CS205 C/ C++ Programming - Project5

name: 廖铭骞

SID: 12012919

Part1 - Analysis

CNN基本组成

张量（tensor）

在CNN当中，张量可以看作是一个三维的矩阵。对于张量来说，一般有三个维度，分别代表展张量的高度H、宽度W以及通道数C，对于一张图片来说，通道数就代表着颜色通道的数量，对于彩色图像来说，颜色通道数为3，而对于灰度图像来说，颜色通道为1。

神经元和神经层（neuron and layer）

对于CNN来说，神经元可以看做一个将多个输入转化为单个输出的函数，而神经层是一个执行相同操作的神经元的集合

核权重以及核偏差（kernel weights and biases）

每一个神经元的核权重与偏差都是唯一的，并且会随着神经网络的训练而不断调整，而在图像处理当中，不同的权重往往代表不同的特征，通过使用固定大小的权重矩阵与图像比对匹配可以提取出相应的特征。

CNN层次结构

输入层（Input Layer）

在本程序当中，输入层即为三维的彩色图像，包含RGB三个颜色通道，是一个长度宽度分别为H和W组成的三维像素矩阵

卷积层（Convolutional Layers）

卷积层通常用作对输入层输入数据进行特征提取，卷积操作的原理实际上就是对两个矩阵进行对应元素点乘求和的过程，其中两个矩阵分别是输入的像素矩阵以及卷积核矩阵。

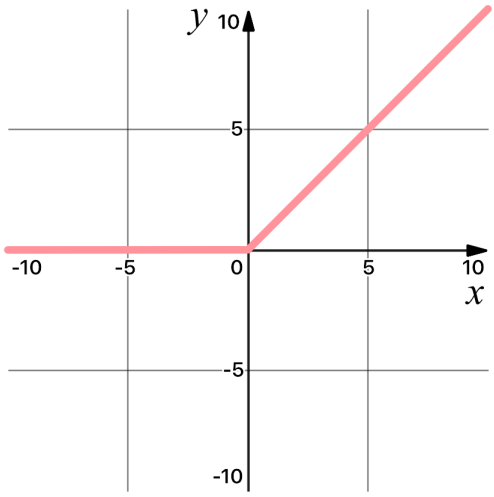
对于图像张量而言，卷积是在图像当中利用核/过滤器进行操作，每次卷积计算之后都会缩小图像的尺寸。假设原图是 $n \times n$的输入矩阵，核为 $f \times f$，则进行卷积运算之后，得到的矩阵大小为 $(n-f+1) \times (n-f+1)$，为了使得卷积之后图像张量的尺寸能按照目标尺寸进行转化以及减少图像边缘信息的损失，可以在原图像张量周围填充像素点，填充数量p满足关系$n-f+1+2p=n$，解得$p=\frac{f-1}{2}$，而核通常为奇数，便于找到核所在的位置，并且p可以被整除得到。假设卷积步长是s,则卷积后的矩阵大小为$\frac{(n-f+2p)}{s}+1$

激活层（Activative Layer）

激活层主要由激活函数组成，一般是在卷积层的输出结果矩阵之上使用一个非线性的激活函数。

如果在输出结果矩阵之上是使用的是线性的函数，那么，无论卷积神经网络有多少层，输出的结果矩阵永远是输入矩阵的线性组合，由于数据大部分情况并不是线性可分的，所以使用线性函数进行训练效果不佳，并且这样深度的CNN架构也会退化成单一的等效卷积层，这时候可以通过使用非线性激活函数进行对数据的变换，使其接近线性可分。

在CNN当中通常使用ReLU激励函数来组成激活层，相比于其他的激励函数，ReLU在训练当中可以更快地收敛、更有效率地梯度下降以及反向传播。



池化层（Pooling Layers）

池化层可以对输入矩阵的特征进行筛选，提取区域内最具代表性的特征，能够有效地降低输出矩阵的大小，进而减少网络的空间范围，从而减少神经网络的参数和整体计算，在本程序的实现当中，使用最大化池化方法，通过提取出一块区域中最大的值来代表这一块区域，通过减少不必要的参数来提升神经网络计算的效率

扁平层（Flatten Layer）

在神经网络的最后是展平层，该层将神经网络中的输出三维矩阵转换成一维向量，通过将之前网络卷积层从输入张量（图像）提取的特征通过使用`softmax`函数进行分类。

图像读入与转换

通过使用`OpenCV`中的`imread`方法将一个`128*128`的彩色图像进行读入。

由于通过文件读取图像的时候可能由于文件缺失或是权限不允许等原因导致读取失败，此时如果程序继续运行就会导致一些列危险的错误，所以有必要在读取文件之后对读取是否成功做检查，即判断`Mat::data`是否为空指针，如果是空指针，则报错并退出程序。

为了确认读入图像的通道数以及长度和宽度，使用`channel()`以及`size()`方法获取图像的相关信息。

如果读入图像的通道数，长度或者宽度不符合条件，则会导致后续运算出现错误，所以为了保证程序运算的正确性，同样需要进行相关信息的检查。

同时，为了便于之后的一系列操作，需要将字符类型矩阵`Mat`使用`project4`中实现的多通道矩阵类型进行存储，由于在OpenCV当中的通道顺序是`GBR`，而需要进行运算的顺序是`RGB`，所以需要将`Mat`当中的信息转存到`Matrix`类当中，并将相应的元素进行赋值。

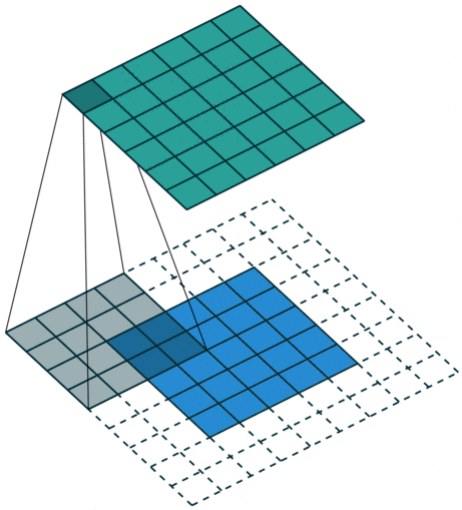
相关代码如下所示:

```
    Mat img = imread("face.jpg");

    if(!img.data){//读入图片失败的检查
        cerr <<"Error in file " << __FILE__ <<" while loading image in line " <<
__LINE__ << endl;
        exit(EXIT_FAILURE);
    }else{
        cout << "Successfully loading image" << endl;
    }

    if(img.rows != ROW || img.cols != COL || img.channels() != CHANNEL){//确保矩阵
规模正确的参数检查
        cerr << "Image Size Mismatch in File " << __FILE__ << " in line " <<
__LINE__<< endl;
        exit(EXIT_FAILURE);
    }

    Matrix<float> mat(ROW, COL, CHANNEL);//创建行数、列数均为128，通道数为3的浮点数矩阵
    //将BRG通道转换存储为RBG通道
   for(int r = 0; r < ROW; r++){
        for(int c = 0; c < COL; c++){
            mat(r, c, 2) = img.at<Vec3b>(r, c)[1] / 255.0;//规范化//G
            mat(r, c, 3) = img.at<Vec3b>(r, c)[0] / 255.0;//B
            mat(r, c, 1) = img.at<Vec3b>(r, c)[2] / 255.0;//R
        }
    }
```

卷积层分析与实现

将图像张量信息转化正确之后，下面进行卷积层的运算。

下图形象地展示了卷积层的实现原理，即通过将原图像的一块区域分别与滤波器`filter`（一个带有不同权重值的矩阵）做内积后等到新的二维数据，作为新的矩阵。（如下图所示）



单通道单步长卷积运算分析

首先考虑最基础的情况，先通过两层循环定位到图像矩阵的起始运算点$point(x,y)$(对应进行卷积区域的左上角的点)，再通过两层循环遍历滤波器的每一个点$kernel(i.j)$(对应卷积核左上角的点)，分别进行点积运算，将对于同一个运算点$point(x,y)$的点积结果累加作为输出矩阵对应位置的值。

为了保持矩阵运算前后的规模不发生改变，会采取在矩阵周围填充`0像素点`的操作，对于有填充的情况，进行点积运算可以直接忽略填充`0像素点`的位置，即在运算前先判断运算区域点是否在原矩阵的有效运算范围内，如果在，则进行点积运算，否则不进行相关运算，即默认将滤波器对应元素乘以0。

单通道多步长卷积运算分析

进一步地，分析步长不为`1`的情况，此时，矩阵起始运算点$point(x,y)$的移动并不是逐个移动遍历了，而是每次加上步长的"跳步"遍历，设步长为$stride$，则$point(x,y)$对应结果矩阵的位置为$point'(x/stride, y/stride)$。而点积运算依然与基础情况一样，经过验证，通过上述公式对应到结果矩阵的点恰好是连续的，且对应的值也是正确的。

多通道卷积运算分析

在多通道矩阵卷积运算下，输出通道的每一个结果矩阵都是所有传入通道矩阵与对应卷积核的点积结果矩阵累加并在相应区域加上偏置（bias）的结果。

所以在最外层的循环当中循环的是输出矩阵的对应通道，之后对传入通道的每一个通道的矩阵都进行点积运算，并将点积运算之后得到的对应点累加之后的结果加上偏置（bias）之后作为输出通道矩阵的对应元素。

减少循环中计算量的优化分析

由于朴素方法当中实现卷积运算需要的循环层数非常多，则在循环当中的计算会被重复很多次，带来的开销是很大的，可以事先将对应的变量在循环外部先进行计算，这样可以有效减少在循环当中的计算以及函数的调用，提升效率。如可以将获取矩阵行数、列数以及通道数的函数调用在所有循环外层都事先进行调用获取相应值；

通过分析循环内部的冗杂计算，发现在使用给定的`weight`数组当中寻找对应通道的权重时，在寻找的时候会使用到最外两层的输出通道数以及输入通道数，如下列代码所示：

```
ans += mat(img_height + ker_height - padding, img_width + ker_width - padding,
in_ch + 1) *
                                conv.p_weight[out_ch*(in_channel*3*3) + in_ch*
(3*3) + ker_height * ker_width];
```

由于`out_ch*(in_channel*3*3) + in_ch*(3*3)`在最外层已经计算得到，如果依然放在最内层就会导致冗杂的计算被重复了很多次，造成效率的降低，所以可以在最外的两层循环就使用变量将其计算并存储。

```
for(int out_ch = 0; out_ch < out_channel; out_ch++){//输出通道数量
    //...
        for(int in_ch = 0; in_ch < in_channel; in_ch++){//传入通道数量
            //...
            int conv_move = out_ch * (in_channel * 3 * 3) + in_ch * (3 * 3);
            //for(){}...
                ans += mat(img_height + ker_height - padding, img_width +
ker_width - padding, in_ch + 1) *
                                conv.p_weight[conv_move + ker_height *
ker_width];
```

