


C++从零实现神经网络 (收藏版：两万字长文)

小白学视觉 2021-12-03 10:05

以下文章来源于CVPy，作者冰不语



CVPy
人脸识别、猫脸识别、C++实现神经网络、Python给头像加圣诞帽，OpenCV解九宫格...

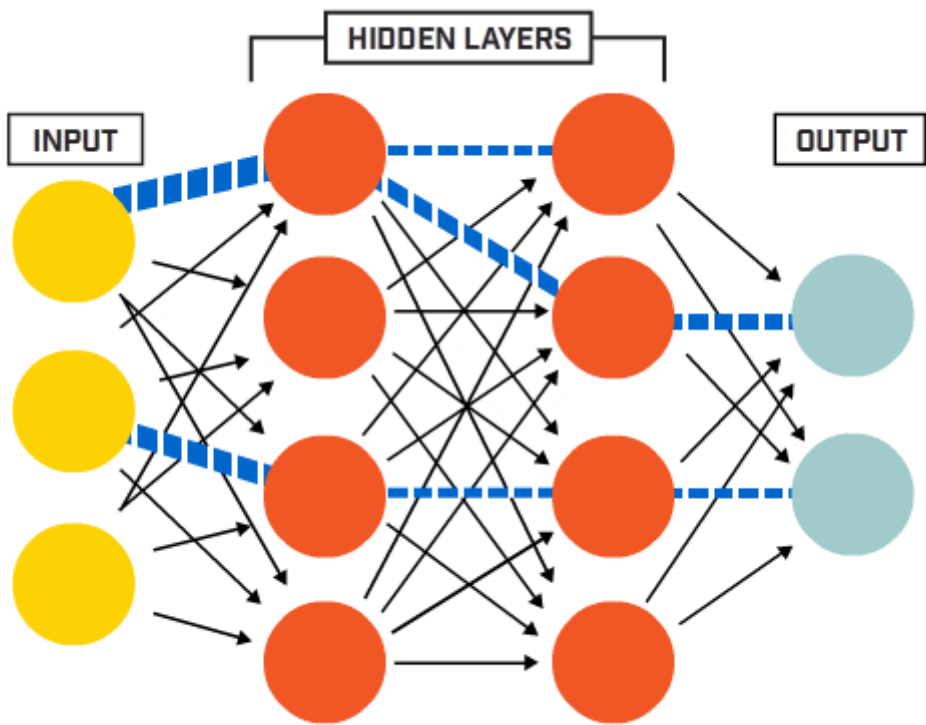
>

点击上方“**小白学视觉**”，选择加“**星标**”或“**置顶**”
重磅干货，第一时间送达

一、Net类的设计与神经网络初始化

闲言少叙，直接开始

既然是要用C++来实现，那么我们自然而然的想到设计一个神经网络类来表示神经网络，这里我称之为Net类。由于这个类名太过普遍，很有可能跟其他人写的程序冲突，所以我的所有程序都包含在namespace liu中，由此不难想到我姓刘。在之前的博客反向传播算法资源整理中，我列举了几个比较不错的资源。对于理论不熟悉而且学习精神的同学可以出门左转去看看这篇文章的资源。这里假设读者对于神经网络的基本理论有一定的了解。



神经网络的要素

在真正开始coding之前还是有必要交代一下神经网络基础，其实也就是设计类和写程序的思路。简而言之，神经网络的包含几大要素：

- 神经元节点
- 层 (layer)
- 权值 (weights)
- 偏置项 (bias)

神经网络的两大计算过程分别是前向传播和反向传播过程。每层的前向传播分别包含加权求和（卷积？）的线性运算和激活函数的非线性运算。反向传播主要是用BP算法更新权值。虽然里面还有很多细节，但是对于作为第一篇的本文来说，以上内容足够了。

Net类的设计

Net类——基于Mat

神经网络中的计算几乎都可以用矩阵计算的形式表示，这也是我用OpenCV的Mat类的原因之一，它提供了非常完善的、充分优化过的各种矩阵运算方法；另一个原因是我最熟悉的库就是OpenCV.....有很多比较好的库和框架在实现神经网络的时候会用很多类来表示不同的部分。比如Blob类表示数据，Layer类表示各种层，Optimizer类来表示各种优化算法。但是这里没那么复杂，主要还是能力有限，只用一个Net类表示神经网络。

还是直接让程序说话，Net类包含在Net.h中，大致如下。

```
#ifndef NET_H
#define NET_H
#endif // NET_H
#pragma once
#include <iostream>
#include<opencv2\core\core.hpp>
#include<opencv2\highgui\highgui.hpp>
//#include<iomanip>
#include"Function.h"
namespace liu
{
    class Net
    {
    public:
        std::vector<int> layer_neuron_num;
        std::vector<cv::Mat> layer;
        std::vector<cv::Mat> weights;
        std::vector<cv::Mat> bias;
    public:
        Net() {};
```

```

~Net() {};
//Initialize net:generate weights matrices、layer matrices and bias matrices
// bias default all zero
void initNet(std::vector<int> layer_neuron_num_);
//Initialise the weights matrices.
void initWeights(int type = 0, double a = 0., double b = 0.1);
//Initialise the bias matrices.
void initBias(cv::Scalar& bias);
//Forward
void forward();
//Forward
void backward();
protected:
//initialise the weight matrix.if type =0,Gaussian.else uniform.
void initWeight(cv::Mat &dst, int type, double a, double b);
//Activation function
cv::Mat activationFunction(cv::Mat &x, std::string func_type);
//Compute delta error
void deltaError();
//Update weights
void updateWeights();
};
}

```

说明

以上不是Net类的完整形态，只是对应于本文内容的一个简化版，简化之后看起来会更加清晰明了。

成员变量与成员函数

成员变量与成员函数

现在Net类只有四个成员变量，分别是：

- 每一层神经元数目 (*layer_neuron_num*)
- 层 (*layer*)
- 权值矩阵 (*weights*)
- 偏置项 (*bias*)

权值用矩阵表示就不用说了，需要说明的是，为了计算方便，这里每一层和偏置项也用Mat表示，每一层和偏置都用一个单列矩阵来表示。

Net类的成员函数除了默认的构造函数和析构函数，还有：

- *initNet()*：用来初始化神经网络
- *initWeights()*：初始化权值矩阵，调用*initWeight()*函数
- *initBias()*：初始化偏置项

- forward(): 执行前向运算, 包括线性运算和非线性激活, 同时计算误差
- backward(): 执行反向传播, 调用updateWeights()函数更新权值。

这些函数已经是神经网络程序核心中的核心。剩下的内容就是慢慢实现了, 实现的时候需要什么添加什么, 逢山开路, 遇河架桥。

神经网络初始化

initNet()函数

先说一下initNet()函数, 这个函数只接受一个参数——每一层神经元数目, 然后借此初始化神经网络。这里所谓初始化神经网络的含义是: 生成每一层的矩阵、每一个权值矩阵和每一个偏置矩阵。听起来很简单, 其实也很简单。

实现代码在Net.cpp中。

这里生成各种矩阵没啥难点, 唯一需要留心的是权值矩阵的行数和列数的确定。值得一提的是这里把权值默认全设为0。

```
//Initialize net
void Net::initNet(std::vector<int> layer_neuron_num_)
{
    layer_neuron_num = layer_neuron_num_;
    //Generate every layer.
    layer.resize(layer_neuron_num.size());
    for (int i = 0; i < layer.size(); i++)
    {
        layer[i].create(layer_neuron_num[i], 1, CV_32FC1);
    }
    std::cout << "Generate layers, successfully!" << std::endl;
    //Generate every weights matrix and bias
    weights.resize(layer.size() - 1);
    bias.resize(layer.size() - 1);
    for (int i = 0; i < (layer.size() - 1); ++i)
    {
        weights[i].create(layer[i + 1].rows, layer[i].rows, CV_32FC1);
        //bias[i].create(layer[i + 1].rows, 1, CV_32FC1);
        bias[i] = cv::Mat::zeros(layer[i + 1].rows, 1, CV_32FC1);
    }
    std::cout << "Generate weights matrices and bias, successfully!" << std::endl;
    std::cout << "Initialise Net, done!" << std::endl;
}
```

