CS205 C/C++ Programming - Project 5

Name: 陈一戈

SID: 12011625

Part I Analysis

1-1 The goal of the project

本次 Project 要求使用 C/C++ 来实现一个简单的卷积神经网络的模型（训练数据老师已经提供），这个模型可以用来检测一张图片是否是一张人脸（但仅限于上半身）。最后将我们写好的程序分别在 X86 系统和 ARM 系统上进行测试。

1-2 The analysis of implementation

本次 Project 中整个卷积的过程如下图所示：

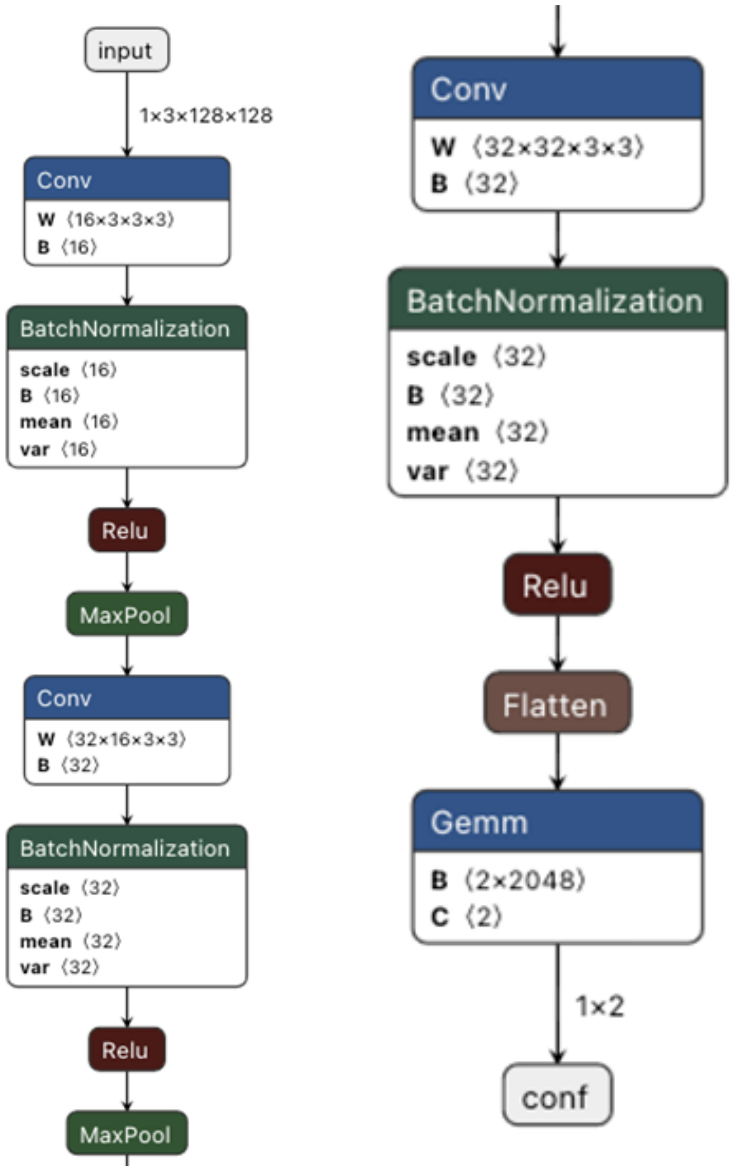

图1-1

首先，通过 OpenCV 库中的 cv::Mat 类读取输入的图片，并将其统一转为 128 * 128 像素的图片，该图片就是一个3通道行列均为128的矩阵（简记为 3 * 128 * 128），每个矩阵元素都为unsigned char类型，范围为[0-255]。我们先通过 cv::Mat 中的 convertTo() 函数将矩阵中的每一个元素都除以255，转为 float 类型，范围为[0-1]。之后进入卷积层(Convolutional layer)，通过和老师所给训练数据中的 conv_params[0] 的 conv0_weight 卷积核进行卷积（BatchNormalization这一步在老师所给训练数据中已经考虑）。在这里，为了提高运算速度，我并没有用卷积的定义来实现这步操作，而是采用了一种名为 im2col 的优化方法，其主要思想就是先不去计算每个卷积核和输入矩阵的每个 3*3 子矩阵做点乘得到的值，而是先将传入的矩阵变形并将卷积核展开，直接计算这两个矩阵相乘（可直接使用成熟的矩阵乘法库比如 OpenBLAS），即可得到正确的结果。为了防止过拟合的问题，还要通过 Relu 激活函数将矩阵中的每个元素都变成非负数。由于此次操作的填充(padding)为1，步长(stride)为2，输出通道数(out_channels)为16，得到的结果矩阵大小为 16 * 64 * 64 。

之后，进入池化层(Pooling)，做步长为2的 MaxPooling 操作，提取每个 2*2 子矩阵中的最大的那个元素值，得到 16 * 32 * 32 的矩阵。接着重复上述卷积层和池化层的操作，得到 32 * 16 * 16 的矩阵。再重复一次卷积层的操作，得到 32 * 8 * 8 的矩阵。最后，进入全连接层(Fully Connected Layer)，由于按照我的方法得到的 32 * 8 * 8 的矩阵中的元素已经是按顺序排列好的，不需要再做 flatten 的操作。此后将所给训练数据中的 fc_params[0] 中大小为 2 * 2048 的 fc0_weight 矩阵与该矩阵(可以看作是2048 * 1的矩阵)进行相乘，最终得到了2个数据，再通过 softmax 得到图片是人脸的概率。整个卷积过程完毕。