传入参数合法性检查

在卷积运算当中，矩阵的相对大小关系对结果矩阵的运算的准确性有着非常关键的影响，所以在对传入参数进行计算之前需要对其进行合法性的检查。

首先需要检查传入的用于存放输出通道矩阵的矩阵数组是否为空指针，如果该数组为空指针，则在对其中元素进行赋值时会发生段错误导致程序异常终止，所以需要检查空指针并在出现空指针时及时提示用户。

其次是在逻辑层面的检查，输出通道数必须与传入的矩阵数组的规模相匹配，卷积核的传入通道数也必须与传入矩阵的通道数相一致，通过检查传入参数的逻辑是否符合要求可以在函数的起始阶段就减少不必要的`debug`时间，保证了程序的稳健性。

Relu 激活函数分析与实现

由于经过`Relu`函数前后不改变矩阵的通道个数，为了节省空间，可以直接在传入的矩阵当中对每一个通道的矩阵内部的值进行修改。修改方案为，如果在原矩阵当中对应的元素小于零，则其在输出矩阵当中对应的元素映射为0，否则映射为自身。即

$$tar\_val = max(src\_val, 0)$$

池化分析与实现

池化操作不改变矩阵的通道个数，而如果原矩阵的行数或者列数为奇数，则输出矩阵的高度以及宽度在池化后会改变为原矩阵的1/2 + 1，否则输出矩阵的高度或者宽度在池化后会成为原矩阵的1/2。卷积神经网络也经常使用池化层来缩减模型的大小，提高计算速度。而在CNN当中通常使用的是最大池化的运算，最大化操作的功能就是只要在任何一个象限内提取到某个特征，它都会保留在最大化的池化输出里，并以此作为该象限的特征。

在池化的运算过程之前，依然需要进行对传入参数合法性的检查，需要检查传入矩阵的高度以及宽度以及通道数与输出矩阵的关系是否正确对应，如果出现不对应的情况，则说明不能进行正常的池化操作，此时提示相关逻辑错误信息。

在具体实现中，由于在本实现当中最大池是一个2 * 2 的矩阵，所以在最外层两层循环定位到需要计算区域的左上角点，在最内层的两层循环以该点为运算起始点进行最大池运算，并通过

$$output\_cur\_height = (src\_cur\_height + 1)/step$$
$$output\_cur\_width = (src\_cur\_width + 1)/step$$

的关系(此时step为2)得出待填充输出矩阵点的对应坐标并进行赋值，考虑原矩阵大小存在奇数的情况，故在进行计算时需要取原矩阵对应点加一除以`step`作为输出矩阵点的对应坐标。

考虑程序的鲁棒性，当进行最内两层循环时，有可能出现已经计算点越界情况，此时不将越界的点纳入对该区域最大值的计算当中。

全连接分析与实现

由于全连接层的操作对原矩阵的逐个像素进行运算，而展开层则是逐个对像素进行拷贝存放在一个向量当中，所以展开层的步骤可以在全连接层通过遍历矩阵元素来进行替代，避免逐个复制像素的额外开销。

在开始运算之前，需要预先对传入的参数进行合法性检查，首先是对`in_feature`与传入矩阵的规模进行是否相等的判断，如果不相等，则代表不能进行运算，其次是对`out_feature`与传入的输出向量的规模进行相等性的检查，如果不相等，依然不能进行相应的运算。只有在以上条件都满足的情况下，才能保证实际运算当中程序运行的安全性以及最终运算结果的正确性。

具体实现则对原矩阵所有通道的值都分别乘以`full_connecting`的对应权重并累加，在对应同一`out_feature`的所有值都完成累加之后，在对应的值上加上全连接的`bias`，作为最终对应的`out_feature`的值。

Softmax 函数分析与实现

分析`Softmax`函数的作用，由于通过卷积神经网络求得的是对于不同分类的可能性的概率反映，而概率是一个非负的值，所以该函数首要的是需要将前一步运算得到的最终结果向量的所有值全部转化成为非负数，这可以通过结果向量的值转化到指数函数之上。其次需要保证所有的概率之和相加为1，所以需要对转化之后的所有值进行归一化处理，除以所有转化后结果之和，即求出该值占总值的百分比。表达成公式如下所示：

$$对于n维向量R^n=(x_1,x_2,\dots,x_n),设输出概率n维向量p^n=(p_1,p_2,\dots,p_n)$$

$$p_i=\frac{e^{x_i}}{\sum_{j=1}^{n}e^{x_j}}$$

CNN层级结构实现

对于传统的CNN，通过将上述方法分别实现成不同的函数之后，进行各函数的CNN的网络层次的搭建。

```
    //根据公式算出第一层卷积后的结果矩阵的高度以及宽度，之后的规模换算同公式
    int conv_size1 = ((ROW - conv_params[0].kernel_size + 2
                       * conv_params[0].pad) / conv_params[0].stride + 1);

    //根据计算得到的规模以及通道数创建第一层卷积得到的矩阵
    Matrix<float> conv_mat1(conv_size1, conv_size1,
conv_params[0].out_channels);

    //创建最大池化后的矩阵
    //此处+1再除以2的目的是考虑了矩阵长度或者宽度出现奇数的情况
    //使得最大池化操作不仅适用于偶数规模的矩阵,更具有普适性，之后的池化矩阵规模换算关系同理
    Matrix<float> max_pool_mat1((conv_size1 + 1) / 2,
                                (conv_size1 + 1) / 2,
conv_params[0].out_channels);

    //进行第一层卷积运算
    convolution(mat, conv_params[0], conv_mat1);
    Relu(conv_mat1);
    max_pooling(conv_mat1, max_pool_mat1);//32 * 32 * 16

    int conv_size2 = ((max_pool_mat1.getRow() - conv_params[1].kernel_size +
                       2 * conv_params[1].pad) / conv_params[1].stride + 1);

    Matrix<float> conv_mat2(conv_size2, conv_size2,
conv_params[1].out_channels);
    Matrix<float> max_pool_mat2((conv_size2 + 1) / 2,
                                (conv_size2 + 1) / 2,
conv_params[1].out_channels);

    //第二层卷积运算
    convolution(max_pool_mat1, conv_params[1], conv_mat2);
    Relu(conv_mat2);
    max_pooling(conv_mat2, max_pool_mat2);//15 * 15 * 32

    int conv_size3 = ((max_pool_mat2.getRow() - conv_params[2].kernel_size +
                       2 * conv_params[2].pad) / conv_params[2].stride + 1);
```

```
    Matrix<float> conv_mat3(conv_size3, conv_size3,
conv_params[2].out_channels);

    //第三层卷积运算
    convolution(max_pool_mat2, conv_params[2], conv_mat3);
    Relu(conv_mat3);

    //通过全连接层的out_feature计算结果向量的维数
    int vec_size = fc_params[0].out_features;
    //创建结果向量
    int* vec = new int[vec_size];
    memset(vec, 0, sizeof(float) * vec_size);

    //检查内存是否分配成功
    if(vec == nullptr){
        cerr << "Error in file " << __FILE__ <<
            " in function " << __FUNCTION__ << " in line " << __LINE__ << endl;
        cerr << "Failure in allocating memory." << endl;
        exit(EXIT_FAILURE);
    }

    //全连接层运算，将矩阵展平并转化成结果向量
    full_connect(conv_mat3, fc_params[0], vec, vec_size);
    //将结果向量转化成[0.0, 1.0]的概率区间范围
    soft_max(vec, vec_size);

    //打印最终运算结果
    printf("bg score: %f, face score: %f\n", vec[0], vec[1]);
```

计时的方式分析

由于本程序是在Linux环境下运行，所以使用Linux环境下的计时函数`gettimeofday()`进行程序的计时，该计时函数精度较高，可以达到微秒级别。
通过查看计时函数的结构可以发现，该结构体的定义为

```
struct timeval
{
  __time_t tv_sec;      /* Seconds.  */
  __suseconds_t tv_usec;    /* Microseconds.  */
};
```

这个函数获取从1970年1月1日到现在经过的时间和时区（UTC时间），但是按照Linux的官方文档，该时区已经不再使用，所以在使用的时候传入NULL即可。

在进行卷积运算之前以及卷积运算全部完成之后分别获取一次当前时间，它们之间的差值即为程序运行的时间。
代码如下：

```
    gettimeofday(&total_t1, NULL);
   //卷积的所有步骤
    gettimeofday(&total_t2, NULL);

    double total_time_use = (total_t2.tv_sec - total_t1.tv_sec) * 1000 +
        (double)(total_t2.tv_usec - total_t1.tv_usec) / 1000.0;
    cout << "The total time used in processing CNN is: "
        << total_time_use << "ms" << endl;
```

使用该种方式进行计时，在最后输出的是CNN网络全部运算时间的总和，单位是毫秒。

比较时间差距使用的方法

在比较不同算法以及优化策略时，对于随着矩阵规模的增大各算法时间的相对差距需要定义一个公式来进行对两组时间统一地比较，这里采用如下的公式进行比较：

$$\mathrm{E}=\frac{T_{long}-T_{short}}{T_{long}}\times 100\%(T_{long}\text{代表用时长的一组，}T_{short}\text{代表用时短的一组})$$