权值初始化

initWeight()函数

权值初始化函数initWeights()调用initWeight()函数，其实就是初始化一个和多个的区别。偏置初始化是给所有的偏置赋相同的值。这里用Scalar对象来给矩阵赋值。

```
//initialise the weights matrix.if type =0,Gaussian.else uniform.
void Net::initWeight(cv::Mat &dst, int type, double a, double b)
{
    if (type == 0)
    {
        randn(dst, a, b);
    }
    else
    {
        randu(dst, a, b);
    }
}
//initialise the weights matrix.
void Net::initWeights(int type, double a, double b)
{
    //Initialise weights cv::Matrices and bias
    for (int i = 0; i < weights.size(); ++i)
    {
        initWeight(weights[i], 0, 0., 0.1);
    }
}
```

偏置初始化是给所有的偏置赋相同的值。这里用Scalar对象来给矩阵赋值。

```
//Initialise the bias matrices.
void Net::initBias(cv::Scalar& bias_)
{
    for (int i = 0; i < bias.size(); i++)
    {
        bias[i] = bias_;
    }
}
```

至此，神经网络需要初始化的部分已经全部初始化完成了。

初始化测试

我们可以用下面的代码来初始化一个神经网络，虽然没有什么功能，但是至少可以测试下现在的代码是否有BUG：

```
#include "../include/Net.h"
//<opencv2\opencv.hpp>
using namespace std;
using namespace cv;
using namespace liu;
int main(int argc, char *argv[])
{
    //Set neuron number of every layer
    vector<int> layer_neuron_num = { 784,100,10 };
    // Initialise Net and weights
    Net net;
    net.initNet(layer_neuron_num);
    net.initWeights(0, 0., 0.01);
    net.initBias(Scalar(0.05));
    getchar();
    return 0;
}
```

亲测没有问题。

本文先到这里，前向传播和反向传播放在下一篇文章里面。

源码

源码链接

神经网络源码：https://github.com/LiuXiaolong19920720/simple_net

二、前向传播与反向传播

前言

前一篇文章C++实现神经网络之壹—Net类的设计和神经网络的初始化中，大部分还是比较简单的。因为最重要事情就是生成各种矩阵并初始化。神经网络中的重点和核心就是本文的内容——前

向和反向传播两大计算过程。每层的前向传播分别包含加权求和（卷积？）的线性运算和激活函数的非线性运算。反向传播主要是用BP算法更新权值。本文也分为两部分介绍。

前向过程

前向过程简介

如前所述，前向过程分为线性运算和非线性运算两部分。相对来说比较简单。

线性运算可以用 $Y = WX + b$ 来表示，其中X是输入样本，这里即是第N层的单列矩阵，W是权值矩阵，Y是加权求和之后的结果矩阵，大小与N+1层的单列矩阵相同。b是偏置，默认初始化全部为0。不难推知（鬼知道我推了多久！），W的大小是 $(N+1).rows * N.rows$ 。正如上一篇中生成weights矩阵的代码实现一样：

```
weights[i].create(layer[i + 1].rows, layer[i].rows, CV_32FC1);
```

非线性运算可以用 $O = f(Y)$ 来表示。Y就是上面得到的Y。O就是第N+1层的输出。f就是我们一直说的激活函数。激活函数一般都是非线性函数。它存在的价值就是给神经网络提供非线性建模能力。激活函数的种类有很多，比如sigmoid函数，tanh函数，ReLU函数等。各种函数的优缺点可以参考更为专业的论文和其他更为专业的资料。

我们可以先来看一下前向函数forward()的代码：

```
//Forward
void Net::forward()
{
    for (int i = 0; i < layer_neuron_num.size() - 1; ++i)
    {
        cv::Mat product = weights[i] * layer[i] + bias[i];
        layer[i + 1] = activationFunction(product, activation_function);
    }
}
```

for循环里面的两句就分别是上面说的线性运算和激活函数的非线性运算。

激活函数 activationFunction() 里面实现了不同种类的激活函数，可以通过第二个参数来选取用哪一种。代码如下：

```
//Activation function
cv::Mat Net::activationFunction(cv::Mat &x, std::string func_type)
{
```

```
activation_function = func_type;
cv::Mat fx;
if (func_type == "sigmoid")
{
    fx = sigmoid(x);
}
if (func_type == "tanh")
{
    fx = tanh(x);
}
if (func_type == "ReLU")
{
    fx = ReLU(x);
}
return fx;
}
```

各个函数更为细节的部分在 `Function.h` 和 `Function.cpp` 文件中。在此略去不表，感兴趣的请君移步 [Github](#)。

需要再次提醒的是，上一篇博客中给出的Net类是精简过的，下面可能会出现一些上一篇Net类里没有出现过的成员变量。完整的Net类的定义还是在 [Github](#) 里。

反向传播过程

反向传播

反向传播原理是链式求导法则，其实就是我们高数中学的复合函数求导法则。这只是在推导公式的时候用的到。具体的推导过程我推荐看看下面这一篇教程，用图示的方法，把前向传播和反向传播表现的清晰明了，强烈推荐！

Principles of training multi-layer neural network using backpropagation.

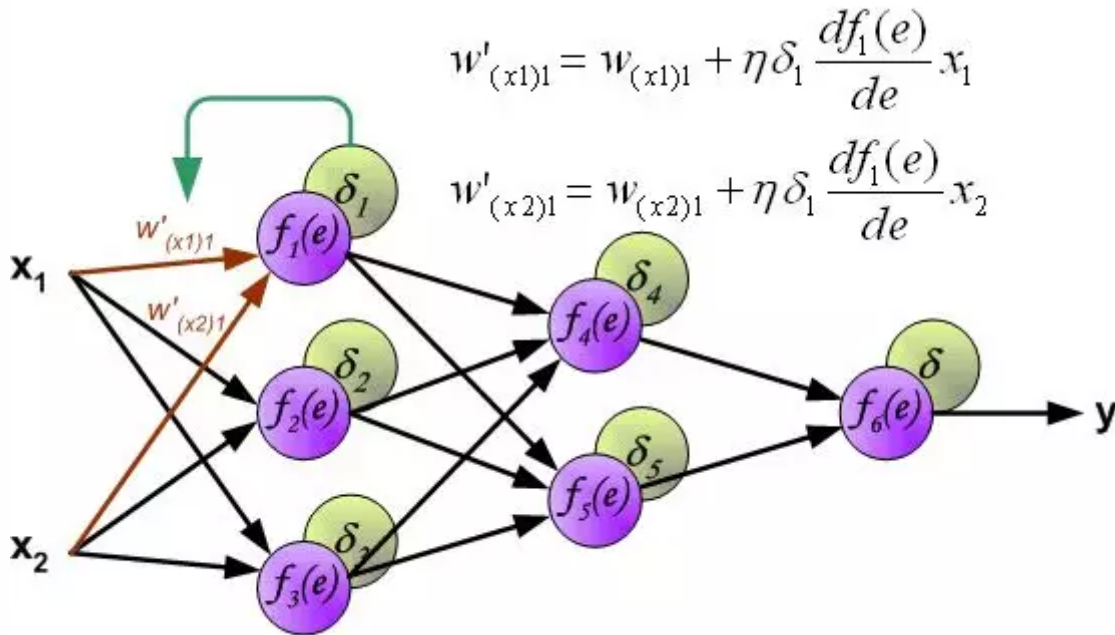
一会将从这一篇文章中截取一张图来说明权值更新的代码。在此之前，还是先看一下反向传播函数 `backward()` 的代码是什么样的：

```
//Forward
void Net::backward()
{
    calcLoss(layer[layer.size() - 1], target, output_error, loss);
    deltaError();
    updateWeights();
}
```


可以看到主要是三行代码，也就是调用了三个函数：

- 第一个函数`calcLoss()`计算输出误差和目标函数，所有输出误差平方和的均值作为需要最小化的目标函数。
- 第二个函数`deltaError()`计算delta误差，也就是下图中 $\delta_1 * df()$ 那部分。
- 第三个函数`updateWeights()`更新权值，也就是用下图中的公式更新权值。

