CS205 C/ C++ Programming Project 5: A Simple CNN Model

Name: 眭和(SUI He)

SID: 12012929

Requirement

Please implement a simple convolutional neural network (CNN) model using C/C++. You can follow https://github.com/ShiqiYu/SimpleCNNbyCPP where a pretrained CNN model is provided. The model contains 3 convolutional layers and 1 fully connected layer. The model can predict if the input image is a person (upper body only) or not. More details about the model can be found at SimpleCNNbyCPP web site.

You are welcome to implement more CNN layers, and to make the implemented CNN to be more general (such as the convolutional layer can be for any size of kernels, not just 3x3).

Do not use any third-party library except OpenCV to read image data from image files. You should implement all CNN layers using C/C++.

Part 1 - Analysis

本次project目标实现一个简单的卷积神经网络模型，最终可以识别人脸。其中预训练模型包括3个卷积层，2个池化层和1个全连接层。此卷积神经网络模型主要由图片读取与转化、卷积、池化、全连接等方法实现，其中数据的存储与运算基于上次project实现的Matrix类（调整了数据的存储方式，将行，列，通道的顺序改为了通道，行，列）。

图片的读取与转化

首先使用OpenCV读取图片信息，OpenCV读取的图片像素颜色通道顺序是BGR, 而模型的数据是RGB，因此需要先将通道顺序转变为RGB。又因读入的数据是`unsigned char`类型，还需要将数据转换成$[0.0, 1.0]$的`float`类型数据。

由于模型只支持$128 * 128$的图片，为了增加模型的通用性，对不是$128 * 128$的图片用OpenCV的resize方法将其转换成了$128 * 128$的图片后，再进行下一步的计算。

卷积

我首先根据卷积的定义实现了一个通过滑动卷积核窗口进行计算的方法，实现了可以支持不同大小的卷积核，以及不同pad, stride, 输入通道数和输出通道数的通用的一个函数。

实现过程为首先根据参数确定输出矩阵的规模，然后依次计算输出矩阵中每一位元素的值。即依次加上窗口内对应位置元素与卷积核对应位置元素的乘积。但是因为有填充0的情况，因此需要记录原矩阵当前待计算位置的坐标。起始的坐标为$(-pad, -pad)$, 每次需要判断坐标是否在矩阵的范围内，若不在则进入下一个循环。在的话则将乘积加到输出矩阵对应位置。在所有通道的卷积核窗口内每个位置都计算完后再加上对应的bias，并在此时完成ReLU（若值小于0，则让其等于0）。之后根据stride的值将窗口向右滑动。一行计算完后，根据stride的值将窗口向下滑动，并从左向右计算。依此类推。

同时，为了提升卷积运算效率，绝大部分计算都是运用指针进行计算，因为指针移动操作较多，可能实现过程看起来较为复杂，但这也非常大程度上提高了运算效率。

即便是这种暴力算法，整个人脸检测过程运行下来也仅需**10ms**左右。以下为实现代码：

```
Matrix<float> ConvBNRelu_BruteForce(Matrix<float> &matrix, conv_param
&param)
{
    if (matrix.channel != param.in_channels)
        throw "Invalid conv_param!";
    Matrix<float> output((matrix.rows - param.kernel_size + 2 * param.pad +
param.stride) / param.stride, (matrix.cols - param.kernel_size + 2 *
param.pad + param.stride) / param.stride, param.out_channels);
    size_t outChannel = output.channel;
    size_t outRow = output.rows;
    size_t outCol = output.cols;
    size_t size = matrix.rows * matrix.cols;
    size_t weight_size = param.kernel_size * param.kernel_size;
    float *weight_ptr;
    float *out_ptr = output.data;
    float *in_ptr;
    for (size_t channel = 0; channel < outChannel; ++channel)
    {
        weight_ptr = param.p_weight + (channel * param.in_channels) *
weight_size;
        long long row_index = -param.pad;
        long long col_index = -param.pad;
        for (size_t row = 0; row < outRow; ++row)
        {
            for (size_t col = 0; col < outCol; ++col)
            {
                for (size_t i = 0; i < param.kernel_size; ++i)
                {
                    for (size_t j = 0; j < param.kernel_size; ++j)
                    {
```

```
                        if (row_index >= 0 && row_index < matrix.rows &&
col_index >= 0 && col_index < matrix.cols)
                        {
                            in_ptr = matrix.data + row_index * matrix.cols +
col_index;
                            for (size_t counter = 0; counter <
param.in_channels; ++counter)
                            {
                                *out_ptr += *weight_ptr * *in_ptr;
                                in_ptr += size;
                                weight_ptr += weight_size;
                            }
                            weight_ptr -= param.in_channels * weight_size;
                        }
                        ++col_index;
                        ++weight_ptr;
                    }
                    ++row_index;
                    col_index -= param.kernel_size;
                }
                *out_ptr += param.p_bias[channel];
                *out_ptr = max(0.f, *out_ptr);
                col_index += param.stride;
                row_index -= param.kernel_size;
                ++out_ptr;
                weight_ptr -= weight_size;
            }
            row_index += param.stride;
            col_index = -param.pad;
        }
    }
    return output;
}
```