E的值越大，反映出两组所用时间的差距就越大，差距越明显，在后续的时间差距比较当中使用的都是这一公式。

矩阵乘法卷积优化

通过观察朴素实现的卷积运算可以发现，其实在卷积运算当中进行的实质上是矩阵的乘法运算，而如何正确地将一层层的`for`循环抽象提炼出来成为矩阵成为了思考上的难点。

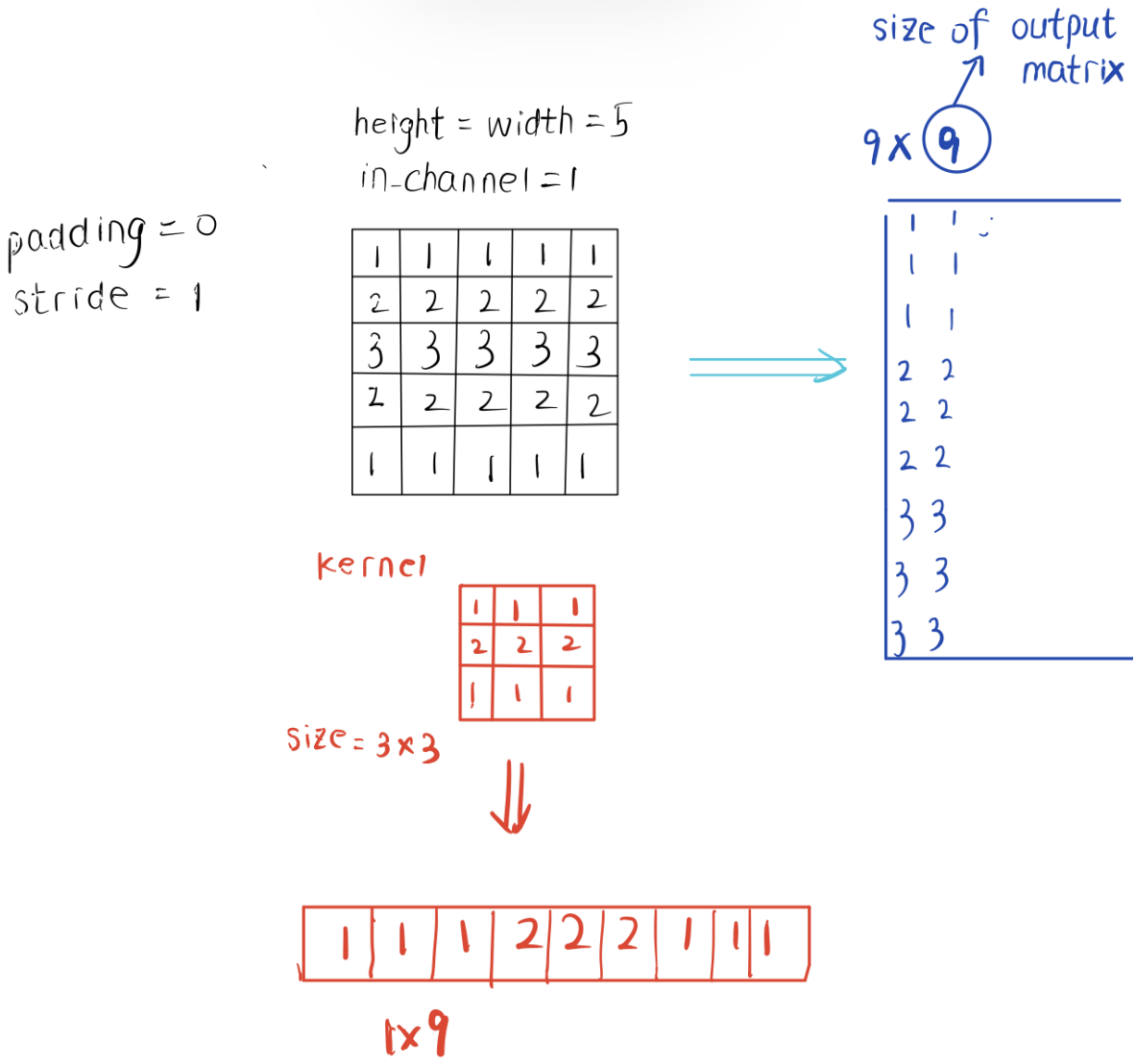
通过分析矩阵乘法的本质公式（如下）可以发现，对于最终的结果矩阵的对应通道的元素而言，它的产生其实是每

$$C_{iq}=\sum_{k=1}^{n}A_{ik}B_{kq}$$

一个输入通道的矩阵对应元素乘以对应输入通道的卷积层的权重累加加上对应偏置(bias)的结果。

以单通道矩阵为例，如图所示，假设输入为单通道的5 * 5 的矩阵，输出矩阵也为单通道矩阵，卷积核的填充为0，步长设置为1，则根据前述公式，可以算出输出矩阵的规模也为3 * 3。

根据矩阵乘法的原理，可以将卷积核同一通道的元素转为同一行，而将输入矩阵每一次需要做卷积的象限区域转化成同一列，如此便可以将输入矩阵转化成的新矩阵作为矩阵乘法的右元，卷积核转化成为的矩阵作为左元进行矩阵的乘法，而得到结果矩阵之后，每一个位置在加上对应输出通道的偏置(bias)值对应的就是实际进行卷积运算后的输出矩阵的对应元素。



核心代码与思路如下所示：

```
    Matrix<float> tmp_mat(ker_c * ker_r * in_channel, res_height * res_width,
1);
    for(int in_ch = 0; in_ch < in_channel; in_ch ++){//传入通道数量

        int span = in_ch * ker_size;//将最内层需要频繁计算的值先提前计算，提升性能

        int col_cnt = -1;
        for(int img_height = 0; img_height + ker_r <= mat_r
            + 2 * padding; img_height += stride){//图像高度
            for(int img_width = 0; img_width + ker_c <= mat_c +
                2 * padding; img_width += stride){//图像宽度
                col_cnt ++;
                for(int ker_height = 0; ker_height < ker_r; ker_height ++){//核高
度
                    for(int ker_width = 0; ker_width < ker_c; ker_width ++){//核
宽度

                        if(img_height + ker_height >= padding
                           && img_width + ker_width >= padding //判断计算点是否位于
目标范围内
                            && img_height + ker_height < padding + mat_r
                           && img_width + ker_width < padding + mat_c){
                            //如果在范围内，就将对应位置赋值为输入矩阵对应的元素
```

```
                            tmp_mat(span + ker_height * ker_c + ker_width,
col_cnt, 1) =
                                mat(img_height + ker_height - padding,
                                    img_width + ker_width - padding, in_ch + 1);
                        }else{
                            //不在范围内就将其赋值为0
                            tmp_mat(span + ker_height * ker_c + ker_width,
                                    col_cnt, 1) = 0;

                        }
                    }
                }
            }
        }
    }

    Matrix<float> ker_mat(out_channel, in_channel * ker_size, 1);
    //将核转化成行矩阵ker_mat
    for(int o = 0; o < out_channel; o++){
        for(int i = 0; i < in_channel; i++){
            int span = i * ker_size;//将内层变量提前进行计算，提升性能
            int move = o*(in_channel*3*3) + i*(3*3);
            for(int ker_height = 0; ker_height < ker_r; ker_height++){
                for(int ker_width = 0; ker_width < ker_c; ker_width++){
                    ker_mat(o, span + ker_height * ker_c + ker_width, 1) =
                    conv.p_weight[move + ker_height*ker_r + ker_width];
                }
            }
        }
    }

    Matrix<float> ans_mat = ker_mat * tmp_mat;//直接使用矩阵乘法进行卷积运算

    for(int o = 0; o < out_channel; o++){
        for(int h = 0; h < res_height; h++){
            for(int c = 0; c < res_width; c++){
                //将结果矩阵转化为输出矩阵并加上偏置
                res_mat(h, c, o + 1) = ans_mat(o, h*res_width + c, 1) +
conv.p_bias[o];
            }
        }
    }
```

通过将卷积运算转化成为矩阵乘法运算，成功将朴素卷积运算时的最多六层`for`循环提升成了至多5层`for`循环。

而两种实现方式的每一层卷积运算结果正确性也都有所保证，具体可参见下文`Part3 正确性检验`部分。

两种实现方式对于样例`face.jpg`的最终测试结果分别如下所示：

- simple method(朴素实现)

```
lmq@LAPTOP-4MG6A2H2:/mnt/c/大二课程/c++/C++Code/project5$ g++ cnn.cpp  `pkg-config --cflags --libs opencv4` && ./a.out
Background possibility: 0.000000
Face possibility: 1.000000
The total time used in processing CNN using simple method is: 43.102000ms
```

- Matrix multiplication（矩阵乘法优化）

```
lmq@LAPTOP-4MG6A2H2:/mnt/c/大二课程/c++/C++Code/project5$ g++ cnn.cpp  `pkg-config --cflags --libs opencv4` && ./a.out
Background possibility: 0.000000
Face possibility: 1.000000
The total time used in processing CNN using matrix multiplication is: 26.211000ms
```

结果均正确并且一致。

CNN程序通用性与鲁棒性分析

通用性

在本次`project`当中，虽然卷积层数有限，卷积核规模、步长以及填充数量都相对固定，但是本程序的CNN不仅仅适用于这些卷积核，具有通用性，因为每一层的计算都抽象成了一个方法，在传入参数的时候根据前述公式进行对输出矩阵的规模以及通道数的计算，在方法当中也根据该公式对输入矩阵以及对应的卷积核的规模、步长以及填充数量进行逻辑正确的卷积运算，最终可以普适地获得正确的卷积结果。

鲁棒性

在本次`project`的程序当中，在方法当中进行的参数合法性检查有效地保证了程序运行的安全性以及计算的准确性。

以卷积层为例，在卷积层当中首先对传入矩阵、输出矩阵的通道数与卷积核的输入输出通道数的匹配进行检查，保证通道的对应正确。

其次是对传入矩阵以及输出矩阵对应的元素数组进行检查，检查其是否为空指针，如果不进行这类检查，在实际获取矩阵元素的时候就有可能会造成程序崩溃的严重后果，有效保证了程序的安全性。

另外，对于卷积的运算，每一次对数组元素的访问都面临着数组越界的风险，所以在循环运行开始之前，先对循环内部可能访问到的最大范围进行检查，确保该值不会大于数组的长度才允许循环的执行，这避免了非法访问的问题。

对于矩阵类来说，每创建一个矩阵类，也会对它的元素数组是否创建成功进行检验，进行矩阵乘法之前也有对两个矩阵规模（行列数以及通道数）是否匹配的逻辑判断，保证矩阵乘法能够正确运行。

在本程序当中，随处可见都是对于参数的合法性检查，并且在出错时都有错误流的准确提示，提升了程序的鲁棒性以及调试程序的便捷性。

CMakeLists.txt 的编写以及分析

由于需要链接Opencv库进行编译使用其方法，每一次都需要执行如下图所示的长指令

```
lmq@LAPTOP-4MG6A2H2:/mnt/c/大二课程/c++/C++Code/project5$ g++  cnn.cpp  `pkg-config --cflags --libs opencv4`
```

感到很不方便，于是通过做第三次`project`的经验得知可以使用CMake进行Opencv以及OpenBLAS库的链接以及生成对应可执行文件来简化操作。

指定cmake的最低版本是3.16版本，在之后是要创建project的名字，此处名字为`cnn`，之后如果要使用到不属于当前目录的头文件，所以需要进行头文件的目录的添加，进而需要添加需要链接的头文件的目录以及名称。综合起来的CmakeLists.txt的代码如下所示：

```
cmake_minimum_required(VERSION 3.16)

project(cnn)

include_directories(/opt/OpenBLAS/include)

find_package(OpenCV REQUIRED)

include_directories(${OpenCV_INCLUDE_DIRS})

link_libraries("/opt/OpenBLAS/lib/libopenblas.so")
```

```
add_executable(cnn cnn.cpp)

target_link_libraries(cnn ${OpenCV_LIBS})
```

通过编写cmake将opencv的库以及openblas的库进行链接生成链接后的文件，在运行时只需要运行`./cnn`即可运行代码，大大简便了操作。

之后只要运行`cmake .`就可以自动实现MakeFile的生成，之后可以直接通过用make命令编译源码生成可执行程序，自动实现各文件的链接以及动态库的调用，在处理一个包含有多级文件的C语言或者C++大工程的时候，将会大大地提高管理代码的效率。