下面是从前面强烈推荐的文章中截的一张图：



就看下`updateWeights()`函数的代码：

```
//Update weights
void Net::updateWeights()
{
    for (int i = 0; i < weights.size(); ++i)
    {
        cv::Mat delta_weights = learning_rate * (delta_err[i] * layer[i].t());
        weights[i] = weights[i] + delta_weights;
    }
}
```

核心的两行代码应该还是能比较清晰反映上图中的那个权值更新的公式的。图中公式里的eta常被称作学习率。训练神经网络调参的时候经常要调节这货。

计算输出误差和delta误差的部分纯粹是数学运算，乏善可陈。但是把代码贴在下面吧。

`calcLoss()` 函数在 `Function.cpp` 文件中：

```
//Objective function
void calcLoss(cv::Mat &output, cv::Mat &target, cv::Mat &output_error, float &loss)
```

```

{
    if (target.empty())
    {
        std::cout << "Can't find the target cv::Matrix" << std::endl;
        return;
    }
    output_error = target - output;
    cv::Mat err_sqrare;
    pow(output_error, 2., err_sqrare);
    cv::Scalar err_sqr_sum = sum(err_sqrare);
    loss = err_sqr_sum[0] / (float)(output.rows);
}

```

`deltaError()` 在 `Net.cpp` 中:

```

//Compute delta error
void Net::deltaError()
{
    delta_err.resize(layer.size() - 1);
    for (int i = delta_err.size() - 1; i >= 0; i--)
    {
        delta_err[i].create(layer[i + 1].size(), layer[i + 1].type());
        //cv::Mat dx = layer[i+1].mul(1 - layer[i+1]);
        cv::Mat dx = derivativeFunction(layer[i + 1], activation_function);
        //Output layer delta error
        if (i == delta_err.size() - 1)
        {
            delta_err[i] = dx.mul(output_error);
        }
        else //Hidden layer delta error
        {
            cv::Mat weight = weights[i];
            cv::Mat weight_t = weights[i].t();
            cv::Mat delta_err_1 = delta_err[i + 1];
            delta_err[i] = dx.mul((weights[i + 1]).t() * delta_err[i + 1]);
        }
    }
}

```

注意

需要注意的就是计算的时候输出层和隐藏层的计算公式是不一样的。

另一个需要注意的就是.....难道大家没觉得本系列文章的代码看起来非常友好吗

至此,神经网络最核心的部分已经实现完毕。剩下的就是想想该如何训练了。这个时候你如果愿意的话仍然可以写一个小程序进行几次前向传播和反向传播。还是那句话, **鬼知道我在能进行传播之前到底花了多长时间调试!**

源码链接

神经网络源码：https://github.com/LiuXiaolong19920720/simple_net

三、神经网络的训练和测试

前言

在之前的文章中我们已经实现了Net类的设计和前向传播和反向传播的过程。可以说神经网络的核心部分已经完成。接下来就是应用层面了。

要想利用神经网络解决实际的问题，比如说进行手写数字的识别，需要用神经网络对样本进行迭代训练，训练完成之后，训练得到的模型是好是坏，我们需要对之进行测试。这正是我们现在需要实现的部分的内容。

完善后的Net类

需要知道的是现在的Net类已经相对完善了，为了实现接下来的功能，不论是成员变量还是成员函数都变得更加的丰富。现在的Net类看起来是下面的样子：

```
class Net
{
public:
    //Integer vector specifying the number of neurons in each layer including the in
    std::vector<int> layer_neuron_num;
    std::string activation_function = "sigmoid";
    double learning_rate;
    double accuracy = 0.;
    std::vector<double> loss_vec;
    float fine_tune_factor = 1.01;
protected:
    std::vector<cv::Mat> layer;
    std::vector<cv::Mat> weights;
    std::vector<cv::Mat> bias;
    std::vector<cv::Mat> delta_err;

    cv::Mat output_error;
    cv::Mat target;
    float loss;

public:
    Net() {};
    ~Net() {};
```

```

//Initialize net:generate weights matrices、layer matrices and bias matrices
// bias default all zero
void initNet(std::vector<int> layer_neuron_num_);

//Initialise the weights matrices.
void initWeights(int type = 0, double a = 0., double b = 0.1);

//Initialise the bias matrices.
void initBias(cv::Scalar& bias);

//Forward
void forward();

//Forward
void backward();

//Train,use loss_threshold
void train(cv::Mat input, cv::Mat target_, float loss_threshold, bool draw_loss_);
void test(cv::Mat &input, cv::Mat &target_);

//Predict,just one sample
int predict_one(cv::Mat &input);

//Predict,more than one samples
std::vector<int> predict(cv::Mat &input);

//Save model;
void save(std::string filename);

//Load model;
void load(std::string filename);

protected:
//initialise the weight matrix.if type =0,Gaussian.else uniform.
void initWeight(cv::Mat &dst, int type, double a, double b);

//Activation function
cv::Mat activationFunction(cv::Mat &x, std::string func_type);

//Compute delta error
void deltaError();

//Update weights
void updateWeights();
};

```

可以看到已经有了训练的函数train()、测试的函数test()，还有实际应用训练好的模型的predict()函数，以及保存和加载模型的函数save()和load()。大部分成员变量和成员函数应该还是能够通过名字就能够知道其功能的。

训练

训练函数train()

本文重点说的是训练函数train()和测试函数test()。这两个函数接受输入 (input) 和标签 (或称为目标值target) 作为输入参数。其中训练函数还要接受一个阈值作为迭代终止条件, 最后一个函数可以暂时忽略不计, 那是选择要不要把loss值实时画出来的标识。

训练的过程如下:

1. 接受一个样本 (即一个单列矩阵) 作为输入, 也即神经网络的第一层;
2. 进行前向传播, 也即forward()函数做的事情。然后计算loss;
3. 如果loss值小于设定的阈值loss_threshold, 则进行反向传播更新阈值;
4. 重复以上过程直到loss小于等于设定的阈值。

train函数的实现如下:

```
//Train,use loss_threshold
void Net::train(cv::Mat input, cv::Mat target_, float loss_threshold, bool draw_loss)
{
    if (input.empty())
    {
        std::cout << "Input is empty!" << std::endl;
        return;
    }

    std::cout << "Train,begain!" << std::endl;

    cv::Mat sample;
    if (input.rows == (layer[0].rows) && input.cols == 1)
    {
        target = target_;
        sample = input;
        layer[0] = sample;
        forward();
        //backward();
        int num_of_train = 0;
        while (loss > loss_threshold)
        {
            backward();
            forward();
            num_of_train++;
            if (num_of_train % 500 == 0)
            {
                std::cout << "Train " << num_of_train << " times" << std::endl;
                std::cout << "Loss: " << loss << std::endl;
            }
        }
        std::cout << std::endl << "Train " << num_of_train << " times" << std::endl;
        std::cout << "Loss: " << loss << std::endl;
        std::cout << "Train sucessfully!" << std::endl;
    }
}
```

```

else if (input.rows == (layer[0].rows) && input.cols > 1)
{
    double batch_loss = loss_threshold + 0.01;
    int epoch = 0;
    while (batch_loss > loss_threshold)
    {
        batch_loss = 0.;
        for (int i = 0; i < input.cols; ++i)
        {
            target = target_.col(i);
            sample = input.col(i);
            layer[0] = sample;

            forward();
            backward();

            batch_loss += loss;
        }

        loss_vec.push_back(batch_loss);

        if (loss_vec.size() >= 2 && draw_loss_curve)
        {
            draw_curve(board, loss_vec);
        }
        epoch++;
        if (epoch % output_interval == 0)
        {
            std::cout << "Number of epoch: " << epoch << std::endl;
            std::cout << "Loss sum: " << batch_loss << std::endl;
        }
        if (epoch % 100 == 0)
        {
            learning_rate *= fine_tune_factor;
        }
    }
    std::cout << std::endl << "Number of epoch: " << epoch << std::endl;
    std::cout << "Loss sum: " << batch_loss << std::endl;
    std::cout << "Train sucessfully!" << std::endl;
}
else
{
    std::cout << "Rows of input don't cv::Match the number of input!" << std::en
}
}

