch: 3,    accuray rate: 0.9722
epoch: 4,    accuray rate: 0.9756
epoch: 5,    accuray rate: 0.9775
epoch: 6,    accuray rate: 0.9793
epoch: 7,    accuray rate: 0.9798
epoch: 8,    accuray rate: 0.9805
epoch: 9,    accuray rate: 0.9809
epoch: 10,    accuray rate: 0.9819
epoch: 11,    accuray rate: 0.9825
epoch: 12,    accuray rate: 0.9828
epoch: 13,    accuray rate: 0.9832
epoch: 14,    accuray rate: 0.9833
epoch: 15,    accuray rate: 0.9835
epoch: 16,    accuray rate: 0.9837
epoch: 17,    accuray rate: 0.9843
epoch: 18,    accuray rate: 0.9844
epoch: 19,    accuray rate: 0.9847
epoch: 20,    accuray rate: 0.9847
epoch: 21,    accuray rate: 0.9848
epoch: 22,    accuray rate: 0.9849
epoch: 23,    accuray rate: 0.9851
generate cnn model
请按任意键继续. . .
```

通过执行test_CNN_predict()函数来测试CNN的准确率，通过画图工具，每个数字生成一张图像，共10幅，如下图：



测试结果如下：



```
C:\Windows\system32\cmd.exe
the actual digit is: 0, correct digit is: 0
the actual digit is: 1, correct digit is: 1
the actual digit is: 2, correct digit is: 2
the actual digit is: 3, correct digit is: 3
the actual digit is: 2, correct digit is: 4
the actual digit is: 5, correct digit is: 5
the actual digit is: 6, correct digit is: 6
the actual digit is: 7, correct digit is: 7
the actual digit is: 2, correct digit is: 8
the actual digit is: 1, correct digit is: 9
test ok!
请按任意键继续. . .
```

代码实现解析见：http://blog.csdn.net/fengbingchun/article/details/53445209

GitHub：**https://github.com/fengbingchun/NN**

关闭

顶 3

踩 0

下一篇 [illegible]

我的同类文章

Caffe（19） Deep Learning（8） Neural Network（12）

- Caffe中Layer注册机制 2017-01-10 阅读 87
- windows7下解决caffe check… 2017-01-09 阅读 121
- cifar数据集介绍及到图像转… 2016-12-10 阅读 245
- 深度学习开源库tiny-dnn的使… 2016-12-04 阅读 465
- 卷积神经网络(CNN)代码实… 2016-12-03 阅读 2188
- 一步一步指引你在Windows… 2016-03-26 阅读 1381
- Windows7 64bit VS2013 Ca… 2016-03-26 阅读 1550
- tiny-cnn执行过程分析(MNIST) 2016-01-31 阅读 3557
- tiny-cnn开源库的使用(MNIST) 2016-01-24 阅读 9464
- 卷积神经网络(CNN)基础介绍 2016-01-16 阅读 11361

更多文章

广告

JD.COM 京东

猜你在找

《C语言/C++学习指南》数据库篇(MySQL& sqlite)
Deep Learning模型之CNN的反向求导及练习
C++ 单元测试（GoogleTest）
CNN
Swift与Objective-C\C\C++混合编程
CNN
C/C++单元测试培训
深度学习DL与卷积神经网络CNN学习笔记随笔-03-基于
TCP/IP/UDP Socket通讯开发实战 适合iOS/Android/Lin
深度学习DL与卷积神经网络CNN学习笔记随笔-03-基于

app开发报价单 短信接口 一元手机 云服务器免费 图书馆管理系统 免费云服务器

查看评论

8楼 hugl950123 5天前 16:59发表

博主，请问我按照您的代码成功编译后执行结果窗口一闪而过，并且里面什么内容也没有，应该如何解决，能不能帮帮忙=-=

Re: fengbingchun 5天前 18:05发表

回复hugl950123：你用的是GitHub上最新的吗？既然能编译过，在Debug下设断点，应该很快能定位到问题原因

Re: hugl950123 5天前 20:24发表

回复fengbingchun：下的是新的，我在CNN.cpp文件中每个函数都设置了断点，还是没有变化=-=执行结果的窗口还是一闪而过并且里面什么都没有，是我设置断点的方法不对么。。。还有想请教一下现在好多tiny_cnn算法代码的GitHub地址都链接到了tiny_dnn算法，是没法看到原来的tiny_cnn代码了么

Re: fengbingchun 5天前 21:13发表

回复hugl950123：NN中一共有四个工程，它们之间没有任何关系，都是独立的，如果要运行这篇文章的代码，只需选中NN工程，编译运行它即可。

Re: hugl950123 前天 23:52发表

回复fengbingchun：博主请问一下，test_CNN_predict()函数是不是需要opencv的支持，为什么我加上了#include <opencv2/opencv.hpp>J

关闭

Re: fengbingchun 昨天 08:35发表

回复hugl950123：是需要opencv的支持，你在本地opencv的环境配好了吗，配好了就应该没问题了

Re: hugl950123 前天 23:51发表

回复fengbingchun：博主请问一下，test_CNN_predict()函数是不是需要opencv的支持，为什么我加上了#include <opencv2/opencv.hpp>后会出现error LNK2019的错误呢=-=