Part2 - Code

图片的读取与转换

```
    Mat img = imread("face.jpg");

    if(!img.data){//读入图片失败的检查
        cerr <<"Error in file " << __FILE__ <<
            " while loading image in line " << __LINE__ <<
            " in FUCTION " << __FUNCTION__ << endl;
        exit(EXIT_FAILURE);
    }else{
        cout << "Successfully loading image" << endl;
    }

    if(img.rows != ROW || img.cols != COL || img.channels() != CHANNEL){//确保矩阵规模正确
        cerr << "Size Mismatch of image in File " << __FILE__ << " in line " << __LINE__<< "in FUCTION " << __FUNCTION__ << endl;
        exit(EXIT_FAILURE);
    }

    Matrix<float> mat(ROW, COL, CHANNEL);//创建行数列数均为128，通道数为3的浮点数矩阵

    //在将读入的图片矩阵转化成实现过的矩阵类型
    //同时将BRG通道转换存储为RBG通道

    for(int r = 0; r < ROW; r++){
        for(int c = 0; c < COL; c++){
            mat(r, c, 2) = img.at<Vec3b>(r, c)[1] / 255.0;//规范化//g
            mat(r, c, 3) = img.at<Vec3b>(r, c)[0] / 255.0;//b
            mat(r, c, 1) = img.at<Vec3b>(r, c)[2] / 255.0;//r
        }
    }
```

卷积层次运算与计时

```
//根据公式算出第一层卷积后的结果矩阵的高度以及宽度，之后的规模换算同公式
    int conv_size1 = ((ROW - conv_params[0].kernel_size + 2 * conv_params[0].pad) / conv_params[0].stride + 1);

    //根据计算得到的规模以及通道数创建第一层卷积得到的矩阵
```

```
    Matrix<float> conv_mat1(conv_size1, conv_size1,
conv_params[0].out_channels);

    //创建最大池化后的矩阵
    //此处+1再除以2的目的是考虑了矩阵长度或者宽度出现奇数的情况，使得最大池化操作不仅适用于偶数规模的矩阵,更具有普适性，之后的池化矩阵规模换算关系同理
    Matrix<float> max_pool_mat1((conv_size1 + 1) / 2, (conv_size1 + 1) / 2,
conv_params[0].out_channels);

    timeval total_t1, total_t2;//计时
    gettimeofday(&total_t1, NULL);
    //进行第一层卷积运算
    im2col_convolution(mat, conv_params[0], conv_mat1);

    timeval relu_time1, relu_time2;
    double relu_time_use;

    gettimeofday(&relu_time1, NULL);
    Relu(conv_mat1);
    gettimeofday(&relu_time2, NULL);
    relu_time_use += (relu_time2.tv_sec - relu_time1.tv_sec) * 1000000.0 +
(double)(relu_time2.tv_usec - relu_time1.tv_usec);


    max_pooling(conv_mat1, max_pool_mat1);//32 * 32 * 16

    //第一层卷积结束

    int conv_size2 = ((max_pool_mat1.getRow() - conv_params[1].kernel_size + 2 *
conv_params[1].pad) / conv_params[1].stride + 1);
    Matrix<float> conv_mat2(conv_size2, conv_size2,
conv_params[1].out_channels);
    Matrix<float> max_pool_mat2((conv_size2 + 1) / 2, (conv_size2 + 1) / 2,
conv_params[1].out_channels);


    //第二层卷积运算
    im2col_convolution(max_pool_mat1, conv_params[1], conv_mat2);

    gettimeofday(&relu_time1, NULL);
    Relu(conv_mat2);
    gettimeofday(&relu_time2, NULL);
    relu_time_use += (relu_time2.tv_sec - relu_time1.tv_sec) * 1000000.0 +
(double)(relu_time2.tv_usec - relu_time1.tv_usec);


    max_pooling(conv_mat2, max_pool_mat2);//15 * 15 * 32


    //第二层卷积结束

    int conv_size3 = ((max_pool_mat2.getRow() - conv_params[2].kernel_size + 2 *
conv_params[2].pad) / conv_params[2].stride + 1);

    Matrix<float> conv_mat3(conv_size3, conv_size3,
conv_params[2].out_channels);

    //第三层卷积运算
```

```
    im2col_convolution(max_pool_mat2, conv_params[2], conv_mat3);

    gettimeofday(&relu_time1, NULL);
    Relu(conv_mat3);
    gettimeofday(&relu_time2, NULL);
    relu_time_use += (relu_time2.tv_sec - relu_time1.tv_sec) * 1000000.0 +
(double)(relu_time2.tv_usec - relu_time1.tv_usec);


    //第三层卷积结束

    //通过全连接层的out_feature计算结果向量的维数
    int vec_size = fc_params[0].out_features;
    //创建结果向量
    float* vec = new float[vec_size];
    memset(vec, 0, sizeof(float) * vec_size);

    //检查内存是否分配成功
    if(vec == nullptr){
        cerr << "Error in file " << __FILE__ << " in function " << __FUNCTION__
<< " in line " << __LINE__ << endl;
        cerr << "Failure in allocating memory." << endl;
        exit(EXIT_FAILURE);
    }


    //全连接层运算，将矩阵展平并转化成结果向量
    full_connect(conv_mat3, fc_params[0], vec, vec_size);

    //将结果向量转化成[0.0, 1.0]的概率区间范围
    soft_max(vec, vec_size);
    gettimeofday(&total_t2, NULL);
    double total_time_use = (total_t2.tv_sec - total_t1.tv_sec) * 1000 +
(double)(total_t2.tv_usec - total_t1.tv_usec) / 1000.0;

    //打印最终运算结果
    printf("Background possibility: %f\nFace possibility: %f\n", vec[0],
vec[1]);

    cout << "The total relu time used is " << relu_time_use << " us\n";

    cout << "The total time used in processing CNN using simple method is: " <<
total_time_use << "ms" << endl;
```

朴素卷积实现

```
//朴素卷积运算
template <typename T>
void convolution(Matrix<T>& mat, const conv_param& conv, Matrix<T>& res_mat){
    const int mat_channel = mat.getChannel();
    const int res_mat_channel = res_mat.getChannel();
    //参数检查
    if(res_mat_channel != conv.out_channels){
```

```
        cerr << "Error in file " << __FILE__ << " in function " << __FUNCTION__
<< " in line " << __LINE__ << endl;
        cerr << "The size of output matrix is not equal to the output channel
number " << endl;
        exit(EXIT_FAILURE);
    }

    if(mat_channel != conv.in_channels){
        cerr << "Error in file " << __FILE__ << " in function " << __FUNCTION__
<< " in line " << __LINE__ << endl;
        cerr << "Mismatch of input channel number" << endl;
        exit(EXIT_FAILURE);
    }

    int in_channel = conv.in_channels;
    int out_channel = conv.out_channels;
    const int mat_r = mat.getRow();
    const int mat_c = mat.getCol();
    int ker_r = conv.kernel_size;
    int ker_c = conv.kernel_size;
    int padding = conv.pad;
    int stride = conv.stride;

    int res_height = res_mat.getRow();
    int res_width = res_mat.getCol();

    for(int out_ch = 0; out_ch < out_channel; out_ch++){//输出通道数量

        for(int in_ch = 0; in_ch < in_channel; in_ch++){//传入通道数量

            float ans = 0;
            // int conv_move = 0;

            for(int img_height = 0, res_h = 0; img_height + ker_r <= mat_r + 2 *
padding; img_height += stride, res_h++){//图像高度
                for(int img_width = 0, res_w = 0; img_width + ker_c <= mat_c + 2
* padding; img_width += stride, res_w++){//图像宽度

                    for(int ker_height = 0; ker_height < ker_r; ker_height ++)
{//核高度
                        for(int ker_width = 0; ker_width < ker_c; ker_width ++)
{//核宽度

                            if(img_height + ker_height >= padding && img_width +
ker_width >= padding //在目标范围内
                                && img_height + ker_height < padding + mat_r &&
img_width + ker_width < padding + mat_c){

                                ans += mat(img_height + ker_height - padding,
img_width + ker_width - padding, in_ch + 1)
                                    * conv.p_weight[out_ch * (in_channel * 3 *
3) + in_ch * (3 * 3) + ker_height*ker_r + ker_width];

                            }

                        }
                    }
                    res_mat(res_h, res_w, out_ch + 1) += ans;
```

```
                    ans = 0;
                }
            }

        }
    }
//给每个通道加上对应的偏置值
    for(int ch = 1; ch <= out_channel; ch++){
        for(int h = 0; h < res_height; h++){
            for(int w = 0; w < res_width; w++){
                res_mat(h, w, ch) += conv.p_bias[ch - 1];
            }
        }
    }

}
```