Part II Code

本次 Project 一共有5个文件，分别是matrix.hpp、cnn.hpp、cnn.cpp、main.cpp、face_binary_cls.cpp(已提供)。

首先是矩阵模板类 Matrix 的定义和实现，在 matrix.hpp 文件中。由于本次 Project 中矩阵类的作用几乎只是计算矩阵乘法，为了提高效率，本次 Project 我并没有直接沿用上次 Project 写好的矩阵类，而将一些属性和方法做了删除。我认为我的代码通用性较高，也较好地体现了 "simple, beautiful and efficient" 的特点。

matrix.hpp

```cpp
#pragma once

#include <iostream>

template <typename T>
class Matrix
{
private:
    size_t rows{};
    size_t cols{};
    T *data;
    int *refcount{};

public:
    Matrix();

    Matrix(size_t row, size_t col);

    Matrix(const Matrix<T> &m);

    ~Matrix();

    [[nodiscard]] size_t getRows() const;

    [[nodiscard]] size_t getCols() const;

    T *&getData();

```

```
    void setRows(size_t row);

    void setCols(size_t col);

    void release();

    //实现的是weight * Matrix(使用ikj访存优化)并且加上偏移量bias，flag变量来确定此次要不要做relu操作
    void mul(T *weight, T *bias, size_t row, size_t col, Matrix<T> &out, bool flag) const;

    //使用OpenBLAS计算矩阵乘法weight * Matrix并且加上偏移量，flag变量来确定此次要不要做relu操作
    void mul_openblas(T *weight, T *bias, size_t row, size_t col, Matrix<T> &out, bool flag) const;

    Matrix<T> operator+(const Matrix<T> &m) const;

    Matrix<T> operator-(const Matrix<T> &m) const;

    Matrix<T> operator*(const Matrix<T> &m) const;

    T &operator()(size_t row, size_t col) const;

    T *operator[](size_t row) const;

    template <typename E>
    friend std::ostream &operator<<(std::ostream &os, const Matrix<E> &m);

    template <typename E>
    friend std::istream &operator>>(std::istream &is, Matrix<E> &m);
};

template <typename T>
Matrix<T>::Matrix() : rows(0), cols(0), data(nullptr), refcount(nullptr) {}

template <typename T>
Matrix<T>::Matrix(size_t row, size_t col) : rows(row), cols(col)
{
    data = new T[row * col];
    memset(data, 0, sizeof(T) * row * col);
    refcount = new int(0);
}

template <typename T>
Matrix<T>::Matrix(const Matrix<T> &m)
{
    if (m.refcount)
    {
        (*(m.refcount))++;
    }
    this->rows = m.rows;
    this->cols = m.cols;
    this->refcount = m.refcount;
    this->data = m.data;
}

template <typename T>
Matrix<T>::~Matrix()
```

```
{
    release();
}

template <typename T>
size_t Matrix<T>::getRows() const
{
    return rows;
}

template <typename T>
size_t Matrix<T>::getCols() const
{
    return cols;
}

template <typename T>
void Matrix<T>::setRows(size_t row)
{
    this->rows = row;
}

template <typename T>
void Matrix<T>::setCols(size_t col)
{
    this->cols = col;
}

template <typename T>
T *&Matrix<T>::getData()
{
    return data;
}

template <typename T>
void Matrix<T>::release()
{
    if (refcount != nullptr)
    {
        if (*refcount == 0)
        {
            delete refcount;
            delete[] data;
        }
        else
        {
            (*refcount)--;
        }
    }
    this->rows = 0;
    this->cols = 0;
    this->data = nullptr;
    this->refcount = nullptr;
}

template <typename T>
void Matrix<T>::mul(T *weight, T *bias, size_t row, size_t col,
Matrix<T> &out, bool flag) const
{
    assert(col == this->rows);
    if (out.data)
```

```
    {
        delete[] out.data;
    }
    out.rows = row;
    out.cols = this->cols;
    out.data = new T[row * this->cols];
    memset(out.data, 0, sizeof(T) * row * this->cols);
    for (size_t i = 0; i < row; i++)
    {
        for (size_t k = 0; k < col; k++)
        {
            T t = weight[i * col + k];
            for (size_t j = 0; j < this->cols; j++)
            {
                out.data[i * this->cols + j] += t * this->data[k * this->cols + j];
            }
        }
    }
    for (size_t channel = 0; channel < out.rows; channel++)
    {
        for (size_t y = 0; y < out.cols; y++)
        {
            //加上bias偏移量
            out[channel][y] += bias[channel];
            if (flag)
            {
                //去掉负数
                out[channel][y] = std::max(0.0f, out[channel][y]);
            }
        }
    }
}

template <typename T>
void Matrix<T>::mul_openblas(T *weight, T *bias, size_t row, size_t col, Matrix<T> &out, bool flag) const
{
    assert(col == this->rows);
    delete[] out.data;
    out.rows = row;
    out.cols = this->cols;
    out.data = new T[row * this->cols];
    memset(out.data, 0, sizeof(T) * row * this->cols);
    cblas_sgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, row, this->cols, col, 1, weight, col, this->data, this->cols, 0, out.getData(), this->cols);
    for (size_t channel = 0; channel < out.rows; channel++)
    {
        for (size_t y = 0; y < out.cols; y++)
        {
            out[channel][y] += bias[channel];
            if (flag)
            {
                out[channel][y] = std::max(0.0f, out[channel][y]);
            }
        }
    }
}

template <typename T>
```

```
Matrix<T> Matrix<T>::operator+(const Matrix<T> &m) const
{
    assert(this->rows == m.rows && this->cols == m.cols);
    Matrix add(this->rows, this->cols);
    size_t length = this->rows * this->cols;
    const T *p1 = this->data;
    const T *p2 = m.data;
    float *p3 = add.data;
    for (size_t i = 0; i < length; i++)
    {
        *(p3++) = *(p1++) + *(p2++);
    }
    return add;
}

template <typename T>
Matrix<T> Matrix<T>::operator-(const Matrix<T> &m) const
{
    assert(this->rows == m.rows && this->cols == m.cols);
    Matrix sub(this->rows, this->cols);
    size_t length = this->rows * this->cols;
    const T *p1 = this->data;
    const T *p2 = m.data;
    float *p3 = sub.data;
    for (size_t i = 0; i < length; i++)
    {
        *(p3++) = *(p1++) - *(p2++);
    }
    return sub;
}

template <typename T>
Matrix<T> Matrix<T>::operator*(const Matrix<T> &m) const
{
    assert(this->cols == m.rows);
    Matrix<T> mul(this->rows, m.cols);
    cblas_sgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, this->rows,
m.cols, this->cols, 1, this->data, this->cols, m.data, m.cols, 0,
mul.data, m.cols);
    return mul;
}

template <typename T>
T &Matrix<T>::operator()(size_t row, size_t col) const
{
    return data[row * this->cols + col];
}

template <typename T>
T *Matrix<T>::operator[](size_t row) const
{
    return data + row * cols;
}

template <typename T>
std::ostream &operator<<(std::ostream &os, const Matrix<T> &m)
{
    for (size_t i = 0; i < m.rows; i++)
    {
        for (size_t j = 0; j < m.cols; j++)
        {
```

```
            os << m.data[i * m.cols + j] << " ";
        }
        os << std::endl;
    }
    return os;
}

template <typename T>
std::istream &operator>>(std::istream &is, Matrix<T> &m)
{
    is >> m.rows;
    is >> m.cols;
    m.data = new T[m.rows * m.cols];
    for (size_t i = 0; i < m.rows; i++)
    {
        for (size_t j = 0; j < m.cols; j++)
        {
            is >> m.data[i * m.cols + j];
        }
    }
    return is;
}
```

