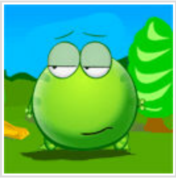
网络资源是无限的

个人资料



fengbingchun

卷积神经网络(CNN)代码实现(MNIST)解析

2016-12-03 16:09 2194人阅读 评论(0) 收藏 举报

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

在http://blog.csdn.net/fengbingchun/article/details/50814710中给出了CNN的简单实现，这里对每一步的实现作个说明：

共7层：依次为输入层、C1层、S2层、C3层、S4层、C5层、输出层，C代表卷积层(特征提取)，S代表降采样层或池化层(Pooling)，输出层为全连接层。

1. 各层权值、偏置(阈值)初始化：

各层权值、偏置个数计算如下：

(1)、输入层：预处理后的32*32图像数据，无权值和偏置；

(2)、C1层：卷积窗大小5*5，输出特征图数量6，卷积窗种类1*6=6，输出特征图大小28*28，因此可训练参数(权值+偏置)：(5*5*1)*6+6=150+6；

(3)、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，因此可训练参数(权值+偏置)：1*6+6=6+6；

(4)、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，因此可训练参数(权值+偏置)：(5*5*6)*16+16=2400+16；

(5)、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，因此可训练参数(权值+偏置)：1*16+16=16+16；

(6)、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，因此可训练参数(权值+偏置)：(5*5*16)*120+120=48000+120；

(7)、输出层：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，因此可训练参数(权值+偏置)：(1*120)*10+10=1200+10.

代码段如下：

[cpp]

```cpp
#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 len_weight_C1_CNN       150 //C1层权值数，(5*5*1)*6=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*6)*16=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，(1*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)*1=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*1)*120=120
#define num_neuron_output_CNN   10 //输出层神经元数，(1*1)*10=10
```

权值、偏置初始化：

(1)、权值使用函数uniform_real_distribution均匀分布初始化，tiny-cnn中每次初始化权值数值都相同，这里作了调整，使每次初始化的权值均不同。每层权值初始化大小范围都不一样；

(2)、所有层的偏置均初始化为0.

代码段如下：

[cpp]

```cpp
double CNN::uniform_rand(double min, double max)
{
    //static std::mt19937 gen(1);
    std::random_device rd;
    std::mt19937 gen(rd());
    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++) {
```

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

```
        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;
}
```

2. 加载MNIST数据：

关于MNIST的介绍可以参考：http://blog.csdn.net/fengbingchun/article/details/49611549

使用MNIST库作为训练集和测试集，训练样本集为60000个，测试样本集为10000个。

(1)、MNIST库中图像原始大小为28*28，这里缩放为32*32，数据取值范围为[-1,1]，扩充值均取-1，作为输入层输入数据。

代码段如下：

[cpp]

```cpp
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] =
            }
        }
    }
}
```

关闭

(2)、对于Label,输出层有10个节点，对应位置的节点值设为0.8，其它节点设为-0.8，作为输出层数据。

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位机上不	(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...

代码段如下：

[cpp]

```cpp
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;
    }
}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;
    }
}
```

3. 前向传播：主要计算每层的神经元值；其中C1层、C3层、C5层操作过程相同；S2层、S4层操作过程相同。

(1)、输入层：神经元数为(32*32)*1=1024。

(2)、C1层：神经元数为(28*28)*6=4704，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。

激活函数的作用：它是用来加入非线性因素的，解决线性模型所不能解决的问题，提供网络的非线性建模能力。如果没有激活函数，那么该网络仅能够表达线性映射，此时即便有再多的隐藏层，其整个网络跟单层神经网络也是等价的。因此也可以认为，只有加入了激活函数之后，深度神经网络才具备了分层的非线性映射学习能力。

代码段如下：

[cpp]

关闭

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

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

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

```

```cpp
    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_C
        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;
}
```

(3)、S2层：神经元数为(14*14)*6=1176，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乘以同一个权值再进行相加求和，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。

代码段如下：

[cpp]

```cpp
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);

    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(
            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;
}
```

(4)、C3层：神经元数为(10*10)*16=1600，C3层实现方式与C1层完全相同，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。

也可按照Y.Lecun给出的表进行计算，即对于生成的每一个10*10的特征图，是由n个5*5的卷积图像去乘以n个14*14的下采样图，其中n是小于6的，即不完全连接。这样做的原因：第一，不完全的连接机制将连接的数量保持在合理的范围内。第二，也是最重要的，其破坏了网络的对称性。由于不同的特征图有不同的输入，所以迫使他们抽取不同的特征。

代码段如下：

[cpp]

```cpp
// 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

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_
        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;
```

关闭

```
}
```

(5)、S4层：神经元数为(5*5)*16=400，S4层实现方式与S2层完全相同，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乘以同一个权值再进行相加求和，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。

代码段如下：

[cpp]

```cpp
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++) {
        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[i
        }

        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;
}
```

(6)、C5层：神经元数为(1*1)*120=120，也可看为全连接层，C5层实现方式与C1、C3层完全相同，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。

代码段如下：

[cpp]

```cpp
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
                    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;
                }
            }
        }
```

关闭

```

        int addr3 = get_index(0, 0, o, width_image_C5_CNN, height_image_C5_CNN, num_map_C!
        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;
}
```

(7)、输出层：神经元数为(1*1)*10=10，为全连接层，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。

代码段如下：

[cpp]