矩阵乘法优化卷积层实现

```
//矩阵乘法卷积运算
template <typename T>
void im2col_convolution(Matrix<T>& mat, const conv_param& conv, Matrix<T>&
res_mat){
    const int mat_channel = mat.getChannel();
    const int res_mat_channel = res_mat.getChannel();
    //参数检查

    if(res_mat_channel != conv.out_channels){
        cerr << "Error in file " << __FILE__ << " in function " << __FUNCTION__
<< " in line " << __LINE__ << endl;
        cerr << "The size of output matrix is not equal to the output channel
number " << endl;
        exit(EXIT_FAILURE);
    }

    if(mat_channel != conv.in_channels){
        cerr << "Error in file " << __FILE__ << " in function " << __FUNCTION__
<< " in line " << __LINE__ << endl;
        cerr << "Mismatch of input channel number" << endl;
        exit(EXIT_FAILURE);
    }
    if(mat.nums == nullptr){
        cerr << "Error in file " << __FILE__ << " in function " << __FUNCTION__
<< " in line " << __LINE__ << endl;
        cerr << "The input matrix's array is nullptr." << endl;
        exit(EXIT_FAILURE);
    }
    if(res_mat.nums == nullptr){
        cerr << "Error in file " << __FILE__ << " in function " << __FUNCTION__
<< " in line " << __LINE__ << endl;
        cerr << "the element array of the result matrix is nullptr" << endl;
        exit(EXIT_FAILURE);
    }

    int in_channel = conv.in_channels;
```

```
    int out_channel = conv.out_channels;
    const int mat_r = mat.getRow();
    const int mat_c = mat.getCol();
    int ker_size = conv.kernel_size * conv.kernel_size;
    int ker_r = conv.kernel_size;
    int ker_c = conv.kernel_size;
    int padding = conv.pad;
    int stride = conv.stride;


    int res_height = res_mat.getRow();
    int res_width = res_mat.getCol();

    Matrix<float> tmp_mat(ker_c * ker_r * in_channel, res_height * res_width,
1);
    Matrix<float> ker_mat(out_channel, in_channel * ker_size, 1);

    int mat_nums_size = mat.getCol() * mat.getRow() * mat.getChannel();
    int tmp_mat_size = tmp_mat.getCol() * tmp_mat.getRow() *
tmp_mat.getChannel();
    int ker_mat_size = ker_mat.getCol() * ker_mat.getRow() *
ker_mat.getChannel();

    //在实际访问之前先判断是否可以在即将进行的循环中保证访问不会越界导致程序崩溃
    if(((in_channel - 1) * ker_size + (ker_r - 1) * ker_c + ker_c - 1) *
tmp_mat.getSpan() + res_height * res_width * tmp_mat.getChannel() >
tmp_mat_size){
        cerr << "Error in file " << __FILE__ << " in function " << __FUNCTION__
<< " in line " << __LINE__ << endl;
        cerr << "The size of array of the temporary matrix is not match of the
source matrix." << endl;
        exit(EXIT_FAILURE);
    }

    for(int in_ch = 0; in_ch < in_channel; in_ch ++){//传入通道数量

        int span = in_ch * ker_size;//将最内层需要频繁计算的值先提前计算，提升性能

        int col_cnt = -1;
        for(int img_height = 0; img_height + ker_r <= mat_r + 2 * padding;
img_height += stride){//图像高度
            for(int img_width = 0; img_width + ker_c <= mat_c + 2 * padding;
img_width += stride){//图像宽度
                col_cnt ++;
                for(int ker_height = 0; ker_height < ker_r; ker_height ++){//核高
度
                    for(int ker_width = 0; ker_width < ker_c; ker_width ++){//核
宽度

                        if(img_height + ker_height >= padding && img_width +
ker_width >= padding //判断计算点是否位于目标范围内
                            && img_height + ker_height < padding + mat_r &&
img_width + ker_width < padding + mat_c){

                            tmp_mat.nums[(span + ker_height * ker_c + ker_width)
* tmp_mat.span + col_cnt * tmp_mat.channel] =
```

```
                                mat.nums[(img_height + ker_height - padding) *
mat.span + (img_width + ker_width - padding) * mat.channel + in_ch];//如果在范围
内，就将对应位置赋值为输入矩阵对应的元素

                        }else{

                            tmp_mat.nums[(span + ker_height * ker_c + ker_width)
* tmp_mat.span + col_cnt * tmp_mat.channel] = 0;//不在范围内就将其赋值为0

                        }

                    }
                }


            }
        }
    }

    //在实际访问之前先判断是否可以在即将进行的循环中保证访问不会越界导致程序崩溃

    if((out_channel - 1) * ker_mat.getSpan() + ((in_channel - 1) * ker_size +
(ker_r - 1) *  ker_c + ker_c  - 1) * ker_mat.getChannel() > ker_mat_size){
        cerr << "Error in file " << __FILE__ << " in function " << __FUNCTION__
<< " in line " << __LINE__ << endl;
        cerr << "The size of kernel transformation matrix is not equal to the
kernel's size" << endl;
        exit(EXIT_FAILURE);
    }

    //将核转化成行矩阵ker_mat
    for(int o = 0; o < out_channel; o++){
        for(int i = 0; i < in_channel; i++){
            int span = i * ker_size;//将内层变量提前进行计算，提升性能
            int move = o*(in_channel*3*3) + i*(3*3);
            for(int ker_height = 0; ker_height < ker_r; ker_height++){
                for(int ker_width = 0; ker_width < ker_c; ker_width++){

                    ker_mat.nums[o * ker_mat.span + (span + ker_height * ker_c +
ker_width) * ker_mat.channel] =
                    conv.p_weight[move + ker_height*ker_r + ker_width];
                }
            }
        }
    }

    Matrix<float> ans_mat = ker_mat * tmp_mat;

    for(int o = 0; o < out_channel; o++){
        for(int h = 0; h < res_height; h++){
            for(int c = 0; c < res_width; c++){

                res_mat.nums[h * res_mat.span + c * res_mat.channel + o] =
ans_mat.nums[o * ans_mat.span  +  (h*res_width + c) * ans_mat.channel] +
conv.p_bias[o];//将结果矩阵转化为输出矩阵并加上偏置
            }
        }
    }
```

```
}
```

Relu激活函数

```
//Relu激活函数
template<typename T>
void Relu(Matrix<T>& in_mat){

    const int height = in_mat.getRow();
    const int width = in_mat.getCol();
    const int channel = in_mat.getChannel();
    for(int h = 0; h < height; h++){
        for(int w = 0; w < width; w++){
            for(int ch = 1; ch <= channel; ch++){

                //max实现
                // in_mat.nums[h * in_mat.span + w * in_mat.channel + ch - 1] =
max(in_mat.nums[h * in_mat.span + w * in_mat.channel + ch - 1], 0.0f);

                //三目运算符实现
                in_mat.nums[h * in_mat.span + w * in_mat.channel + ch - 1] =
(in_mat.nums[h * in_mat.span + w * in_mat.channel + ch - 1] < 0)
                 ? 0 : in_mat.nums[h * in_mat.span + w * in_mat.channel + ch -
1];

                //if 实现
                // if(in_mat.nums[h * in_mat.span + w * in_mat.channel + ch - 1]
< 0){
                //     in_mat.nums[h * in_mat.span + w * in_mat.channel + ch - 1]
= 0;
                // }

                //乘法实现
                // in_mat.nums[h * in_mat.span + w * in_mat.channel + ch - 1] *=
(in_mat.nums[h * in_mat.span + w * in_mat.channel + ch - 1] > 0.f);

            }
        }
    }
}
```

池化函数实现

```
//最大池化函数
template<typename T>
void max_pooling(Matrix<T>& in_mat, Matrix<T>& res_mat){

    const int src_height = in_mat.getRow();
    const int src_width = in_mat.getCol();
    const int src_channel = in_mat.getChannel();
    const int out_height = res_mat.getRow();
```

```
    const int out_width = res_mat.getCol();
    const int out_channel = res_mat.getChannel();

    //参数检查
    if((src_height % 2 == 0 && src_height != out_height * 2) || (src_width % 2
== 1 && src_height != out_height * 2 - 1)){
        cerr << "Error in file " << __FILE__ << " in function " << __FUNCTION__
<< " in line " << __LINE__ << endl;
        cerr << "The height of output matrix: " << out_height << "should be half
of the height of the input matrix: " << src_height << endl;
        exit(EXIT_FAILURE);
    }
    if((src_width % 2 == 0 && src_width != out_height * 2) || (src_width % 2 ==
1 && src_width != out_width * 2 - 1)){
        cerr << "Error in file " << __FILE__ << " in function " << __FUNCTION__
<< " in line " << __LINE__ << endl;
        cerr << "The width of output matrix: " << out_width << "should be half
of the width of the input matrix: " << src_width << endl;
        exit(EXIT_FAILURE);
    }
    if(src_channel != out_channel){
        cerr << "Error in file " << __FILE__ << " in function " << __FUNCTION__
<< " in line " << __LINE__ << endl;
        cerr << "The channel of output matrix:" << out_channel << " should be
the same as the output matrix: " << src_channel << endl;
        exit(EXIT_FAILURE);
    }

    for(int ch = 1; ch <= src_channel; ch++){
        for(int src_h = 0; src_h < src_height; src_h += 2){
            for(int src_c = 0; src_c < src_width; src_c += 2){

                float maximum = in_mat.nums[src_h * in_mat.span + src_c *
in_mat.channel + ch - 1];
                for(int i = 0; i < 2; i++){
                    for(int j = 0; j < 2; j ++){
                        int h = src_h + i;
                        int c = src_c + j;
                        if(h < src_height && c < src_width && in_mat.nums[h *
in_mat.span + c * in_mat.channel +ch - 1] > maximum){
                            maximum =in_mat.nums[h * in_mat.span + c *
in_mat.channel +ch - 1];
                        }
                    }
                }
                res_mat.nums[((src_h + 1)/ 2) * res_mat.span + ((src_c + 1) / 2)
* res_mat.channel + ch -1] = maximum;
            }
        }
    }

}
```

全连接层实现

```
//全连接函数
template<typename T>
void full_connect(Matrix<T>& mat, fc_param& fc_params, float* vec, int size){
    const int channel = mat.getChannel();
    const int height = mat.getRow();
    const int width = mat.getCol();
    int out_feature = fc_params.out_features;
    int in_feature = fc_params.in_features;
    //参数检查
    if(vec == nullptr){
        cerr << "Error in file " << __FILE__ << " in function " << __FUNCTION__
<< " in line " << __LINE__ << endl;
        cerr << "The input of the result vector is nullptr" << endl;
        exit(EXIT_FAILURE);
    }

    if(channel * height * width != in_feature){
        cerr << "Error in file " << __FILE__ << " in function " << __FUNCTION__
<< " in line " << __LINE__ << endl;
        cerr << "The product of the channel and height and width of output
matrix:" << channel * height * width << " should be the same as the in_feature "
<< in_feature << endl;
        exit(EXIT_FAILURE);
    }
    if(size != out_feature){
        cerr << "Error in file " << __FILE__ << " in function " << __FUNCTION__
<< " in line " << __LINE__ << endl;
        cerr << "The size of output vector" << size << " should be the same as
the out_feature " << out_feature << endl;
        exit(EXIT_FAILURE);
    }
    int cnt = 0;
    for(int o = 0; o < out_feature; o++){
        float ans = 0;

        for(int ch = 1; ch <= channel; ch++){
            for(int h = 0; h < height; h++){
                for(int c = 0; c < width; c++){
                   ans += mat.nums[h * mat.span + c * mat.channel +  ch - 1]
                   * fc0_weight[cnt++];
                }
            }
        }

        vec[o] = ans + fc0_bias[o];
    }

}
```

Softmax 归一化实现

```
void soft_max(float* vec, int size){
    //参数检查
    if(vec == nullptr){
        cerr << "Error in file " << __FILE__ << " in function " << __FUNCTION__
<< " in line " << __LINE__ << endl;
        cerr << "The input of the result vector is nullptr" << endl;
        exit(EXIT_FAILURE);
    }
    if(size <= 0){
        cerr << "Error in file " << __FILE__ << " in function " << __FUNCTION__
<< " in line " << __LINE__ << endl;
        cerr << "The input of vector size should be larger than 0" << endl;
        exit(EXIT_FAILURE);
    }

    float exp_sum = 0.0;

    for(int i = 0; i < size; i++){
        exp_sum += exp(vec[i]);
    }

    for(int i = 0; i < size; i++){
        vec[i] = exp(vec[i]) / exp_sum;
    }

}
```