接着我将有关卷积运算函数的定义和实现分别放在 cnn.hpp 以及 cnn.cpp 中:

cnn.hpp

```
#pragma once

#include <opencv2/opencv.hpp>
#include <cblas.h>
#include <algorithm>
#include "face_binary_cls.cpp"
#include "matrix.hpp"

//将输入进来的图片img展开为矩阵乘法的标准形式，输出矩阵为out
void imgToMat(const cv::Mat &img, Matrix<float> &out, size_t kernel_size, size_t paddings, size_t stride);

//将每一次卷积后得到的结果矩阵展开为矩阵乘法的标准形式作为下一次卷积的输入矩阵，输出矩阵为out
void convResToMat(Matrix<float> &in, Matrix<float> &out, size_t kernel_size, size_t paddings, size_t stride);

//对输入进来的矩阵in和卷积核做矩阵乘法，结果为out
void convAndRelu(Matrix<float> &in, conv_param &conv_p, Matrix<float> &out);

//对in做池化操作，结果为out
void maxPooling(Matrix<float> &in, Matrix<float> &out);

//全连接层的矩阵乘法，结果为out
void fullyConnected(Matrix<float> &in, fc_param &fc_p, Matrix<float> &out);
```

cnn.cpp

```
#include "cnn.hpp"