```

这里考虑到了用单个样本和多个样本迭代训练两种情况。而且还有另一种不用loss阈值作为迭代终止条件，而是用正确率的train()函数，内容大致相同，此处略去不表。

在经过train()函数的训练之后，就可以得到一个模型了。所谓模型，可以简单的认为就是权值矩阵。简单的说，可以把神经网络当成一个超级函数组合，我们姑且认为这个超级函数就是 $y = f(x) = ax + b$ 。那么权值就是a和b。反向传播的过程是把a和b当成自变量来处理的，不断调整以得到

最优值或逼近最优值。在完成反向传播之后，训练得到了参数 a 和 b 的最优值，是一个固定值了。这时自变量又变回了 x 。我们希望 a 、 b 最优值作为已知参数的情况下，对于我们的输入样本 x ，通过神经网络计算得到的结果 y ，与实际结果相符合是大概率事件。

测试

测试函数test()

test()函数的作用就是用一组训练时没用到的样本，对训练得到的模型进行测试，把通过这个模型得到的结果与实际想要的结果进行比较，看正确来说到底是多少，我们希望正确率越多越好。

test()的步骤大致如下几步：

1. 用一组样本逐个输入神经网络；
2. 通过前向传播得到一个输出值；
3. 比较实际输出与理想输出，计算正确率。

test()函数的实现如下：

```
//Test
void Net::test(cv::Mat &input, cv::Mat &target_)
{
    if (input.empty())
    {
        std::cout << "Input is empty!" << std::endl;
        return;
    }
    std::cout << std::endl << "Predict,begain!" << std::endl;

    if (input.rows == (layer[0].rows) && input.cols == 1)
    {
        int predict_number = predict_one(input);

        cv::Point target_maxLoc;
        minMaxLoc(target_, NULL, NULL, NULL, &target_maxLoc, cv::noArray());
        int target_number = target_maxLoc.y;

        std::cout << "Predict: " << predict_number << std::endl;
        std::cout << "Target: " << target_number << std::endl;
        std::cout << "Loss: " << loss << std::endl;
    }
    else if (input.rows == (layer[0].rows) && input.cols > 1)
    {
        double loss_sum = 0;
        int right_num = 0;
        cv::Mat sample;
        for (int i = 0; i < input.cols; ++i)
        {
```

```

        sample = input.col(i);
        int predict_number = predict_one(sample);
        loss_sum += loss;

        target = target_.col(i);
        cv::Point target_maxLoc;
        minMaxLoc(target, NULL, NULL, NULL, &target_maxLoc, cv::noArray());
        int target_number = target_maxLoc.y;

        std::cout << "Test sample: " << i << " " << "Predict: " << predict_num
        std::cout << "Test sample: " << i << " " << "Target: " << target_num
        if (predict_number == target_number)
        {
            right_num++;
        }
    }
    accuracy = (double)right_num / input.cols;
    std::cout << "Loss sum: " << loss_sum << std::endl;
    std::cout << "accuracy: " << accuracy << std::endl;
}
else
{
    std::cout << "Rows of input don't cv::Match the number of input!" << std::en
    return;
}
}
}

```

这里在进行前向传播的时候不是直接调用forward()函数，而是调用了predict_one()函数，predict函数的作用是给定一个输入，给出想要的输出值。其中包含了对forward()函数的调用。还有就是对于神经网络的输出进行解析，转换成看起来比较方便的数值。

这一篇的内容已经够多了，我决定把对于predict部分的解释放到下一篇。

源码链接

神经网络源码: https://github.com/LiuXiaolong19920720/simple_net

四、神经网络的预测和输出输出解析

神经网络的预测

预测函数predict()

在上一篇的结尾提到了神经网络的预测函数`predict()`，说道`predict`调用了`forward`函数并进行了输出的解析，输出我们看起来比较方便的值。

`predict()` 函数和 `predict_one()` 函数的区别相信很容易从名字看出来，那就是输入一个样本得到一个输出和输出一组样本得到一组输出的区别，显然 `predict()` 应该是循环调用 `predict_one()` 实现的。所以我们先看一下 `predict_one()` 的代码：

```
int Net::predict_one(cv::Mat &input)
{
    if (input.empty())
    {
        std::cout << "Input is empty!" << std::endl;
        return -1;
    }

    if (input.rows == (layer[0].rows) && input.cols == 1)
    {
        layer[0] = input;
        forward();

        cv::Mat layer_out = layer[layer.size() - 1];
        cv::Point predict_maxLoc;

        minMaxLoc(layer_out, NULL, NULL, NULL, &predict_maxLoc, cv::noArray());
        return predict_maxLoc.y;
    }
    else
    {
        std::cout << "Please give one sample alone and ensure input.rows = layer[0].rows" << std::endl;
        return -1;
    }
}
```

可以在第二个if语句里面看到最主要的内容就是两行：分别是前面提到的前向传播和输出解析。

```
forward();
...
...
minMaxLoc(layer_out, NULL, NULL, NULL, &predict_maxLoc, cv::noArray());
```

前向传播得到最后一层输出层`layer_out`，然后从`layer_out`中提取最大值的位置，最后输出位置的y坐标。

输出的组织方式和解析

输出方式的组织和解析

之所以这么做，就不得不提一下标签或者叫目标值在这里是以何种形式存在的。以激活函数是sigmoid函数为例，sigmoid函数是把实数映射到 $[0,1]$ 区间，所以显然最后的输出 y ： $0 \leq y \leq 1$ 。如果激活函数是tanh函数，则输出区间是 $[-1,1]$ 。如果是sigmoid，而且我们要进行手写字体识别的话，需要识别的数字一共有十个：0-9。显然我们的神经网络没有办法输出大于1的值，所以也就不能直观的用0-9几个数字来作为神经网络的实际目标值或者称之为标签。

这里采用的方案是，把输出层设置为一个单列十行的矩阵，标签是几就把第几行的元素设置为1，其余都设为0。由于编程中一般都是从0开始作为第一位的，所以位置与0-9的数字正好一一对应。我们到时候只需要找到输出最大值所在的位置，也就知道了输出是几。

当然上面说的是激活函数是sigmoid的情况。如果是tanh函数呢？那还是是几就把第几位设为1，而其他位置全部设为-1即可。

如果是ReLU函数呢？ReLU函数的至于是0到正无穷。所以我们可以标签是几就把第几位设为几，其他为全设为0。最后都是找到最大值的位置即可。

这些都是需要根据激活函数来定。代码中是调用opencv的 `minMaxLoc()` 函数来寻找矩阵中最大值的位置。

输入的组织方式和读取方法

输入的组织方式和读取方法

既然说到了输出的组织方式，那就顺便也提一下输入的组织方式。生成神经网络的时候，每一层都是用一个单列矩阵来表示的。显然第一层输入层就是一个单列矩阵。所以在对数据进行预处理的过程中，这里就是把输入样本和标签一列一列地排列起来，作为矩阵存储。标签矩阵的第一列即是第一列样本的标签。以此类推。

值得一提的是，输入的数值全部归一化到0-1之间。

由于这里的数值都是以 `float` 类型保存的，这种数值的矩阵Mat不能直接保存为图片格式，所以这里我选择了把预处理之后的样本矩阵和标签矩阵保存到xml文档中。在源码中可以找到把原始的csv文件转换成xml文件的代码。在 `csv2xml.cpp` 中。而我转换完成的MNIST的部分数据保存在data文件夹中，可以在Github上找到。

在opencv中xml的读写非常方便，如下代码是写入数据：

```
string filename = "input_label.xml";
FileStorage fs(filename, FileStorage::WRITE);
fs << "input" << input_normalized;
fs << "target" << target_; // Write cv::Mat
fs.release();
```

而读取代码的一样简单明了:

```
cv::FileStorage fs;
fs.open(filename, cv::FileStorage::READ);
cv::Mat input_, target_;
fs["input"] >> input_;
fs["target"] >> target_;
fs.release();
```