Matrix.hpp 中对于矩阵乘法的优化实现

```
//乘法运算符重载
template <typename T>
Matrix<T> Matrix<T>::operator*(T num)const{
    Matrix mat;
    mat.row = row;
    mat.col = col;
    mat.channel = channel;
    mat.ROIBeginRow = ROIBeginRow;
    mat.ROIBeginCol = ROIBeginCol;
    mat.ROIRow = ROIRow;
    mat.ROICol = ROICol;
    mat.span = span;
    mat.nums = new T[row * col * channel];
    memset(mat.nums, 0, sizeof(T) * row * col * channel);

    for(int i = 0; i < row; i++){
        for(int j = 0; j < col; j++){
            for(int ch = 1; ch <= channel; ch++){
                mat.nums[i * mat.span + j * mat.channel + ch - 1] = this ->
nums[i * this -> span + j * this -> channel + ch -1] * num;
            }
        }
    }
    num_matrices++;
    return mat;
```

```
}
template<typename T>
Matrix<T>& Matrix<T>::operator*=(T num){
    int ele_num = row * col * channel;
    if(this -> ref_count[0] == 1){
        for(int i = 0; i < row; i++){
            for(int j = 0; j < col; j++){
                for(int ch = 1; ch <= channel; ch++){
                    this -> nums[i * this -> span + j * this -> channel + ch -
1]
                    = this -> nums[i * this -> span + j * this -> channel + ch -
1]
                    * num;
                }
            }
        }
    }else{
        this -> ref_count[0]--;
        this -> ref_count = new int[1];
        this -> ref_count[0] = 1;
        T* n = new T[ele_num];
        memset(n, 0, sizeof(T) * ele_num);
        int pos = 0;
        for(int i = 0; i < row; i++){
            for(int j = 0; j < col; j++){
                for(int ch = 1; ch <= channel; ch++){
                    n[pos++] = this -> nums[i * this -> span + j * this ->
channel + ch - 1]
                    * num;
                }
            }
        }
        this -> nums = n;
    }

    return *this;
}

//矩阵乘法
template<typename T>
Matrix<T> Matrix<T>::operator*(Matrix<T> &mat){
   #pragma omp sections
    //判断参数合法性
    if(mat.row != this -> col || mat.channel != channel){
        std::cerr << "In operator * friend functoin..." << std::endl;
        std::cerr << "The two matrices' sizes and channel number should be the
same." << std::endl;
        exit(EXIT_FAILURE);
    }
    //检查传入的数组是否为空
    if(mat.nums == nullptr){
        std::cerr << "In operator * friend functoin..." << std::endl;
        std::cerr << "The input element array of mat is nullptr." << std::endl;
        exit(EXIT_FAILURE);
    }

    Matrix<T> res;
    num_matrices++;
```

```
    res.row = row;
    res.col = mat.col;
    res.channel = channel;
    res.ROIBeginRow = 0;
    res.ROIBeginCol = 0;
    res.ROIRow = 0;
    res.ROICol = 0;
    res.span = res.channel * res.col;
    int ele_num = res.row * res.col * res.channel;
    res.nums = new T[ele_num];
    memset(res.nums, 0, sizeof(T) * ele_num);


    // #pragma omp parallel sections num_threads(4)

    float* nums = new float[mat.getCol() * mat.getRow() * mat.getChannel()];
    memset(nums, 0, sizeof(float) * (mat.getCol() * mat.getRow() *
mat.getChannel()));
    float* in_nums = new float[row * col * channel];
    memset(in_nums, 0, sizeof(float) * (row * col * channel));
    float* res_nums = new float[res.row * res.col * res.channel];
    memset(res_nums, 0, sizeof(float) * (res.row * res.col * res.channel));

    if(nums == nullptr || in_nums == nullptr || res_nums == nullptr){
        std::cerr << "In fuction " << __FUNCTION__ << " in line " << __LINE__ <<
std::endl;
        std::cerr << "Failure in memory allocating.\n";
    }

    int mat_row = mat.row;
    int mat_col = mat.col;
    int mat_channel = mat.channel;
    int cnt = 0;
// #pragma omp sections
// #pragma omp section
    for(int ch = 1; ch <= mat_channel; ch++){
        for(int r = 0; r < mat_row; r++){
            for(int c = 0; c < mat_col; c++){
                nums[cnt++] = mat.nums[r * span + c * channel + ch - 1];
            }
        }
    }
// #pragma omp section
    int cnt2 = 0;
    for(int ch = 1; ch <= channel; ch++){
        for(int r = 0; r < row; r++){
            for(int c = 0; c < col; c++){
                in_nums[cnt2++] = this -> nums[r * span + c * channel + ch - 1];
            }
        }
    }
    int cnt3 = 0;

// #pragma omp sections
// #pragma omp section
//使用OpenBLAS进行矩阵乘法运算
    cblas_sgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, this -> row, mat.col,
this -> col, 1.0f,
```

```
        this -> nums, this -> col, mat.nums, mat.col, 0, res_nums, mat.col);


    for(int r = 0; r < row; r++){
        for(int c = 0; c < mat.col; c++){
            for(int ch = 1; ch <= channel; ch++){
                res(r, c, ch) = res_nums[cnt3++];
            }
        }
    }

    return res;
}
```

Part3 - Result and Verification

在循环内部调用获取像素方法以及将矩阵数组开放程度提高效率的对比与分析

由于在进行卷积运算的时候每次获取像素都需要通过调用`matrix`类的获取指定像素点的方法实现，而每次调用该方法都会检查传入的参数是否合法，在进行密集计算的时候，这样的函数调用无疑会耗费大量冗余的时间开销，所以考虑将矩阵数组的访问权限设置为公开后进行试验，经过多次试验得到的实验表格如下所示：

				平均值	
开放程度提升矩阵乘法卷积优化/ms	22.337	22.487	22.491	22.43833333	
O3优化矩阵乘法优化卷积时间/ms	26.219	24.638	25.401	25.41933333	

可以通过前文所述计算时间偏差公式计算得到两种方法实现卷积的平均表现，可以得到对矩阵进行数组访问权限的开放会比不开放而通过函数调用访问像素的方式速度平均提升了10%

对于矩阵数组开放访问权限，允许外部直接进行访问后，可以将卷积层，激活层以及池化层对矩阵内部数组的访问直接通过数组访问符实现，在本程序当中，由于在实际访问数组元素之前已经预先进行对循环当中对矩阵数组最大访问界限的判断，如下程序段所示：

```
    int tmp_mat_size = tmp_mat.getCol() * tmp_mat.getRow() *
tmp_mat.getChannel();
    //在实际访问之前先判断是否可以在即将进行的循环中保证访问不会越界导致程序崩溃
    if(((in_channel - 1) * ker_size + (ker_r - 1) * ker_c + ker_c - 1) *
       tmp_mat.getSpan() + res_height * res_width * tmp_mat.getChannel() >
tmp_mat_size){
            cerr << "Error in file " << __FILE__ <<
                " in function " << __FUNCTION__ << " in line " << __LINE__ <<
endl;
            cerr << "The size of array of the temporary matrix
                is not match of the source matrix." << endl;
            exit(EXIT_FAILURE);
        }
```

确定其不会超出矩阵数组的大小之后才会进行实际的运算，这样，可以保证在本程序当中在提升速度性能的同时保证了程序运行的安全性。

将矩阵类成员的类型从`int`更改为`size_t`的对比分析

由于矩阵类内部的行数以及列数，通道数在实际运用过程中可能较大，加上实际中矩阵的行列数以及通道数均为正数，而`int`类型是4位字节的有符号整数，这样就会导致`int`可供存储的范围只有一半适用，造成浪费而`size_t`在本机下是8个字节的无符号整数，可以存储更多的信息，所以考虑使用`size_t`进行相关信息的存储。

经过实际试验，对比使用`int`对相关信息进行存储的卷积时间，对比如下表所示

					平均值	
	O3优化下size_t矩阵乘法卷积优化时间/ms	13.29	12.829	13.944	13.35433333	
	O3优化下int矩阵乘法卷积时间/ms	26.219	24.638	25.401	25.41933333	

使用`size_t`进行运算会使得矩阵乘法的速度比使用`int`快了47%，经过对整体程序的分析，发现对于成员变量的类型的改变，只是改变了矩阵乘法中数组下标的类型，通过查阅《C++ Primer》相关章节，发现数组下标的确是`size_t`类型，而起始使用`size_t`进行存储可以在矩阵乘法内部进行运算的时候避免了`int`类型向`size_t`的频繁转换，有效提升了矩阵乘法的效率，进而使得卷积运算提速。

使用OpenBLAS对矩阵运算进行加速的效率对比与分析

通过project3，4当中对矩阵乘法速度的优化策略的总结，发现对矩阵乘法实现的卷积神经网络算法可以使用前面`project`的使用`OpenBLAS`进行优化。

对于多通道的矩阵而言，由于在上一次的`project`当中对多通道矩阵元素的存储方式是将同一元素的所有通道都相邻地存储在一起，所以在进行`OpenBLAS`操作之前需要先对矩阵的对应元素做相应转换才能够使用`OpenBLAS`当中的`cblas_sgemm`方法进行矩阵乘法的加速。

使用OpenBLAS对矩阵进行加速之后得到的图表如下所示：

				平均值
O3优化矩阵乘法优化卷积时间/ms	26.219	24.638	25.401	25.41933333
OpenBLAS矩阵乘法优化/ms	6.534	6.488	6.385	6.469

经过对平均消耗时间的计算得到，使用OpenBLAS进行矩阵乘法优化可以将卷积的总体速度提升74%

通过在网上查阅与OpenBLAS计算矩阵乘法原理的资料，了解到该库的矩阵乘法利用了下列优化方式：

1. 矩阵是以列优先存储的，这样就可以再访问内存时尽可能顺序访问。
2. 其次是有分块的思想，将一些向量提取出来，并在之后的计算当中多次对其复用，节省高速缓存的使用，进一步地，可以提取出一个矩阵块进一步优化。
3. 第三是使用寄存器变量，而不是直接操作内存，进一步减轻缓存的压力。
4. 再者是访存的优化，除了合理使用指针的移动来进行访存的优化，还可以将矩阵复制一份到连续的空间，这样就避免了在不连续的空间当中进行低效的访存，这样的意义就在于存储是连续的存储，读取也是连续的读取，通过结合在本次程序中的访存优化步骤，可以看出这样使得数据位于连续位置的优化是非常显著的。

5. 最后是利用单指令多数据（SIMD）的向量化指令，优化CPU的性能。

使用并行计算（openmp）对加速卷积运算的对比与分析

通过使用openmp可以实现对矩阵运算的多核并行操作，由于在使用OpenBLAS之前需要对矩阵的数组预先进行转换，将同一通道的元素存储在一起进行之后的运算，这里涉及到三次的矩阵数组转换，如果三次串行执行，那么效率较低，这时候可以使用`openmp`进行并行的运算将三次的转换同时执行。

代码框架如下所示：

```
#pragma omp parallel sections num_threads(4)//放在矩阵乘法执行开始位置，表示将要开启4个线程
#pragma omp sections
#pragma omp section
   for(){}//对矩阵乘法左元进行数组的转换
#pragma omp section
   for(){}//对矩阵乘法右元进行数组的转换

#pragma omp sections//开启单独的sectios
#pragma omp section
//进行OpenBLAS运算
    cblas_sgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, this -> row, mat.col, this -> col, 1.0f,
        this -> nums, this -> col, mat.nums, mat.col, 0, res_nums, mat.col);

//将结果矩阵进行对应数组元素的转换
    for(){}
    return res;
```