```

```cpp
void imgToMat(cv::Mat &img, Matrix<float> &out, size_t kernel_size, size_t paddings, size_t stride)
{
    assert(img.data != nullptr);
    //先将输出的out矩阵里的数据清空
    delete[] out.getData();
    cv::Mat temp;
    //如果输入图片的尺寸不是128*128，就将它转成128*128的图片
    if (img.rows != 128 || img.cols != 128)
    {
        cv::resize(img, temp, cv::Size(128, 128));
        img = temp;
    }
    std::vector<cv::Mat> BGR(3);
    cv::split(img, BGR);
    //将img的三个通道分离得到B、G、R
    cv::Mat B = BGR[0];
    cv::Mat G = BGR[1];
    cv::Mat R = BGR[2];
    size_t in_size = img.rows;
    //由公式可以计算卷积后得到的矩阵有多少行(列)
    size_t size = (in_size + 2 * paddings - kernel_size) / stride + 1;
    out.setRows(kernel_size * kernel_size * 3);
    out.setCols(size * size);
    out.getData() = new float[out.getRows() * out.getCols()];
    memset(out.getData(), 0, sizeof(float) * out.getRows() * out.getCols());
    //将输入的矩阵展开，以R、G、B的顺序输出到out矩阵中，并根据padding的值来确定是否要补0，以及kernel_size的值决定补几圈0
    //circle来确定补几圈0
    size_t circle = (kernel_size - 1) / 2;
    size_t col = 0;
    for (size_t i = 0, cnt1 = 0; cnt1 < size; i += stride, cnt1++)
    {
        for (size_t j = 0, cnt2 = 0; cnt2 < size; j += stride, col++, cnt2++)
        {
            size_t row = 0;
            for (size_t r = i; r < i + kernel_size; r++)
            {
                for (size_t c = j; c < j + kernel_size; c++, row++)
                {
                    if (paddings > 0)
                    {
                        if ((r >= 0 && r <= circle - 1) || (r >= in_size + circle && r <= in_size + 2 * circle - 1) || (c >= 0 && c <= circle - 1) || (c >= in_size + circle && c <= in_size + 2 * circle - 1))
                        {
                            out[row][col] = 0;
                        }
                        else
                        {
                            out[row][col] = R.at<float>(r - paddings, c - paddings);
                        }
                    }
                    else
                    {
                        out[row][col] = R.at<float>(r, c);
                    }
                }
```

```
            }
            for (size_t r = i; r < i + kernel_size; r++)
            {
                for (size_t c = j; c < j + kernel_size; c++, row++)
                {
                    if (paddings > 0)
                    {
                        if ((r ⩾ 0 && r ⩽ circle - 1) || (r ⩾ in_size + circle && r ⩽ in_size + 2 * circle - 1) || (c ⩾ 0 && c ⩽ circle - 1) || (c ⩾ in_size + circle && c ⩽ in_size + 2 * circle - 1))
                        {
                            out[row][col] = 0;
                        }
                        else
                        {
                            out[row][col] = G.at<float>(r - paddings, c - paddings);
                        }
                    }
                    else
                    {
                        out[row][col] = G.at<float>(r, c);
                    }
                }
            }
            for (size_t r = i; r < i + kernel_size; r++)
            {
                for (size_t c = j; c < j + kernel_size; c++, row++)
                {
                    if (paddings > 0)
                    {
                        if ((r ⩾ 0 && r ⩽ circle - 1) || (r ⩾ in_size + circle && r ⩽ in_size + 2 * circle - 1) || (c ⩾ 0 && c ⩽ circle - 1) || (c ⩾ in_size + circle && c ⩽ in_size + 2 * circle - 1))
                        {
                            out[row][col] = 0;
                        }
                        else
                        {
                            out[row][col] = B.at<float>(r - paddings, c - paddings);
                        }
                    }
                    else
                    {
                        out[row][col] = B.at<float>(r, c);
                    }
                }
            }
        }
    }
}

void convResToMat(Matrix<float> &in, Matrix<float> &out, size_t kernel_size, size_t paddings, size_t stride)
{
    assert(in.getData() ≠ nullptr);
    delete[] out.getData();
    auto in_size = (size_t)sqrt(in.getCols());
    //由公式可以计算卷积后得到的矩阵有多少行(列)
    size_t size = (in_size + 2 * paddings - kernel_size) / stride + 1;
```

```
111     out.setRows(kernel_size * kernel_size * in.getRows());
112     out.setCols(size * size);
113     out.getData() = new float[out.getRows() * out.getCols()];
114     memset(out.getData(), 0, sizeof(float) * out.getRows() *
out.getCols());
115     //将输出的多通道矩阵展开成列向量，并根据padding的值来确定是否要补0，以及
kernel_size的值决定补几圈0
116     //circle来确定补几圈0
117     size_t circle = (kernel_size - 1) / 2;
118     size_t col = 0;
119     for (size_t i = 0, cnt1 = 0; cnt1 < size; i += stride, cnt1++)
120     {
121         for (size_t j = 0, cnt2 = 0; cnt2 < size; j += stride, col++,
cnt2++)
122         {
123             size_t row = 0;
124             for (size_t channel = 0; channel < in.getRows(); channel++)
125             {
126                 for (size_t r = i; r < i + kernel_size; r++)
127                 {
128                     for (size_t c = j; c < j + kernel_size; c++, row++)
129                     {
130                         if (paddings > 0)
131                         {
132                             if ((r ⩾ 0 && r ⩽ circle - 1) || (r ⩾
in_size + circle && r ⩽ in_size + 2 * circle - 1) || (c ⩾ 0 && c ⩽
circle - 1) || (c ⩾ in_size + circle && c ⩽ in_size + 2 * circle - 1))
133                             {
134                                 out[row][col] = 0;
135                             }
136                             else
137                             {
138                                 out[row][col] = in[channel][(r -
paddings) * in_size + c - paddings];
139                             }
140                         }
141                         else
142                         {
143                             out[row][col] = in[channel][r * in_size +
c];
144                         }
145                     }
146                 }
147             }
148         }
149     }
150 }
151
152 void convAndRelu(Matrix<float> &in, conv_param &conv_p, Matrix<float>
&out)
153 {
154     in.mul_openblas(conv_p.p_weight, conv_p.p_bias, conv_p.out_channels,
155                     conv_p.in_channels * conv_p.kernel_size *
conv_p.kernel_size, out, true);
156 }
157
158 void maxPooling(Matrix<float> &in, Matrix<float> &out)
159 {
160     delete[] out.getData();
161     auto size = (size_t)sqrt(in.getCols());
162     out.setRows(in.getRows());
```

```
    out.setCols(in.getCols() / 4);
    out.getData() = new float[out.getRows() * out.getCols()];
    memset(out.getData(), 0, sizeof(float) * out.getRows() *
out.getCols());
    float *mat = out.getData();
    for (size_t channel = 0; channel < in.getRows(); channel++)
    {
        for (size_t row = 0; row < size; row += 2)
        {
            for (size_t col = 0; col < size; col += 2)
            {
                float a = in.getData()[channel * in.getCols() + row *
size + col];
                float b = in.getData()[channel * in.getCols() + row *
size + col + 1];
                float c = in.getData()[channel * in.getCols() + row *
size + col + size];
                float d = in.getData()[channel * in.getCols() + row *
size + col + size + 1];
                //取2*2方块中的最大值
                *mat = std::max(a, std::max(b, std::max(c, d)));
                mat++;
            }
        }
    }
}

void fullyConnected(Matrix<float> &in, fc_param &fc_p, Matrix<float>
&out)
{
    in.mul_openblas(fc_p.p_weight, fc_p.p_bias, fc_p.out_features,
fc_p.in_features, out, false);
}
```