读取样本和标签

我写了一个函数 `get_input_label()` 从xml文件中从指定的列开始提取一定数目的样本和标签。默认从第0列开始读取, 只是上面函数的简单封装:

```
//Get sample_number samples in XML file, from the start column.
void get_input_label(std::string filename, cv::Mat& input, cv::Mat& label, int sample_num)
{
    cv::FileStorage fs;
    fs.open(filename, cv::FileStorage::READ);
    cv::Mat input_, target_;
    fs["input"] >> input_;
    fs["target"] >> target_;
    fs.release();
    input = input_(cv::Rect(start, 0, sample_num, input_.rows));
    label = target_(cv::Rect(start, 0, sample_num, target_.rows));
}
```

至此其实已经可以开始实践, 训练神经网络识别手写数字了。只有一部分还没有提到, 那就是模型的保存和加载。下一篇将会讲模型的save和load, 然后就可以实际开始进行例子的训练了。等不及的小伙伴可以直接去github下载完整的程序开始跑了。

源码链接

神经网络源码: https://github.com/LiuXiaolong19920720/simple_net

五、模型的保存和加载及实时画出输出曲线

模型的保存和加载

模型的保存与加载

在我们完成对神经网络的训练之后，一般要把模型保存起来。不然每次使用模型之前都需要先训练模型，对于data hungry的神经网络来说，视数据多寡和精度要求高低，训练一次的时间从几分钟到数百个小时不等，这是任何人都耗不起的。把训练好的模型保存下来，当需要使用它的时候，只需要加载就行了。

现在需要考虑的一个问题是，保存模型的时候，我们到底要保存哪些东西？

之前有提到，可以简单的认为权值矩阵就是所谓模型。所以权值矩阵一定要保存。除此之外呢？不能忘记的一点是，我们保存模型是为了加载后能使用模型。显然要求加载模型之后，输入一个或一组样本就能开始前向运算和反向传播。这也就是说，之前实现的时候，forward()之前需要的，这里也都需要，只是权值不是随意初始化了，而是用训练好的权值矩阵代替。基于以上考虑，最终决定要保存的内容如下4个：

1. layer_neuron_num，各层神经元数目，这是生成神经网络需要的唯一参数。
2. weights，神经网络初始化之后需要用训练好的权值矩阵去初始化权值。
3. activation_function，使用神经网络的过程其实就是前向计算的过程，显然需要知道激活函数是什么。
4. learning_rate，如果要在现有模型的基础上继续训练以得到更好的模型，更新权值的时候需要用到这个函数。

再决定了需要保存的内容之后，接下来就是实现了，仍然是保存为 xml 格式，上一篇已经提到了保存和加载 xml 是多么的方便：

```
//Save model;
void Net::save(std::string filename)
{
    cv::FileStorage model(filename, cv::FileStorage::WRITE);
    model << "layer_neuron_num" << layer_neuron_num;
    model << "learning_rate" << learning_rate;
    model << "activation_function" << activation_function;

    for (int i = 0; i < weights.size(); i++)
    {
        std::string weight_name = "weight_" + std::to_string(i);
        model << weight_name << weights[i];
    }
}
```

```
        model.release();
    }

    //Load model;
    void Net::load(std::string filename)
    {
        cv::FileStorage fs;
        fs.open(filename, cv::FileStorage::READ);
        cv::Mat input_, target_;

        fs["layer_neuron_num"] >> layer_neuron_num;
        initNet(layer_neuron_num);

        for (int i = 0; i < weights.size(); i++)
        {
            std::string weight_name = "weight_" + std::to_string(i);
            fs[weight_name] >> weights[i];
        }

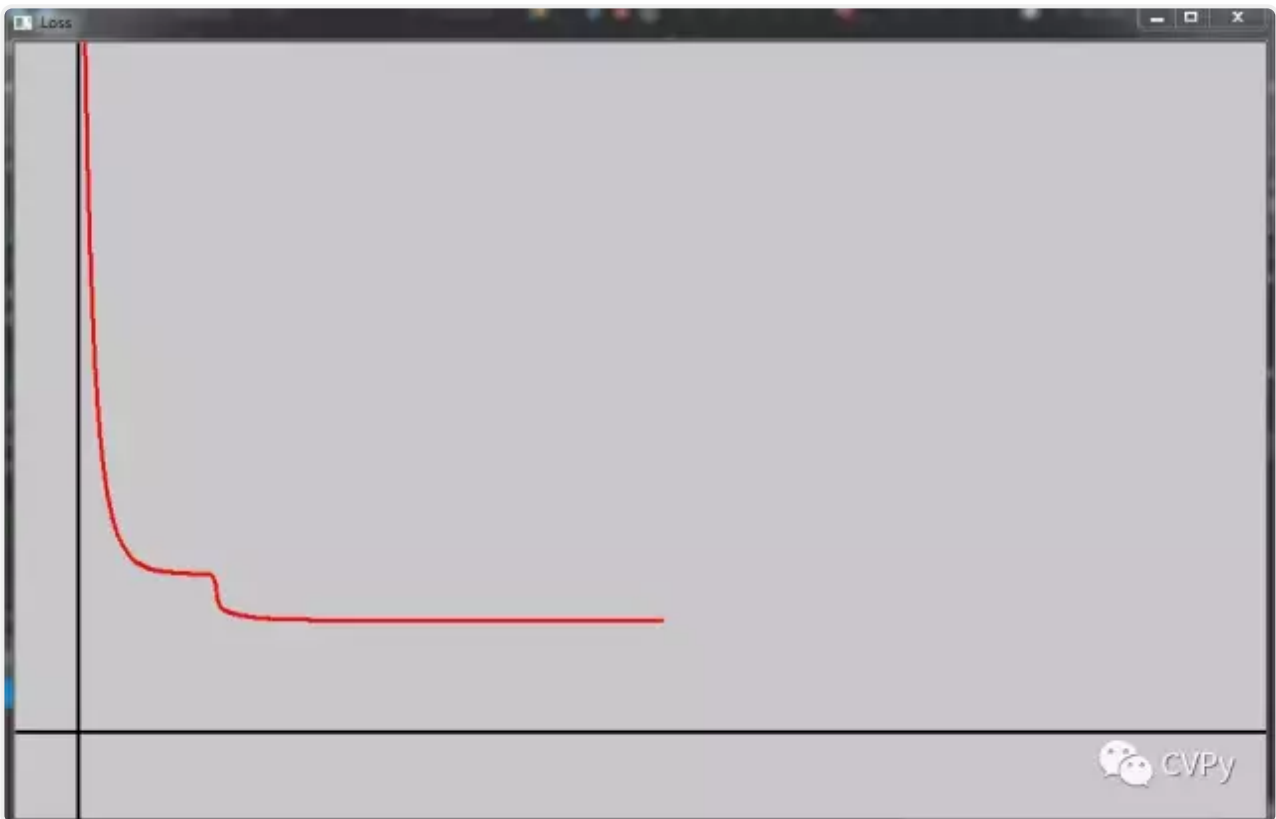
        fs["learning_rate"] >> learning_rate;
        fs["activation_function"] >> activation_function;

        fs.release();
    }
```

实时画出输出曲线

实时画曲线

有时候我们为了有一个直观的观察，我们希望能够是实时的用一个曲线来表示输出误差。但是没有找到满意的程序可用，于是自己就写了一个非常简单的函数，用来实时输出训练时的loss。理想的输出大概像下面这样：



为什么说是理想的输出呢，因为一般来说误差很小，可能曲线直接就是从左下角开始的，上面一大片都没有用到。不过已经能够看出loss的大致走向了。

这个函数的实现其实就是先画两个作为坐标用的直线，然后把相邻点用直线连接起来：

```
//Draw loss curve
void draw_curve(cv::Mat& board, std::vector<double> points)
{
    cv::Mat board_(620, 1000, CV_8UC3, cv::Scalar::all(200));
    board = board_;
    cv::line(board, cv::Point(0, 550), cv::Point(1000, 550), cv::Scalar(0, 0, 0), 2);
    cv::line(board, cv::Point(50, 0), cv::Point(50, 1000), cv::Scalar(0, 0, 0), 2);

    for (size_t i = 0; i < points.size() - 1; i++)
    {
        cv::Point pt1(50 + i * 2, (int)(548 - points[i]));
        cv::Point pt2(50 + i * 2 + 1, (int)(548 - points[i + 1]));
        cv::line(board, pt1, pt2, cv::Scalar(0, 0, 255), 2);
        if (i >= 1000)
        {
            return;
        }
    }
    cv::imshow("Loss", board);
    cv::waitKey(10);
}
```