但实际上卷积操作可以经过处理转换为矩阵乘法以更高效计算，这将在第二部分介绍。

池化

在这一阶段进行max pool操作， 将$C \times H \times W$的矩阵变为$C \times H/2 \times W/2$的矩阵。输出矩阵为原矩阵对应位置相邻四个值的最大值。其中运算也全部是用指针实现，实现过程较为简单。

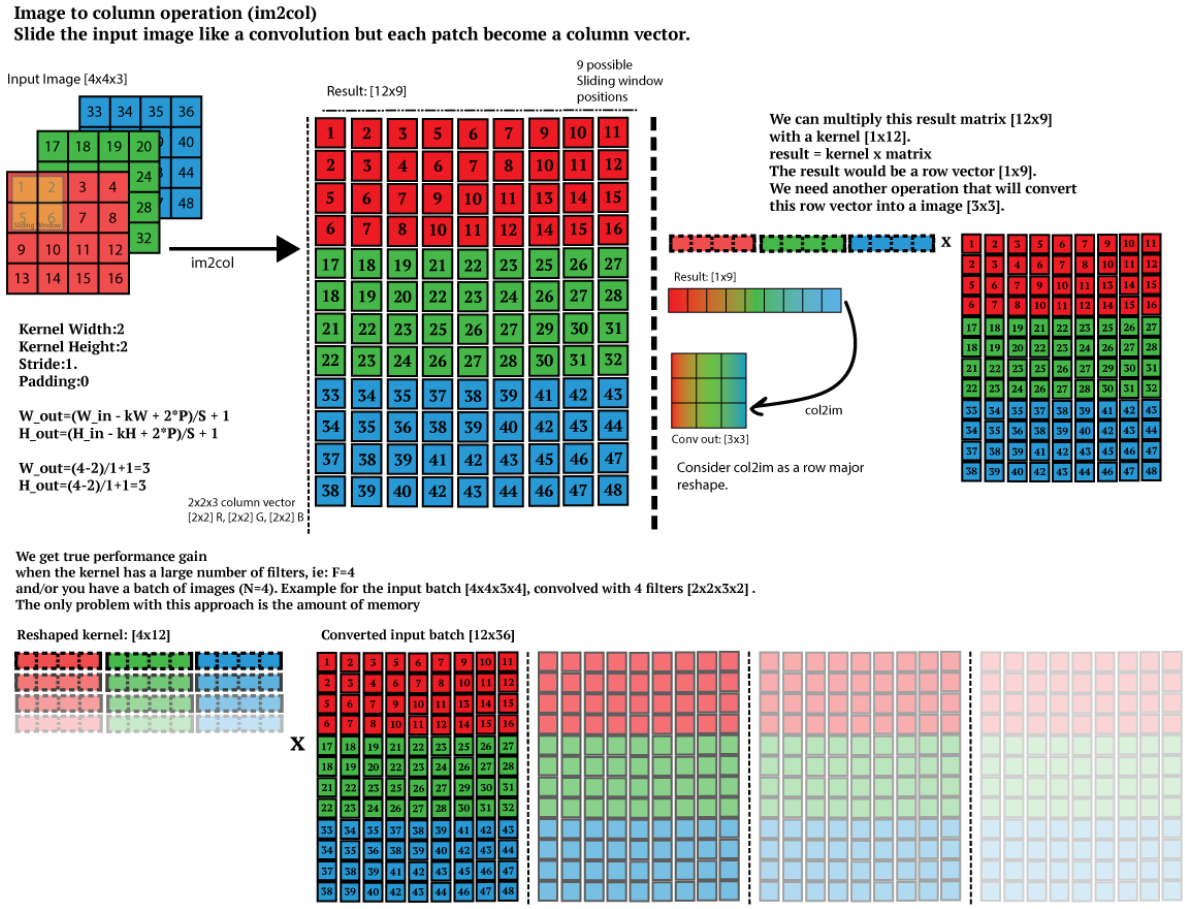
全连接层

这一部分将输入矩阵拉伸成为一个$m \times 1$的矩阵，由于输入的矩阵是一维存储的，且已经是按照通道，行，列的顺序存储的，因此只需要将矩阵的行，列，通道数进行更改。之后可以通过矩阵乘法和矩阵加法进行实现。在本模型中，输出为 $output_{2\times1} = weight_{2\times2048} * input_{2048\times1} + bias_{2\times1}$

Part 2 - Difficulties & Solutions

利用矩阵乘法计算卷积

考虑到转换成矩阵乘法可以连续进行计算，减少内存访问的时间，在一定程度上能提高卷积的效率。但是需要先对输入矩阵进行一个预处理，生成一个新的矩阵。转换如图：



如图所示，先将输入矩阵根据按照图中方法展开为一个新的矩阵。同时，kernel的矩阵也需要改变为图中所示形状，由于kernel数组的存储已经是这样排布的，因此只需要改变矩阵的形状即可。易于观察得出，这样的目的是可以根据卷积的特性让他转换为两个矩阵的乘积。因此只需要计算新的两个矩阵的乘积即可。而矩阵乘法计算过程是连续读取的，而不像第一种方法需要多次跳转，理论上能提升计算速度。

其中输入矩阵的转换通过`im2col`函数转换，对新矩阵从左到右，从上到下赋值。此外，同时记录row_index和col_index以判断什么时候需要补0。转换也直接利用指针的移动赋值，实现较为高效：

```
Matrix<float> im2col(Matrix<float> &im, int pad, int stride, int
kernel_size, int in_channels)
{
    size_t rows = im.rows;
    size_t cols = im.cols;
    size_t out_row = (im.rows - kernel_size + 2 * pad + stride) / stride;
    size_t out_col = (im.cols - kernel_size + 2 * pad + stride) / stride;
    Matrix<float> output(kernel_size * kernel_size * in_channels, out_row *
out_col);
    const float *data_in = im.data;
    float *data_out = output.data;
    size_t size = rows * cols;
    long long row_index;
    long long col_index;
    size_t temp;
    for (int channel = 0; channel < in_channels; ++channel)
    {
```

```
        for (int kernel_row = 0; kernel_row < kernel_size; ++kernel_row)
        {
            for (int kernel_col = 0; kernel_col < kernel_size; ++kernel_col)
            {
                row_index = -pad + kernel_row;
                for (size_t i = 0; i < out_row; ++i)
                {
                    if (row_index >= 0 && row_index < rows)
                    {
                        temp = row_index * cols;
                        col_index = -pad + kernel_col;
                        for (size_t j = 0; j < out_col; ++j)
                        {
                            if (col_index >= 0 && col_index < cols)
                                *(data_out++) = data_in[temp + col_index];
                            else
                                *(data_out++) = 0;
                            col_index += stride;
                        }
                    }
                    else
                    {
                        for (size_t counter = 0; counter < out_col;
++counter)
                            *(data_out++) = 0;
                    }
                    row_index += stride;
                }
            }
        }
        data_in += size;
    }
    return output;
}
```