最后是 main.cpp 测试文件，为了便于观看，我并没有把 CNN () 函数中关于计时的代码放进来：

main.cpp

```
#include <iostream>
#include <string>
#include <vector>
#include <cmath>
#include <chrono>
#include "cnn.cpp"

using namespace std;
using namespace cv;

//获取时间戳
time_t getTimeStamp()
{
    chrono::time_point<chrono::system_clock, chrono::milliseconds> tp =
chrono::time_point_cast<chrono::milliseconds>(
        chrono::system_clock::now());
    time_t timestamp = tp.time_since_epoch().count();
    return timestamp;
}

//对传入的图像做完整的卷积操作，并返回人脸的概率
vector<float> CNN(const string &path)
{
```

```
    Mat img = imread(path);
    //转换成float
    img.convertTo(img, CV_32FC3, 1.0 / 255);
    Matrix<float> input1, Relu1, MaxPooling1, input2, Relu2, MaxPooling2, input3, Relu3, finalResult;
    imgToMat(img, input1, conv_params[0].kernel_size, conv_params[0].pad, conv_params[0].stride);
    //第一次卷积
    convAndRelu(input1, conv_params[0], Relu1);
    maxPooling(Relu1, MaxPooling1);
    convResToMat(MaxPooling1, input2, conv_params[1].kernel_size, conv_params[1].pad, conv_params[1].stride);
    //第二次卷积
    convAndRelu(input2, conv_params[1], Relu2);
    maxPooling(Relu2, MaxPooling2);
    convResToMat(MaxPooling2, input3, conv_params[2].kernel_size, conv_params[2].pad, conv_params[2].stride);
    //第三次卷积
    convAndRelu(input3, conv_params[2], Relu3);
    Relu3.setRows(2048);
    Relu3.setCols(1);
    fullyConnected(Relu3, fc_params[0], finalResult);
    //softmax
    float x = finalResult.getData()[0];
    float y = finalResult.getData()[1];
    float sum = exp(x) + exp(y);
    return {exp(x) / sum, exp(y) / sum};
}

int main()
{
    string path = "../samples/face.jpg";
    string picture = path.substr(11);
    time_t start = getTimeStamp();
    vector<float> ans = CNN(path);
    time_t end = getTimeStamp();
    cout << "整个卷积流程时间: " << (end - start) << " ms" << endl;
    cout << endl;
    cout << picture << ": " << endl;
    cout << "bg score: " << ans[0] << endl;
    cout << "face score: " << ans[1] << endl;
    return 0;
}
```

Part III Result & Verification

标准测试样例: bg.face :

```
cyg@CYG-PC:~/Project5/build$ ./project5
读入图像：0 ms
将图像中的数据转换成float类型：0 ms
------------------------------
第一次卷积：
imgToMat(): 1 ms
convAndRelu(): 2 ms
maxPooling(): 0ms
convResToMat(): 1ms
------------------------------
第二次卷积：
convAndRelu(): 0 ms
maxPooling(): 0ms
convResToMat(): 1ms
------------------------------
第三次卷积：
convAndRelu(): 0 ms
fullyConnected(): 0 ms
softMax: 0 ms
------------------------------
整个卷积流程时间：6 ms

bg.jpg:
bg score: 0.999999
face score: 7.28705e-07
```

图3-1

标准测试样例：face.jpg：

```
cyg@CYG-PC:~/Project5/build$ ./project5
读入图像：1 ms
将图像中的数据转换成float类型：0 ms
----------------------------
第一次卷积：
imgToMat(): 0 ms
convAndRelu(): 0 ms
maxPooling(): 0ms
convResToMat(): 1ms
----------------------------
第二次卷积：
convAndRelu(): 0 ms
maxPooling(): 0ms
convResToMat(): 0ms
----------------------------
第三次卷积：
convAndRelu(): 0 ms
fullyConnected(): 0 ms
softMax: 0 ms
----------------------------
整个卷积流程时间：7 ms

face.jpg:
bg score: 5.20337e-05
face score: 0.999948
```

图3-2

其中 bg score 为背景的概率，face score 为人脸的概率，可以看出该测试是非常准确且高效的。为了进一步验证该模型的准确性，我又做了一些图片的验证，所有结果如下：

测试用例	说明	bg score	face score	时间(X86)	图片
bg.jpg	标准测试样例(128*128)	0.999999	7.28705e-07	6ms	
face.jpg	标准测试样例(128*128)	5.20337e-05	0.999948	7ms	
ysq.png	于老师个人主页照片(1024*1024)	2.43622e-09	1	27ms	
windows.png	Windows7默认背景(313*500)	0.98667	0.0133299	10ms	
HuaShan.jpg	华山风景区(451*679)	0.999643	0.000356841	12ms	
sustech.jpg	南科大风景图(400*600)	1	4.11113e-14	10ms	