Re: hugl950123 前天 09:06发表

回复fengbingchun：谢谢，能够成功运行了现在

7楼 guanzheng9996 2016-11-26 15:51发表

博主，请问这个在什么环境下运行呢？除了vs2013还需要配置什么，还有就是运行出来的结果是什么样子的呢，我是个新手，麻烦博主指点

Re: fengbingchun 2016-11-26 17:06发表

回复guanzheng9996：不需要配置什么，结果于http://blog.csdn.net/fengbingchun/article/details/50573841 中结果类似，这个还有个bug，后面会把修改后的代码放上去。

Re: guanzheng9996 2016-11-26 17:14发表

回复fengbingchun：这个和seetaface比，哪个要好，seetaface可以用自己的照片进行训练么？

Re: fengbingchun 2016-11-26 18:35发表

回复guanzheng9996：好像seetaface还没有提供训练的代码

6楼 guanzheng9996 2016-11-26 15:51发表

博主，请问这个在什么环境下运行呢？除了vs2013还需要配置什么，还有就是运行出来的结果是什么样子的呢，我是个新手，麻烦博主指点

5楼 VR_LFB 2016-08-12 17:43发表

万分感谢楼主贴出如此细致的代码！我尝试修改了UpdateWeights()：对其中的梯度向量先做了normalization。而后accuracy就能达到0.97以上了。

Re: fengbingchun 2016-08-13 18:01发表

回复VR_LFB：赞

4楼 visionfans 2016-05-22 00:01发表

博主一般是怎么找这样隐藏的很深，很难查出来的bug的？

Re: fengbingchun 2016-05-22 10:47发表

回复visionfans：感觉没有什么好方法吧，就是多打log，逐函数打印输出结果，看再哪个函数内出的问题

3楼 fpthink 2016-03-30 22:34发表

博主，我看了你的代码，想请教你一些问题，代码中的和文字描述有不同的地方。关于阈值和权值，的初始值设定。

Re: fengbingchun 2016-03-31 08:15发表

回复fp1527323876：是有些不同的地方，主要是实现完后，发现识别率一直上不去，就仿照tiny-cnn的改写了下，识别率还是很低，现在还是有些bug。

Re: fpthink 2016-03-31 09:19发表

回复fengbingchun：s2到c3的convolution，6到16，有一个映射关系，为了好写，直接用16*6，c3层和每一个s2（6*14*14）层的convolution，

```
//0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
{1, 0, 0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 1, 0, 1, 1},
{1, 1, 0, 0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 1, 0, 1},
{1, 1, 1, 0, 0, 0, 1, 1, 1, 0, 0, 1, 0, 1, 1, 1},
{0, 1, 1, 1, 0, 0, 1, 1, 1, 1, 0, 0, 1, 0, 1, 1},
{0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 1, 0, 1, 1, 0, 1},
{0, 0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 1, 0, 1, 1, 1}
```

关闭

Re: fpthink 2016-03-31 09:10发表

回复fengbingchun：比如s4和c5的全连接，用120个卷积核和每一个s4（16*5*5）层的做卷积，最后用激活函数激活c5

2楼 ccjava5188 2016-03-26 22:48发表

麻烦问一下博主：“C3层：卷积窗大小5*5，输出特征图数量16，卷积窗种类6*16=96”，卷积窗的种类为啥是96个，输出特征图数量为16？

Re: fengbingchun 2016-03-27 10:35发表

回复ccjava5188：特征图数量可以根据实际需要由自己定。仿照LeNet-5结构，对于C3层，有16个特征map，C3中

每个特征图由S2中所有6个或者几个特征map组合而成，如果由S2所有6个特征map组合而成，那么卷积窗种类就是16*6了，这里为了实现方便，没有完全按照原有的LeNet-5结构实现。你可以参考下http://blog.csdn.net/fengbingchun/article/details/50529500

1楼 吴士龙 2016-03-10 18:48发表

虽然看不太懂，但是呢，楼主很细致呢。

您还没有登录,请[登录]或[注册]

* 以上用户言论只代表其个人观点，不代表CSDN网站的观点或立场

核心技术类目

全部主题 Hadoop AWS 移动游戏 Java Android iOS Swift 智能硬件 Docker OpenStack VPN Spark ERP IE10 Eclipse CRM JavaScript 数据库 Ubuntu NFC WAP jQuery BI HTML5 Spring Apache .NET API HTML SDK IIS Fedora XML LBS Unity Splashtop UML components Windows Mobile Rails QEMU KDE Cassandra CloudStack FTC coremail OPhone CouchBase 云计算 iOS6 Rackspace Web App SpringSide Maemo Compuware 大数据 aptech Perl Tornado Ruby Hibernate ThinkPHP HBase Pure Solr Angular Cloud Foundry Redis Scala Django Bootstrap

公司简介 | 招贤纳士 | 广告服务 | 联系方式 | 版权声明 | 法律顾问 | 问题报告 | 合作伙伴 | 论坛反馈

网站客服 杂志客服 微博客服 webmaster@csdn.net 400-600-2320 | 北京创新乐知信息技术有限公司 版权所有 | 江苏知之为计算机有限公司 | 江苏乐知网络技术有限公司

京 ICP 证 09002463 号 | Copyright © 1999-2016, CSDN.NET, All Rights Reserved

关闭