至此，神经网络已经实现完成了。完整的代码可以在Github上找到。

下一步，就是要用编写的神经网络，用实际样本开始训练了。下一篇，用MNIST数据训练神经网络。

源码链接

神经网络源码：https://github.com/LiuXiaolong19920720/simple_net

六、实战手写数字识别

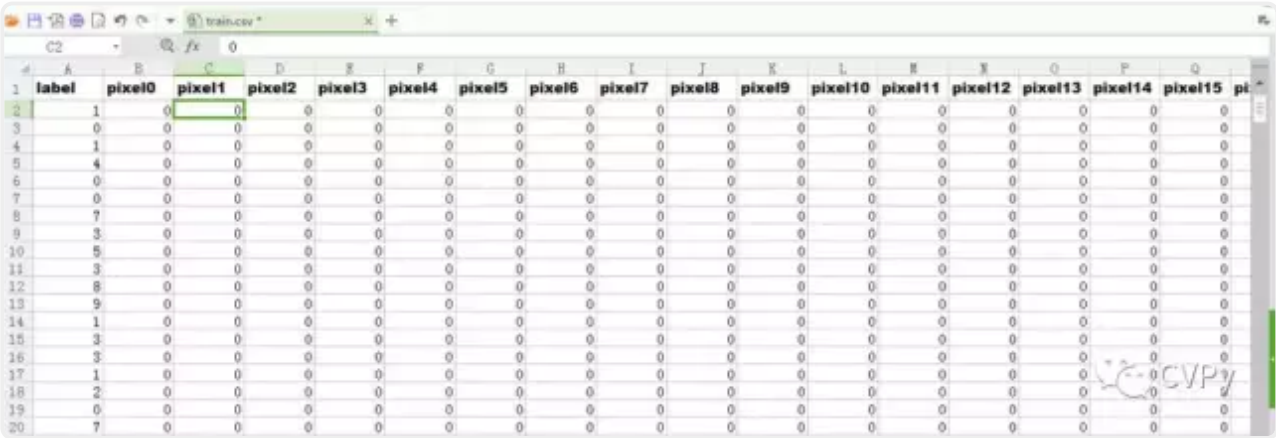
之前的五篇博客讲述的内容应该覆盖了如何编写神经网络的大部分内容，在经过之前的一系列努力之后，终于可以开始实战了。试试写出来的神经网络怎么样吧。

数据准备

MNIST数据集

有人说MNIST手写数字识别是机器学习领域的Hello World，所以我这一次也是从手写字体识别开始。我是从Kaggle找的手写数字识别的数据集。数据已经被保存为csv格式，相对比较方便读取。

数据集包含了数字0-9是个数字的灰度图。但是这个灰度图是展开过的。展开之前都是28x28的图像，展开后成为1x784的一行。csv文件中，每一行有785个元素，第一个元素是数字标签，后面的784个元素分别排列着展开后的184个像素。看起来像下面这样：



	label	pixel0	pixel1	pixel2	pixel3	pixel4	pixel5	pixel6	pixel7	pixel8	pixel9	pixel10	pixel11	pixel12	pixel13	pixel14	pixel15	...
1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	...
2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	...
3	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	...
4	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	...
5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	...
6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	...
7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	...
8	7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	...
9	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	...
10	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	...
11	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	...
12	8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	...
13	9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	...
14	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	...
15	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	...
16	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	...
17	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	...
18	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	...
19	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	...
20	7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	...

也许你已经看到了第一列0-9的标签，但是会疑惑为啥像素值全是0，那是因为这里能显示出来的，甚至不足28x28图像的一行。而数字一般应该在图像中心位置，所以边缘位置当然是啥也没有，往后滑动就能看到非零像素值了。像下面这样：

	DR	DS	DT	DU	DV	DW	DX	DY	DZ	EA	EB	EC	ED	EE	EF	EG	EH
1	pixel120	pixel121	pixel122	pixel123	pixel124	pixel125	pixel126	pixel127	pixel128	pixel129	pixel130	pixel131	pixel132	pixel133	pixel134	pixel135	pixel136
2	0	0	0	0	0	0	0	0	0	0	0	0	188	255	94	0	0
3	0	0	18	30	137	137	192	86	72	1	0	0	0	0	0	0	0
4	0	0	0	0	3	141	139	3	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	0	1	25	130	155	254	254	254	157	30	2	0	0	0	0	0	0
7	0	0	0	0	3	141	202	254	193	44	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
11	60	136	136	147	254	255	199	111	18	9	0	0	0	0	0	0	0
12	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
13	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
16	0	0	101	222	253	253	192	113	88	0	0	0	0	0	0	0	0
17	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
18	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
19	0	0	0	0	0	0	0	0	0	144	254	130	0	0	0	0	0
20	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

这里需要注意的是，像素值的范围是0-255。一般在数据预处理阶段都会归一化，全部除以255，把值转换到0-1之间。

csv文件中包含42000个样本，这么多样本，对于我七年前买的4000元级别的破笔记本来说，单单是读取一次都得半天，更不要提拿这么多样本去迭代训练了，简直是噩梦（兼论一个苦逼的学生几年能挣到换电脑的钱！）。所以我只是提取了前1000个样本，然后把归一化后的样本和标签都保存到一个xml文件中。在前面的一篇博客中已经提到了输入输出的组织形式，偷懒直接复制了。

既然说到了输出的组织方式，那就顺便也提一句输入的组织方式。生成神经网络的时候，每一层都是用一个单列矩阵来表示的。显然第一层输入层就是一个单列矩阵。所以在对数据进行预处理的过程中，我就是把输入样本和标签一列一列地排列起来，作为矩阵存储。标签矩阵的第一列即是第一列样本的标签。以此类推。

把输出层设置为一个单列十行的矩阵，标签是几就把第几行的元素设置为1，其余都设为0。由于编程中一般都是从0开始作为第一位的，所以位置与0-9的数字正好一一对应。我们到时候只需要找到输出最大值所在的位置，也就知道了输出是几。”