从上述实验可以看到，该模型基本能实现检测人脸的功能，准确且高效。

Part IV Test on ARM

标准测试样例：bg.face ：

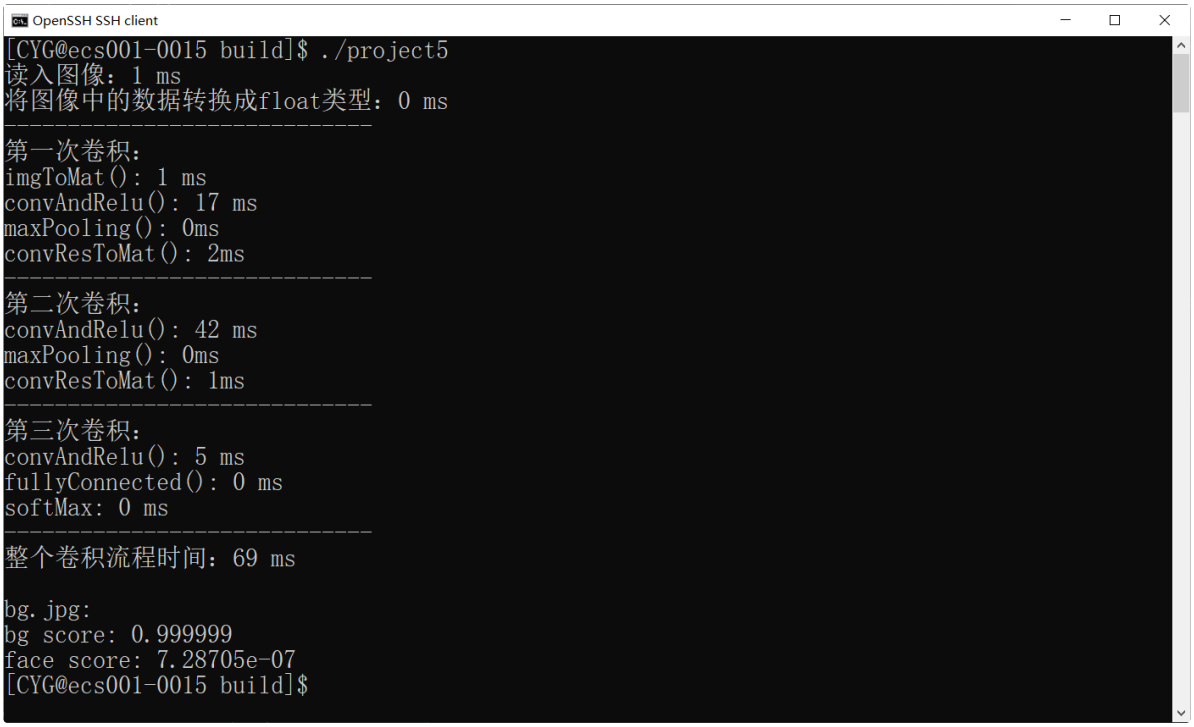

图4-1

标准测试样例：face.jpg：

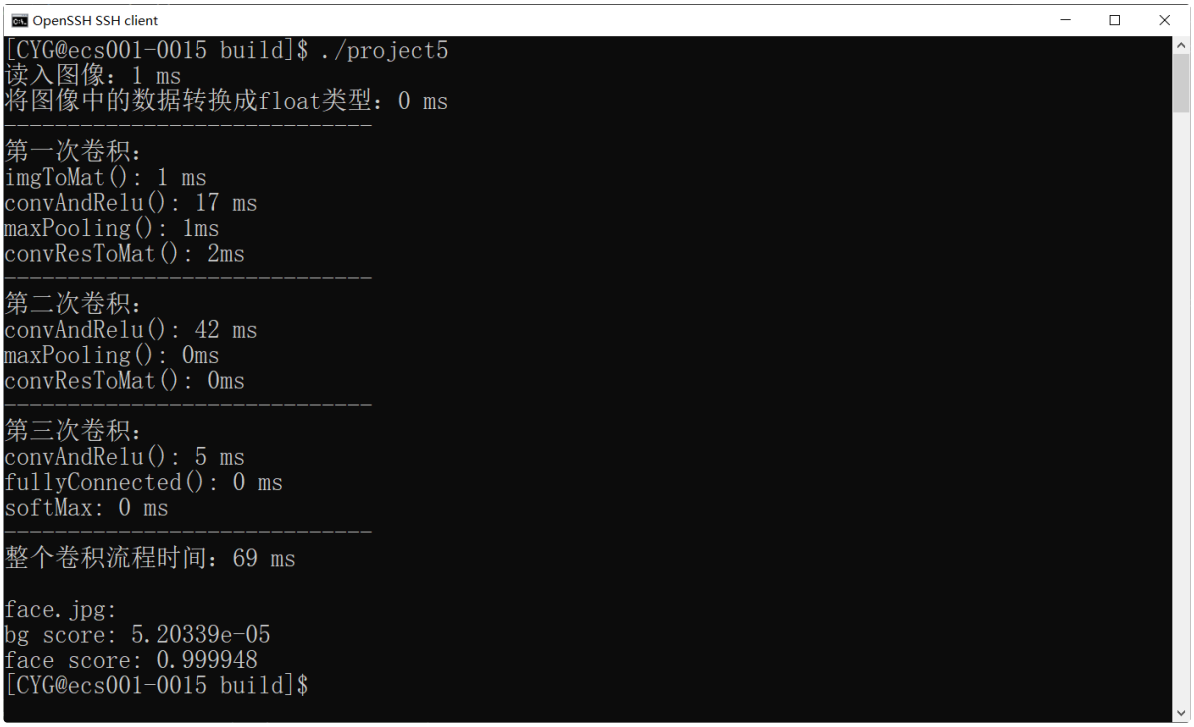

图4-2

同样在ARM上做了和在X86上相同的测试，将测试出来的结果和 X86 进行对比，整个结果如下：

测试用例	bg score(X86)	bg score(ARM)	face score(X86)	face score(ARM)	时间(ARM)
bg.jpg	0.999999	0.999999	7.28705e-07	7.28705e-07	69ms
face.jpg	5.20337e-05	5.20339e-05	0.999948	0.999948	69ms
me.png	8.9297-08	8.92974e-08	1	1	72ms
ysq.png	2.43622e-09	2.43623e-09	1	1	88ms
windows.png	0.98667	0.98667	0.0133299	0.0133299	72ms
LinDan.png	1.83618e-09	1.83618e-09	1	1	72ms
HuaShan.jpg	0.999643	0.999643	0.000356841	0.000356842	74ms
sustech.jpg	1	1	4.11113e-14	4.11114e-14	73ms