经过时间的测速比较，如下表所示:

				平均值
OpenBLAS矩阵优化/ms	6.534	6.488	6.385	6.469
使用并行处理/ms	6.432	7.116	6.806	6.784666667

通过将表中的数据进行比对可以发现，并行加速并没能像预期一般对矩阵的转换进行加速，反而使得程序变慢了，分析原因，第一种可能是对多线程的运作还不太熟悉，导致不能理想地利用线程优化代码，反倒由于额外分配了4个线程而拖慢了程序。第二种可能是由于矩阵计算的数量太小，而时间的消耗又不是非常明显，所以多线程的优势就无法更好地体现出来，在并行处理速度上的优势反倒比不上额外分配的线程所消耗的时间消耗。

对Relu层的优化

由于Relu层的运算逻辑比较简单，依据在矩阵对应位置的元素是否大于零来决定该位置的取值是其本身还是零，而这一步的逻辑的朴素实现是通过`max(value, 0)`来进行实现的，进一步考虑使用三目运算符

```
val = (va > 0)? val :0;
```

与使用乘法表达式

```
val *= (val > 0);
```

进行相关实现

由于Relu层对整体程序运行的时间影响并不大，所以如果要实现对Relu层的精确计时，需要通过分别对三层Relu进行计时之后累加得到的结果进行对比。

通过科学的计时，得到Relu层的四种实现分别对应的时间表格如下所示

				平均值
Relu max实现/us	165	167	174	168.6666667
Relu 三目实现/us	22	23	23	22.66666667
Relu if 实现/us	258	262	272	264
Relu 乘法实现/us	223	248	241	237.3333333

可以看出在使用三目运算符时，进行Relu激活层的运算用时明显减少，平均下来速度较朴素Relu算法提升了86%

经过分析其中的原理，可能是在Relu的循环当中三目运算符对应的汇编代码耗时更短的缘故。

不同方式实现卷积的效率对比以及分析

X86和ARM架构下卷积运算效率对比分析

注意到在ARM服务器的char 类型与X86的char类型不一样，所以在进行相关处理之后再进行矩阵的运算，

通过连接到ARM服务器可以进行程序的运行以及比较具体运行过程如下所示：

```
[root@ecs-101-0005 project5]# make
[100%] Built target cnn
[root@ecs-101-0005 project5]# ./cnn
Background possibility: 0.000000
Face possibility: 1.000000
The total relu time used is 34.000000 us
The total time used in processing CNN using simple method is: 12.873000ms
[root@ecs-101-0005 project5]# ./cnn
Background possibility: 0.000000
Face possibility: 1.000000
The total relu time used is 35.000000 us
The total time used in processing CNN using simple method is: 12.893000ms
[root@ecs-101-0005 project5]# ./cnn
Background possibility: 0.000000
Face possibility: 1.000000
The total relu time used is 35.000000 us
The total time used in processing CNN using simple method is: 12.860000ms
[root@ecs-101-0005 project5]# ls
CMakeCache.txt  CMakeLists.txt  Matrix.hpp  cmake_install.cmake  cnn.cpp   face2.jpg
CMakeFiles      Makefile        bg.jpg      cnn                  face.jpg  face_binary_cls.cpp
```

运行的图片依然是样例图片，将得到的结果与在X86下的运行架构下的运行结果作对比，制作成表格如下所示

				平均值
X86运行时间/ms	14.243	14.421	13.376	14.01333333
ARM运行时间/ms	12.873	12.893	12.86	12.87533333

平均下来，运行没有加上 `OpenBLAS` 的矩阵乘法优化后的卷积运算，从平均时间来看，在ARM平台上的运行时间比在本地X86上运行快1ms左右，换算成比例就是快了8%左右。

分析其中的原因，结合此次的代码更多的是在循环内部进行计算，而矩阵的规模相对于 `project4` 的矩阵规模更小，频繁涉及对内存的访问，综合考虑是由于ARM的对于频繁的内存访问效率比X86更高，但是由于实验结果相差不多，于是考虑使用朴素卷积实现再次对二者进行对比，运行结果如下表所示：

朴素实现				**平均值**
X86运行时间/ms	39.918	43.751	42.715	42.128
ARM运行时间/ms	47.783	44.194	44.119	45.36533333

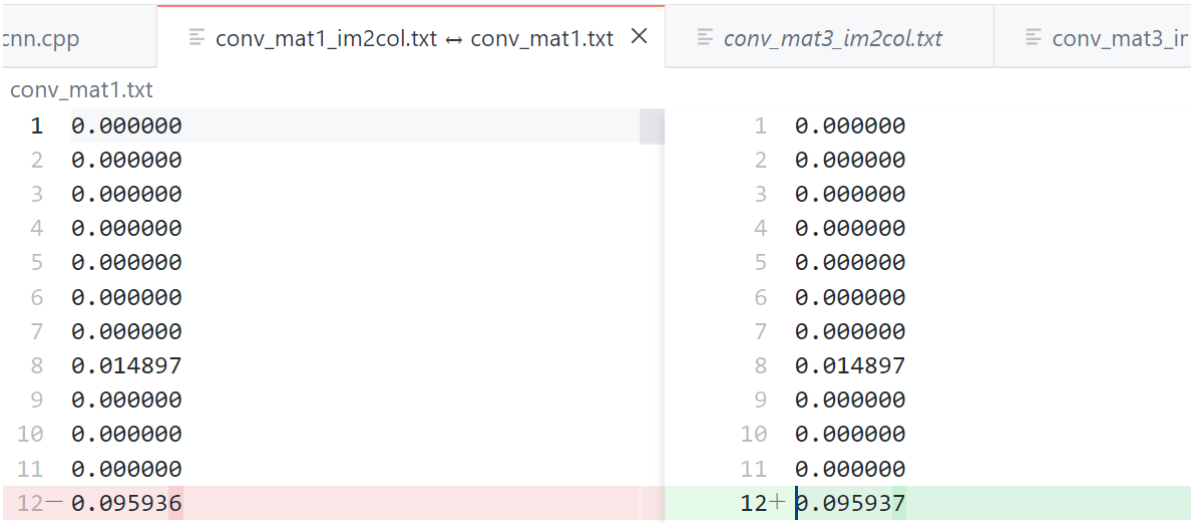
可以发现，在朴素的卷积实现当中ARM服务器反而比不上X86的速度了，然而朴素卷积运算和矩阵乘法不使用 `OpenBLAS` 的区别就在于大量数据的拷贝，即内存的访问，朴素卷积并没有涉及像矩阵乘法优化那么多的内存访问与数据拷贝，所以这也可以从一方面看出，ARM架构下对于内存的访问可能会比X86架构下的访问更优。

结果正确性检验

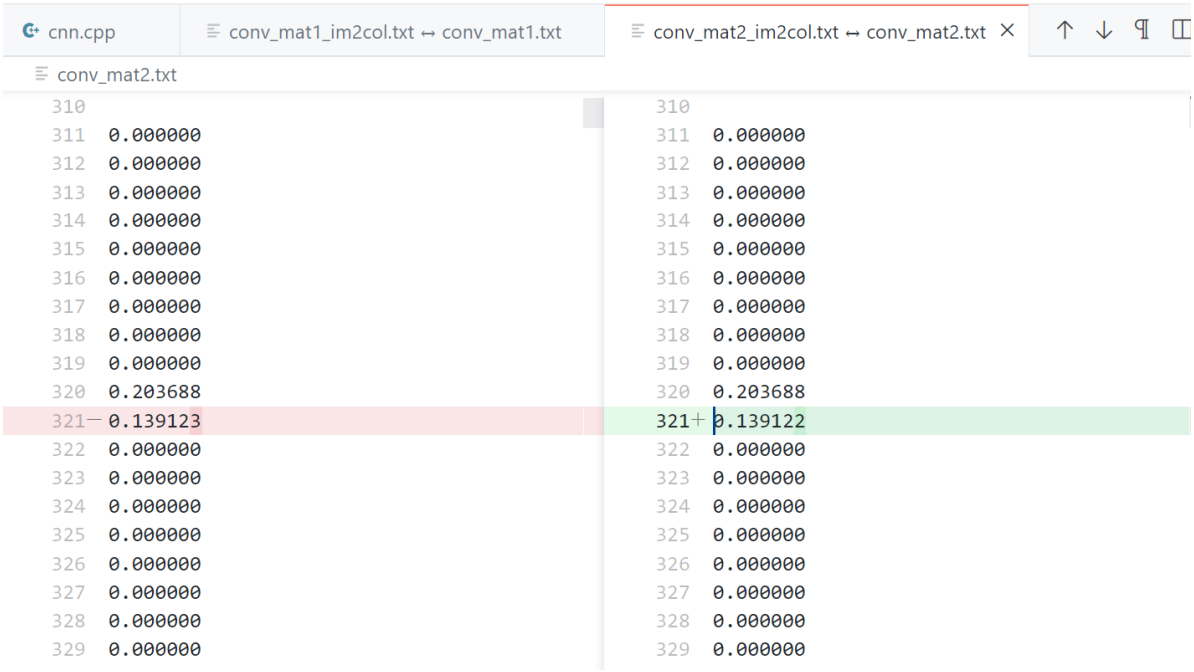
卷积朴素实现与矩阵乘法卷积实现的各层卷积对比结果

通过在上周理论课上学到的将输出结果导出到文件当中的方法，将每一层卷积运算之后的结果都输出到对应的文件当中，再使用 `vsCode` 的文件比对功能，进行三层的比对结果如下所示（由于数据过多，仅展示部分，具体每一层的实验数据也会上传到作业提交系统）：