这里只是重复一下，这一部分的代码在 `csv2xml.cpp` 中：

```
#include<opencv2\opencv.hpp>
#include<iostream>
using namespace std;
using namespace cv;

//int csv2xml()
int main()
{
    CvMLData mldata;
    mldata.read_csv("train.csv");//读取csv文件
    Mat data = cv::Mat(mldata.get_values(), true);
    cout << "Data have been read successfully!" << endl;
    //Mat double_data;
    //data.convertTo(double_data, CV_64F);

    Mat input_ = data(Rect(1, 1, 784, data.rows - 1)).t();
    Mat label_ = data(Rect(0, 1, 1, data.rows - 1));
    Mat target_(10, input_.cols, CV_32F, Scalar::all(0.));
```



```

Mat digit(28, 28, CV_32FC1);
Mat col_0 = input_.col(3);
float label0 = label_.at<float>(3, 0);
cout << label0;
for (int i = 0; i < 28; i++)
{
    for (int j = 0; j < 28; j++)
    {
        digit.at<float>(i, j) = col_0.at<float>(i * 28 + j);
    }
}

for (int i = 0; i < label_.rows; ++i)
{
    float label_num = label_.at<float>(i, 0);
    //target_.at<float>(label_num, i) = 1.;
    target_.at<float>(label_num, i) = label_num;
}

Mat input_normalized(input_.size(), input_.type());
for (int i = 0; i < input_.rows; ++i)
{
    for (int j = 0; j < input_.cols; ++j)
    {
        //if (input_.at<double>(i, j) >= 1.)
        //{
            input_normalized.at<float>(i, j) = input_.at<float>(i, j) / 255.;
        //}
    }
}

string filename = "input_label_0-9.xml";
FileStorage fs(filename, FileStorage::WRITE);
fs << "input" << input_normalized;
fs << "target" << target_; // Write cv::Mat
fs.release();

Mat input_1000 = input_normalized(Rect(0, 0, 10000, input_normalized.rows));
Mat target_1000 = target_(Rect(0, 0, 10000, target_.rows));

string filename2 = "input_label_0-9_10000.xml";
FileStorage fs2(filename2, FileStorage::WRITE);

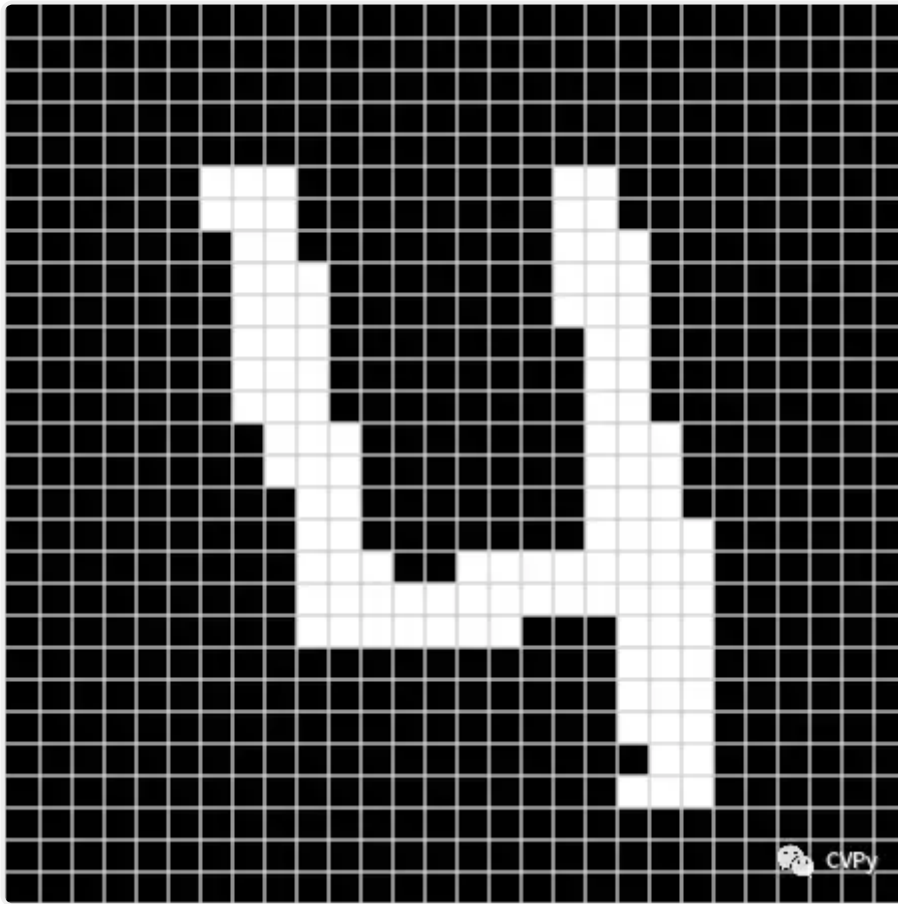
fs2 << "input" << input_1000;
fs2 << "target" << target_1000; // Write cv::Mat
fs2.release();

return 0;
}

```

这是我最近用ReLU的时候的代码，标签是几就把第几位设为几，其他为全设为0。最后都是找到最大值的位置即可。

在代码中 `Mat digit` 的作用是，检验下转换后的矩阵和标签是否对应正确这里是把`col(3)`，也就是第四个样本从一行重新变成28x28的图像，看上面的第一张图的第一列可以看到，第四个样本的标签是4。那么它转换回来的图像时什么样呢？是下面这样：



这里也证明了为啥第一张图看起来像素全是0。边缘全黑能不是0吗？

然后在使用的时候用前面提到过的`get_input_label()`获取一定数目的样本和标签。

实战数字识别

实战

没想到前面数据处理说了那么多。。。。

废话少说，直接说训练的过程：

1. 给定每层的神经元数目，初始化神经网络和权值矩阵
2. 从`input/label1000.xml`文件中取前800个样本作为训练样本，后200作为测试样本。
3. 这是神经网络的一些参数：训练时候的终止条件，学习率，激活函数类型
4. 前800样本训练神经网络，直到满足`loss`小于阈值`loss_threshold`，停止。
5. 后200样本测试神经网络，输出正确率。
6. 保存训练得到的模型。

以sigmoid为激活函数的训练代码如下:

```
#include "../include/Net.h"
//<opencv2\opencv.hpp>

using namespace std;
using namespace cv;
using namespace liu;

int main(int argc, char *argv[])
{
    //Set neuron number of every layer
    vector<int> layer_neuron_num = { 784,100,10 };

    // Initialise Net and weights
    Net net;
    net.initNet(layer_neuron_num);
    net.initWeights(0, 0., 0.01);
    net.initBias(Scalar(0.5));

    //Get test samples and test samples
    Mat input, label, test_input, test_label;
    int sample_number = 800;
    get_input_label("data/input_label_1000.xml", input, label, sample_number);
    get_input_label("data/input_label_1000.xml", test_input, test_label, 200, 800);

    //Set loss threshold, learning rate and activation function
    float loss_threshold = 0.5;
    net.learning_rate = 0.3;
    net.output_interval = 2;
    net.activation_function = "sigmoid";

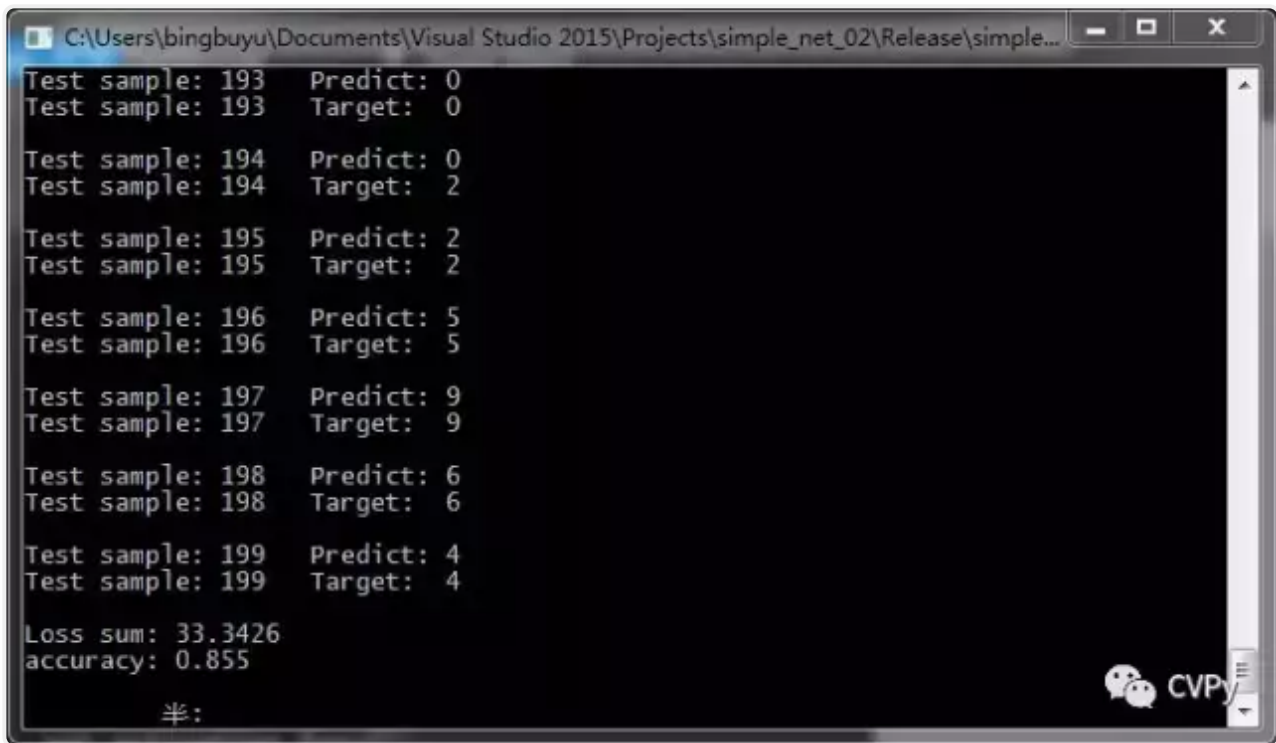
    //Train, and draw the loss curve (cause the last parameter is true) and test the train
    net.train(input, label, loss_threshold, true);
    net.test(test_input, test_label);

    //Save the model
    net.save("models/model_sigmoid_800_200.xml");

    getchar();
    return 0;
}
```