我们可以看到，在两个系统下运行相同的代码得到的结果基本相同，不过有的结果略有一点差异，这可能是因为不同平台上内部的指令集不同造成浮点数的运算不精确(计算机运行浮点数从来不精确)，可以验证本程序跨平台运算的正确性。其中 X86 系统运行速度比 ARM 系统快很多，这是因为 X86 系统因考虑到各种适应的需求，内部的复杂度很高，有着庞大的指令集，其中的 CPU 做运算时很快。而 ARM 系统则是致力于低功耗、高性能的目的，非常简洁，相应运算速度就会比 X86 慢。

Part V Difficulties & Solutions

5-1 CMake

最开始本程序在我本机的 Ubuntu 上运行时，当时我的 CMakeLists.txt 是这么写的：

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.12)
project(project5)

set(CMAKE_CXX_STANDARD 20)

add_executable(project5 main.cpp)

find_package(BLAS REQUIRED)
if(BLAS_FOUND)
    message("OpenBLAS found.")
    include_directories(/opt/OpenBLAS/include/)
    target_link_libraries(project5 ${BLAS_LIBRARIES})
endif(BLAS_FOUND)

find_package(OpenCV REQUIRED)
include_directories(${OpenCV_INCLUDE_DIRS})
target_link_libraries(project5 ${OpenCV_LIBS})
set(BLA_VENDER OpenBLAS)
```

当我在 ARM 服务器上下载并安装好了 OpenBLAS 后(OpenBLAS同样在opt目录下)，同样在 build 文件夹下执行 cmake .. 指令，会报出如下错误：

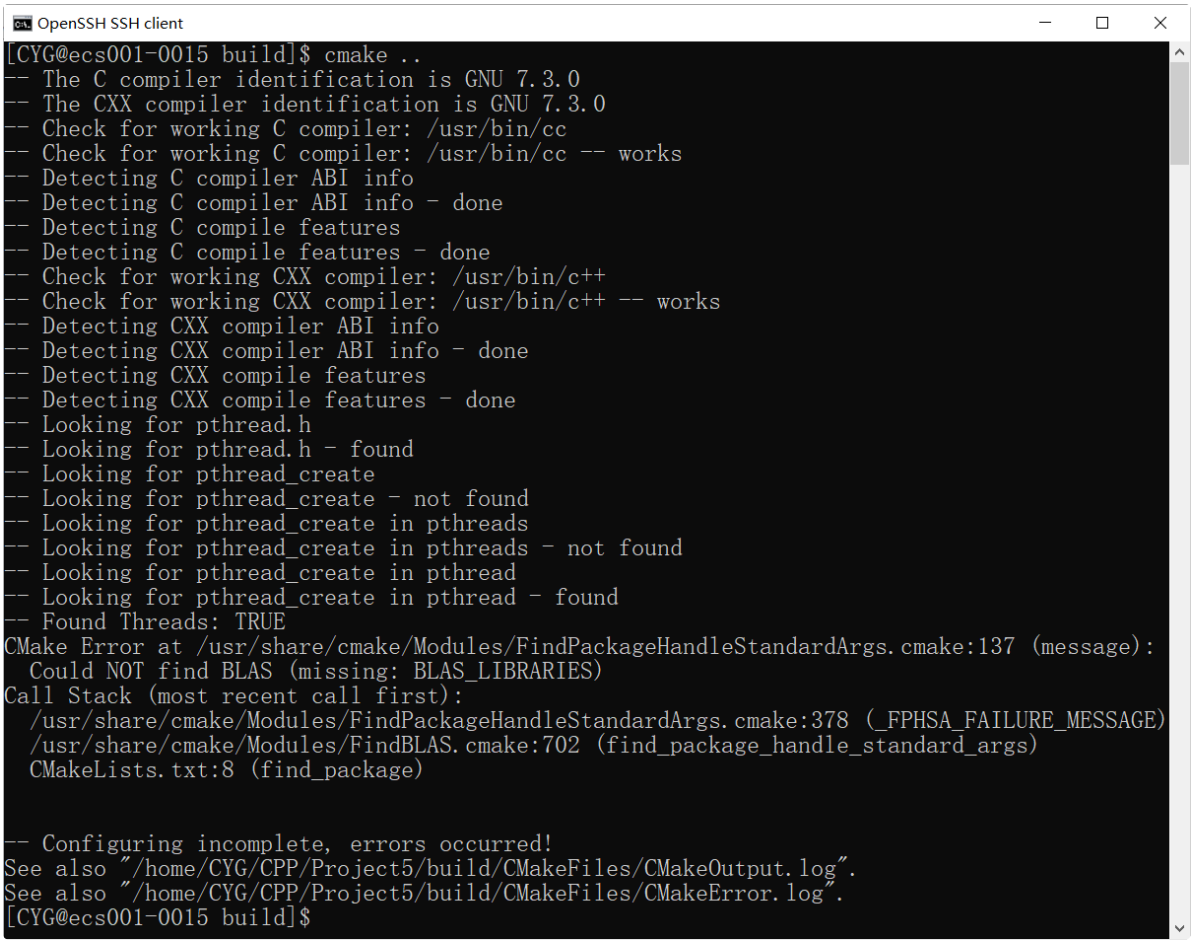

图5-1-1

该错误提示当前找不到 OpenBLAS，我再将该目录下的 include 文件夹的权限给当前用户，还是不行。不过我在网上搜索到了一种解决办法，就是将上面 CMakeLists.txt 的第8行至第13行关于引入 OpenBLAS 的代码删掉，cmake 编译时采用**cmake -DBLAS=Open ..** 的方式进行编译就正常了：

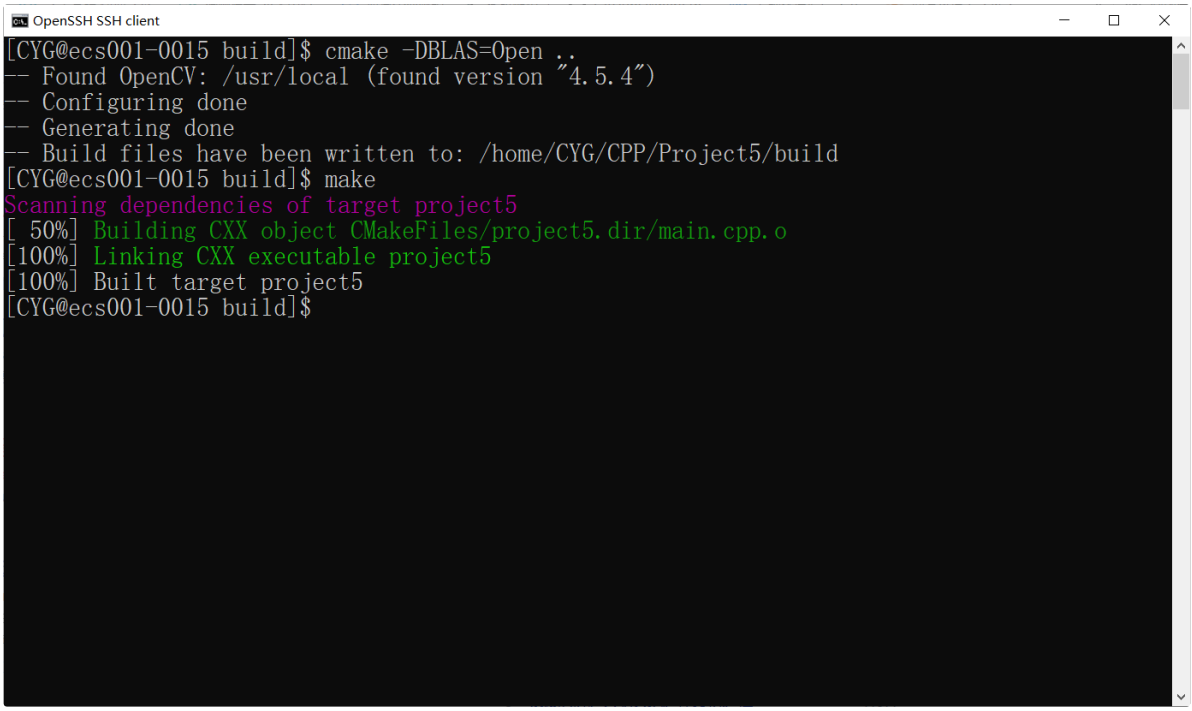

图5-1-2

5-2 The Multiplication of Matrix