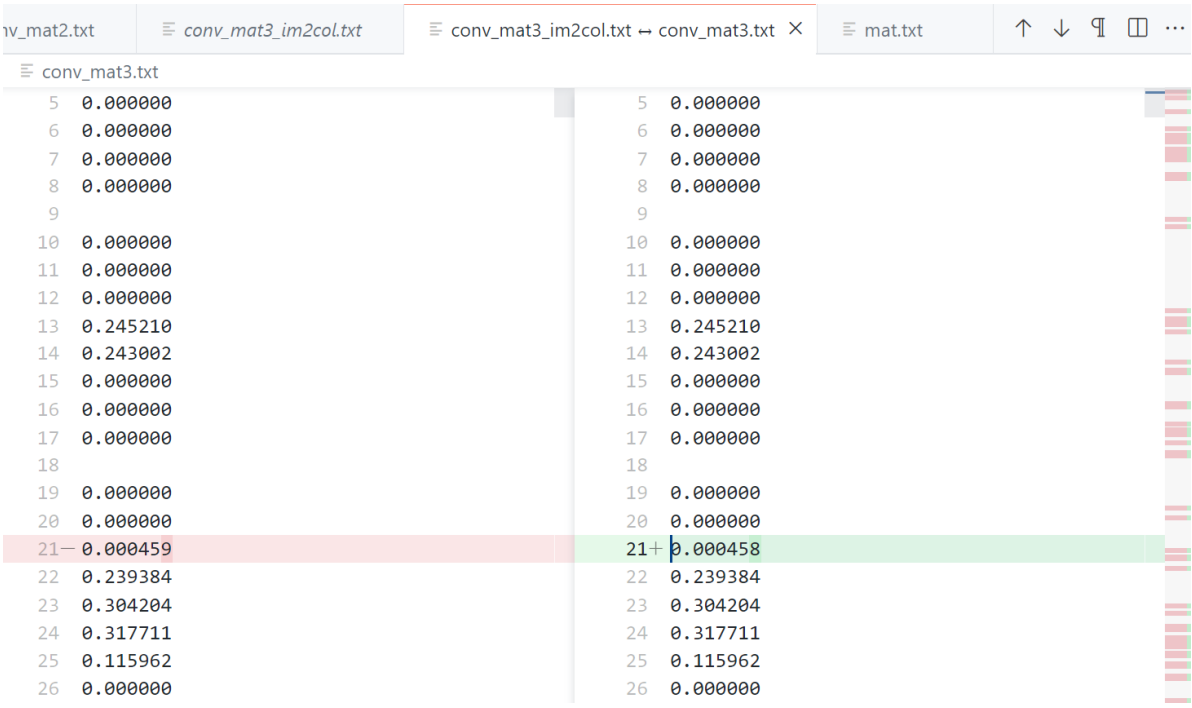
第一层：



第二层：



第三层：



最终卷积结果：

```
lmq@LAPTOP-4MG6A2H2:/mnt/c/大二课程/c++/C++Code/project5$ g++ cnn.cpp  `pkg-config --cflags --libs opencv4` && ./a.out
Background possibility: 0.000000
Face possibility: 1.000000
The total time used in processing CNN using simple method is: 43.102000ms
```

```
lmq@LAPTOP-4MG6A2H2:/mnt/c/大二课程/c++/C++Code/project5$ g++ cnn.cpp  `pkg-config --cflags --libs opencv4` && ./a.out
Background possibility: 0.000000
Face possibility: 1.000000
The total time used in processing CNN using matrix multiplication is: 26.211000ms
```

可以看出，大部分的数据都完全一致，只是少部分数据由于浮点数的精度计算原因在小数点的末位有所偏差，但这并不足以影响最终的CNN对样例图片 `face.jpg` 的识别结果，在允许的误差范围内可以认为朴素实现的卷积与使用了矩阵乘法优化后的卷积的两种实现是完全一致的，正确性是可以保证的。

两种实现对不同图片的识别正确性检验

以上实验为了控制变量，均使用样例图片作为CNN运行速度的测试，这一小节针对不同人脸以及背景的图片进行识别正确性的检验。

首先收集了另外的四张人脸图片以及四张背景图片，将其放入CNN网络当中进行运算，运算的结果如下所示

- 人脸

```
lmq@LAPTOP-4MG6A2H2:/mnt/c/大二课程/c++/C++Code/project5$ g++ cnn.cpp  `pkg-co
& ./a.out
Background possibility: 0.000000
Face possibility: 1.000000
The total relu time used is 26.000000 us
The total time used in processing CNN using simple method is: 16.706000ms
```

```
lmq@LAPTOP-4MG6A2H2:/mnt/c/大二课程/c++/C++Code/project5$ g++ cnn.cpp  `pkg-con
& ./a.out
Background possibility: 0.011302
Face possibility: 0.988698
The total relu time used is 27.000000 us
The total time used in processing CNN using simple method is: 16.137000ms
```

```
lmq@LAPTOP-4MG6A2H2:/mnt/c/大二课程/c++/C++Code/project5$ g++ cnn.cpp  `pkg
& ./a.out
Background possibility: 0.056109
Face possibility: 0.943891
The total relu time used is 38.000000 us
The total time used in processing CNN using simple method is: 16.705000ms
```

```
lmq@LAPTOP-4MG6A2H2:/mnt/c/大二课程/c++/C++Code/project5$ g++ cnn.cpp  `pkg
& ./a.out
Background possibility: 0.056109
Face possibility: 0.943891
The total relu time used is 28.000000 us
The total time used in processing CNN using simple method is: 16.452000ms
```

- 背景

```
lmq@LAPTOP-4MG6A2H2:/mnt/c/大二课程/c++/C++Code/project5$ g++ cnn.cpp  `pkg-c
& ./a.out
Background possibility: 1.000000
Face possibility: 0.000000
The total relu time used is 31.000000 us
The total time used in processing CNN using simple method is: 16.778000ms
```

```
lmq@LAPTOP-4MG6A2H2:/mnt/c/大二课程/c++/C++Code/project5$ g++ cnn.cpp  `pkg-config --cf
& ./a.out
Background possibility: 0.993275
Face possibility: 0.006725
The total relu time used is 30.000000 us
The total time used in processing CNN using simple method is: 17.779000ms
```

```
lmq@LAPTOP-4MG6A2H2:/mnt/c/大二课程/c++/C++Code/project5$ g++ cnn.cpp  `pkg-config --cflags --libs opencv4` &
& ./a.out
Background possibility: 0.999508
Face possibility: 0.000492
The total relu time used is 31.000000 us
The total time used in processing CNN using simple method is: 15.407000ms
```

```
lmq@LAPTOP-4MG6A2H2:/mnt/c/大二课程/c++/C++Code/project5$ g++ cnn.cpp  `pkg-config -
& ./a.out
Background possibility: 0.999508
Face possibility: 0.000492
The total relu time used is 38.000000 us
The total time used in processing CNN using simple method is: 15.663000ms
```

```
lmq@LAPTOP-4MG6A2H2:/mnt/c/大二课程/c++/C++Code/project5$ g++ cnn.cpp  `pkg-co
& ./a.out
Background possibility: 0.293742
Face possibility: 0.706258
The total relu time used is 47.000000 us
The total time used in processing CNN using simple method is: 21.779000ms
```

以上经过卷积出来的结果均准确地完成了识别了人脸以及背景的任务。

同样可以验证CNN编写的正确性。

Part4 - Difficulties and Solutions

卷积出现nan问题

在一开始进行卷积运算的过程中会出现经过一层卷积运算之后数组元素出现随机值，导致最后的结果出现`nan`的错误输出，经过对每一层卷积结果以及卷积前后的矩阵大小变化关系的分析，发现第一个问题是在新创建数组的时候，`new`之后没有对里面的元素进行初始化，使得在之后的矩阵运算中的`+=`运算符作用在一个不确定的数字上，这样就会导致在卷积的过程中出现`nan`的问题，进一步导致不能获取正确的卷积结果。

解决措施就是在每一次创建新的数组的时候，先判断内存有没有成功申请，如果成功申请，显式使用`memset()`函数对其初始化，全部赋值为0，之后进行其他运算的时候就会更加安全。

```
        this -> nums = new T[row * col * channel];
        if(this -> nums == nullptr){
            std::cerr << "FAILURE TO ALLOCATE MEMORY in line "
                << __LINE__ << " in file " << __FILE__ << std::endl;
            exit(EXIT_FAILURE);
        }
        memset(this -> nums, 0, sizeof(T) * (row * col * channel));
```

第二个问题在于在进行每一层的卷积运算之前都没有对其进行相应的规模的逻辑检查，导致了有几处是由于自己的疏忽使得卷积前后的矩阵规模不匹配，改进之后在每一步卷积操作前后都会对传入以及输出的矩阵规模进行检查，确保无误之后才会进行接下来的运算，这样可以有效避免卷积结果错误的问题。

矩阵存储方式对卷积运行时间的影响

由于上一次的`project`，为了访问逐个像素的便捷性，我在程序当中将属于矩阵同一个位置，不同通道的矩阵在内存当中存储在了一起，导致在本次的`project`当中，使用`OpenBLAS`的时候需要事先将矩阵元素进行逐个复制到合适的顺序之后才能进行相应的矩阵乘法操作，导致程序不能更好地加速卷积运算，然而由于上次所写的`project`代码量较大，加上意识到这一点的时候时间已经较为紧迫，所以难以将原有代码进行进一步的改造，以使用SIMD的指令集优化来进一步加速卷积运算。

Part5 -Thinking and Summary

这次的`project`是对前几次`project`的融合与创新，融合了`project4`当中的多通道矩阵存储与运算以及ARM与X86 的比较分析，`project3`当中的`cmake`编写与`OpenBLAS`的运用，创新的是灵活运用相关知识点去完成卷积神经网络的实现，是一次对矩阵更深层次的理解与运用。

通过这次`project`，我成功地完成了一个之前看似不可能的任务——实现卷积神经网络（CNN），并且对其做了不少的优化去改善它的性能。

从零开始实现卷积神经网络，首先是通过广泛查阅资料去深刻地理解了CNN各层次的结构以及相关作用，并且手搓了几个简答的矩阵真正地去模拟对应的卷积核的移动运算过程，一步步地推演矩阵规模的变换，思考其对于加速与优化CNN的作用。实现CNN的过程当中，会出现不少bug，就像上文提到的出现`nan`的问题等等，通过解决这些bug，我对进行参数检查的必要性有了更深刻的理解，虽然在实际编写代码的过程中不断地编写参数检查并编写出精确的报错信息非常耗费时间，但是这一操作在后续debug的过程中反而是效率最高的，这也影响了我对于编程的看法。

原本认为`if``max（）``?:`运算符之间并没有什么区别，但是对Relu函数进行使用三种实现方式的时间对比分析发现，这三者其实并不等价，至少在Relu的实现当中是有不小的区别的。以及使用`size_t`去存储矩阵的行列数以及通道数给程序带来了意外的加速，查阅了相关资料分析可能是改变了数组下标的类型转换使得程序加快，这也改变了我对于`size_t`类型的理解。

这次通过将X86与ARM运行作对比也让我对它们之间的区别有了更深的理解，在本次实验当中，发现ARM服务器相比于X86在内存的访问层面会表现的更加好，而在涉及具体运算的时候X86的速度要更加快，表现出来的性能也更加地好。

在这次`project`当中，我也尝试了使用并行运算机制对矩阵乘法做相关的优化，但是可能由于矩阵规模相对于前几次`project`较小的原因，运行时间在毫秒的量纲之下相差不大，甚至变得更慢了，可以说不是一次成功的尝试，这也是有遗憾的地方。

实现了CNN，我也进行了对于人脸识别相关的探索，通过进行不断的尝试，得到的结果是戴上眼镜或者背景稍乱都会导致人脸识别的精确度有所下降，如果戴上口罩就会使得人脸识别的精确度大大降低，这也是一点对模型的探索。

通过这次`project`，我对问题的抽象能力有了更大的提升，将抽象的神经网络以代码的方式呈现出来也提升了自身的编程能力以及`debug`能力，又是一次难忘的经历。

以上就是我报告的全部内容，感谢于老师的阅读！