对比前面说的六个过程, 代码应该是很清晰的了。参数output_interval是间隔几次迭代输出一次, 这设置为迭代两次输出一次。

如果按照上面的参数来训练, 正确率是0.855:



```

C:\Users\bingbuyu\Documents\Visual Studio 2015\Projects\simple_net_02\Release\simple...
Test sample: 193 Predict: 0
Test sample: 193 Target: 0

Test sample: 194 Predict: 0
Test sample: 194 Target: 2

Test sample: 195 Predict: 2
Test sample: 195 Target: 2

Test sample: 196 Predict: 5
Test sample: 196 Target: 5

Test sample: 197 Predict: 9
Test sample: 197 Target: 9

Test sample: 198 Predict: 6
Test sample: 198 Target: 6

Test sample: 199 Predict: 4
Test sample: 199 Target: 4

Loss sum: 33.3426
accuracy: 0.855

```

在只有800个样本的情况下，这个正确率我认为还是可以接受的。

如果要直接使用训练好的样本，那就更加简单了：

```

//Get test samples and the label is 0--1
Mat test_input, test_label;
int sample_number = 200;
int start_position = 800;
get_input_label("data/input_label_1000.xml", test_input, test_label, sample_number,

//Load the trained net and test.
Net net;
net.load("models/model_sigmoid_800_200.xml");
net.test(test_input, test_label);

getchar();
return 0;

```

如果激活函数是tanh函数，由于tanh函数的值域是 $[-1, 1]$ ，所以在训练的时候要把标签矩阵稍作改动，需要改动的地方如下：

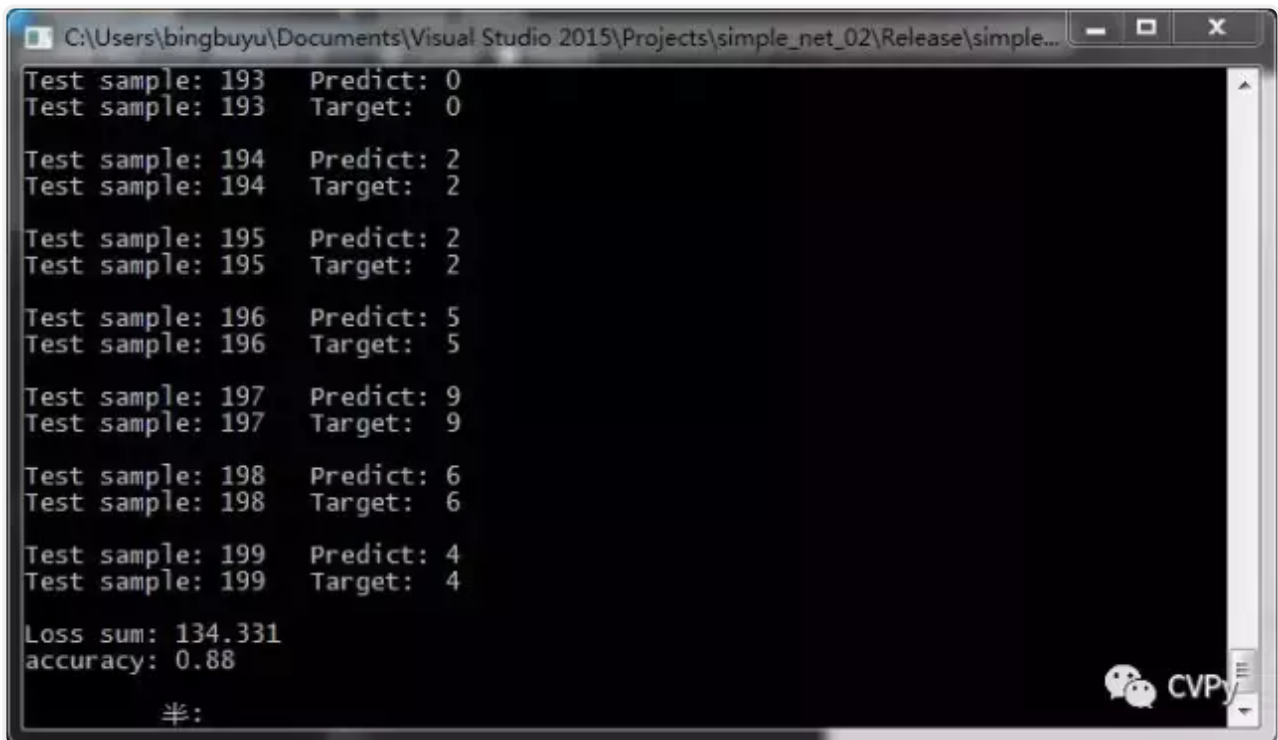
```

//Set loss threshold, learning rate and activation function
float loss_threshold = 0.2;
net.learning_rate = 0.02;
net.output_interval = 2;
net.activation_function = "tanh";

//convert label from 0--1 to -1--1, cause tanh function range is [-1, 1]
label = 2 * label - 1;
test_label = 2 * test_label - 1;

```

这里不光改了标签，还有几个参数也是需要改以下的，学习率比sigmoid的时候要小一个量级，效果会比较好。这样训练出来的正确率大概在0.88左右，也是可以接受的。



```
C:\Users\bingbuyu\Documents\Visual Studio 2015\Projects\simple_net_02\Release\simple...
Test sample: 193 Predict: 0
Test sample: 193 Target: 0

Test sample: 194 Predict: 2
Test sample: 194 Target: 2

Test sample: 195 Predict: 2
Test sample: 195 Target: 2

Test sample: 196 Predict: 5
Test sample: 196 Target: 5

Test sample: 197 Predict: 9
Test sample: 197 Target: 9

Test sample: 198 Predict: 6
Test sample: 198 Target: 6

Test sample: 199 Predict: 4
Test sample: 199 Target: 4

Loss sum: 134.331
accuracy: 0.88

半:
```

源码链接

神经网络源码: https://github.com/LiuXiaolong19920720/simple_net

下载1: OpenCV-Contrib扩展模块中文版教程

在「小白学视觉」公众号后台回复：**扩展模块中文教程**，即可下载全网第一份OpenCV扩展模块教程中文版，涵盖**扩展模块安装、SFM算法、立体视觉、目标跟踪、生物视觉、超分辨率处理**等二十多章内容。

下载2: Python视觉实战项目52讲

在「小白学视觉」公众号后台回复：**Python视觉实战项目**，即可下载包括**图像分割、口罩检测、车道线检测、车辆计数、添加眼线、车牌识别、字符识别、情绪检测、文本内容提取、面部识别**等31个视觉实战项目，助力快速学校计算机视觉。


下载3: OpenCV实战项目20讲

在「小白学视觉」公众号后台回复：**OpenCV实战项目20讲**，即可下载含有**20个基于OpenCV实现20个实战项目**，实现OpenCV学习进阶。

交流群

欢迎加入公众号读者群一起和同行交流，目前有**SLAM**、**三维视觉**、**传感器**、**自动驾驶**、**计算摄影**、**检测**、**分割**、**识别**、**医学影像**、**GAN**、**算法竞赛**等微信群（以后会逐渐细分），请扫描下面微信号加群，备注：“昵称+学校/公司+研究方向”，例如：“张三 + 上海交大 + 视觉SLAM”。**请按照格式备注，否则不予通过**。添加成功后会根据研究方向邀请进入相关微信群。**请勿在群内发送广告**，否则会请出群，谢谢理解~






小白学视觉

计算机视觉
论文解读 求职感想
SLAM技术 深度学习 学习感受

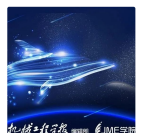
距离我们只差一个
长按关注

聚集地
计算机视觉学者



喜欢此内容的人还喜欢

基于神经网络代理模型的列车侧风稳定性优化 | CJME论文推荐
机械工程学报



在非结构化数据上实现人工神经网络
磐创AI