我本来想沿用上次 Project 做好的矩阵乘法的代码部分，但是我发现这一次的训练数据全是给定的静态数组。如果继续用上次的矩阵乘法，就会在析构函数中调用 release() 函数时将该数组 delete 掉，而静态数组是没办法用 delete 的。一种解决方法就是定义一个 Matrix 类型的矩阵m，将m.data用new的方式开辟内存，然后将训练数据一个一个的拷贝到m.data中。不过我觉得这样重新创建一个矩阵并进行深拷贝比较耗时，就采用了另一种做法，重新定义了一个做矩阵乘法的函数：**void mul(T *weight, T *bias, size_t row, size_t col, Matrix &out, bool flag) const;** (具体可见 matrix.hpp 35行)来计算当前矩阵(Matrix类型)和传进来的 weight 数组(float *类型)做乘法。

5-3 Private member

本次 Project 我将模板类 Matrix里的属性都设置成私有的，如果要获取或修改属性只能通过 get/set 函数。但对于指针类型的 data，只用普通的 **T *getData()** 函数的话，如果我们想在类外对 matrix 中的 data 创建并初始化，**matrix.getData() = new float[length];** 这样做是有问题的，因为 getData()函数只是返回指针的值而已，不能在表达式中作为左值使用。我的解决办法就是将这个函数的返回值设置成指针的引用，这样就可以作为左值去创建并初始化 data 了。

Part VI Summary

本次 Project 是整个 C/C++ 程序设计的最后一个Project，实现了一个很简单的卷积神经网络模型。由于本次 Project 临近期末，时间紧张，刚开始看这道题目觉得无从下手。但是在经过网上很多有关 CNN 的介绍以及于老师给的整个卷积流程图后，我发现这次 Project 并没有多么难，思路非常清晰，同时也加深了对我理解 C/C++ 这门语言的理解。有时间的话还可以将按卷积定义实现的代码和本次用im2col方法实现的代码进行结果的对比。在此非常感谢于老师这一学期细致的讲解，让我对C/C++这门语言，甚至是计算机底层的一些知识都有一定的了解，同时也希望老师后续能在网课中加入这学期在课堂上没时间讲的一些知识。我相信这门课对我以后的学习一定会大有裨益。

References

- [1] 卷积神经网络（CNN）详解
- [2] caffe im2col详解
- [3] (Linux命令行) Cmake使用OpenBLAS编译Caffe
- [4] 卷积计算细节：矩阵乘法实现卷积
- [5] 如何实现高速卷积？深度学习库使用了这些"黑魔法"
- [6] CNN Explainer