在转换后进行矩阵相乘并加上对应的`bias`即可高效率得到卷积结果。

两种实现方式时间比较

经过多次实验取平均值，计算普通方法和矩阵乘法两种实现方式的时间如下**（单位：μs)**:

普通方法实现	矩阵乘法实现
10295μs	2581μs

通过普通的暴力方法进行卷积运算已经能够快速计算出结果，但是发现先预处理矩阵后使用矩阵乘法能在更短的时间内完成运算。实现的时间是原来的$1/4$左右，由此可见在大量运算的时候连续计算减少内存访问时间的必要性。转换成矩阵乘法后即可连续读取元素进行运算，对运算效率提升巨大。

调用OpenBLAS的`cblas_sgemm`进行提速

考虑到矩阵OpenBLAS库可以对矩阵乘法进行大量的优化，且**Caffe**对卷积的高效实现也是通过`im2col`和OpenBLAS进行优化的。因此让OpenBLAS进行矩阵乘法的运算。

arm服务器上运行的比较

程序在两个平台都可兼容，且最终计算出的结果均完全一致。时间上的比较见下表：

平台	普通方法实现	普通矩阵乘法实现	OpenBLAS矩阵乘法实现
X86	10295μs	2581μs	2191μs
arm	10613μs	2610μs	1648μs

在arm服务器上发现运行时间均比较稳定，而WSL下运行时间较不稳定，有时有数毫秒的波动。另外，普通方法和普通矩阵乘法实现arm均只略微慢于X86。但调用OpenBLAS后，运行速度却显著快于X86，说明arm在多线程的密集型运算中优势还是非常的明显。

异常处理

若图片路径不存在OpenCV会直接终止程序。以及若传入的参数与矩阵不匹配，也应及时处理而不继续进行后续计算。因此增加了在对应错误发生时的异常处理，抛出相应的异常并在main内捕捉这些异常并进行及时处理防止程序直接终止。

Part 3 - Result & Verification

Test Case 1: 测试样例face.jpg

Background possibility: 3.76486e-09
Face possibility: 1
Consuming Time: 1657μs

Test Case 2: 测试样例bg.jpg

Background possibility: 1
Face possibility: 3.55696e-07
Consuming Time: 1458μs

Test Case 3: 于老师主页照片

Background possibility: 9.78287e-09
Face possibility: 1
Consuming Time: 1609μs

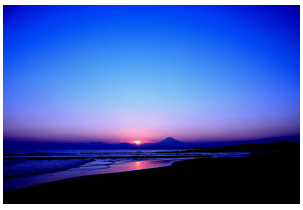
Test Case 4: Biden人脸图片

Background possibility: 0.00650709
Face possibility: 0.993493
Consuming Time: 1489μs

Test Case 5: 网络人脸图

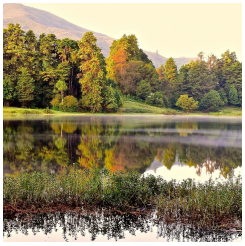
Background possibility: 6.07387e-08
Face possibility: 1
Consuming Time: 1660μs

Test Case 6: 风景图片



Background possibility: 0.999975
Face possibility: 2.44821e-05
Consuming Time: 1459μs

Test Case 7: 风景图片



Background possibility: 1
Face possibility: 2.38557e-13
Consuming Time: 1643μs

Test Case 8: 风景图片

Background possibility: 1
Face possibility: 2.73025e-09
Consuming Time: 1520μs

由测试结果可见，程序对人脸识别准确率非常高。

Part 4 - Code

代码由main.cpp, cnn.cpp, cnn.hpp, matrix.cpp, matrix.hpp, CMakeLists.txt六个文件组成。其中

main.cpp为人脸检测的入口

cnn.hpp为CNN模型方法以及参数结构体的定义

cnn.cpp为CNN模型方法的具体实现

matrix.hpp为矩阵结构体及方法的定义

matrix.cpp为矩阵方法的具体实现

CMakeLists.txt中链接了OpenCV库以及OpenBLAS库，并添加了使用OpenBLAS的宏定义（若不使用可注释掉）

以上代码已上传至Github: https://github.com/He-Sui/CS205-2021Fall/tree/main/Project5