```cpp
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[(
        }

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

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

    return true;
}
```

4. 反向传播：主要计算每层权值和偏置的误差以及每层神经元的误差；其中输入层、S2层、S4层操作过程相同；C1层、C3层操作过程相同。

(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，即a、已计算出的输出层神经元值减去对应label值，b、1.0减去输出层神经元值的平方，c、a与c的乘积和。

损失函数作用：在统计学中损失函数是一种衡量损失和错误(这种损失与"错误地"估计有关)程度的函数。损失函数在实践中最重要的运用，在于协助我们通过过程的改善而持续减少目标值的变异，并非仅仅追求符合逻辑。在深度学习中，对于损失函数的收敛特性，我们期望是当误差越大的时候，收敛(学习)速度应该越快。成为损失函数需要满足两点要求：非负性；预测值和期望值接近时，函数值趋于0.

代码段如下：

关闭

[cpp]

```cpp
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]);
    }
}
```

```cpp

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

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::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_neuro
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_CNN);
    }

    return true;
}
```

(2)、C5层：计算C5层神经元误差、输出层权值误差、输出层偏置误差；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(即1-C5层神经元值的平方)，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；输出层偏置误差即为输出层神经元误差。

代码段如下：

[cpp]

```cpp
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;
}

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 *
        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;
}
```

(3)、S4层：计算S4层神经元误差、C5层权值误差、C5层偏置误差；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(即1-S4神经元的平方)，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层偏置误差即为C5层神经元误差。

代码段如下：

[cpp]

```cpp
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_CNN * outc + inc, width_ke
            int addr2 = get_index(0, 0, outc, width_image_C5_CNN, height_image_C5_CNN, num
            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;

                    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
                    }

                    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_ma
        const double* delta = &delta_neuron_C5[0] + addr2;

        for (int y = 0; y < height_image_C5_CNN; y++) {
```

关闭

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

    return true;
}
```

(4)、C3层：计算C3层神经元误差、S4层权值误差、S4层偏置误差；通过S4层权值乘以S4层神经元误差，求和，结果再乘以C3层神经元的tanh激活函数的导数(即1-S4神经元的平方)，然后再乘以1/4，获得C3层每一个神经元误差；通过C3层神经元乘以S4神经元误差，求和，再乘以1/4，获得S4层权值误差；通过S4层神经元误差求和，来获得S4层偏置误差。

代码段如下：

[cpp]

```cpp
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[i];
        double delta = 0.0;

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

        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;

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

        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;
}
```


(5)、S2层：计算S2层神经元误差、C3层权值误差、C3层偏置误差；通过C3层权值乘以C3层神经元误差，求和，结果再乘以S2层神经元的tanh激活函数的导数(即1-S2神经元的平方)，获得S2层每一个神经元误差；通过S2层神经元乘以C3层神经元误差，求和，获得C3层权值误差；C3层偏置误差即为C3层神经元误差和。

代码段如下：

[cpp]

```cpp
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_CNN; wy++) {
                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
                    }

                    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] + a

        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;
}
```

关闭

(6)、C1层：计算C1层神经元误差、S2层权值误差、S2层偏置误差；通过S2层权值乘以S2层神经元误差，求

和，结果再乘以C1层神经元的tanh激活函数的导数(即1-C1神经元的平方)，然后再乘以1/4，获得C1层每一个神经元误差；通过C1层神经元乘以S2神经元误差，求和，再乘以1/4，获得S2层权值误差；通过S2层神经元误差求和，来获得S4层偏置误差。

代码段如下：

[cpp]

```cpp
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].secon
        }

        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;
}
```

(7)、输入层：计算输入层神经元误差、C1层权值误差、C1层偏置误差；通过C1层权值乘以C1层神经元误差，求和，结果再乘以输入层神经元的tanh激活函数的导数(即1-输入层神经元的平方)，获得输入层每一个神经元误差；通过输入层层神经元乘以C1层神经元误差，求和，获得C1层权值误差；C1层偏置误差即为C1层神经元误差和。

[cpp]

```cpp
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_
            int addr2 = get_index(0, 0, outc, width_image_C1_CNN, height_image_C1_CNN, nu
            int addr3 = get_index(0, 0, inc, width_image_input_CNN, height_image_input_CN

            const double* pw = &weight_C1[0] + addr1;
```

关闭

```cpp
            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;
}
```

5. 更新各层权值、偏置：通过之前计算的各层权值、各层权值误差；各层偏置、各层偏置误差以及学习率来更新各层权值和偏置。

代码段如下：

[cpp]

```cpp
void CNN::update_weights_bias(const double* delta, double* e_weight, double* weight, int 
{
    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()
{
```

```cpp
    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);
    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;
}
```

6. 测试准确率是否达到要求或已达到循环次数：依次循环3至5中操作，根据训练集数量，每循环60000次时，通过计算的权值和偏置，来对10000个测试集进行测试，如果准确率达到0.985或者达到迭代次数时，保存权值和偏置。

代码段如下：

[cpp]

```cpp
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();
        }
```

关闭

```cpp

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

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

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

7. 对输入的图像数据进行识别：载入已保存的权值和偏置，对输入的数据进行识别，过程相当于前向传播。

代码段如下：

[cpp]

```cpp
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;
}
```

GitHub：https://github.com/fengbingchun/NN_Test

上一篇 Dlib简介及在windows7 vs2013编译过程

下一篇 深度学习开源库tiny-dnn的使用(MNIST)

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
- 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
- 卷积神经网络(CNN)的简单... 2016-03-06 阅读 7518
- 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

查看评论

暂无评论

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