CS205:C/C++ Program - Project Report 5

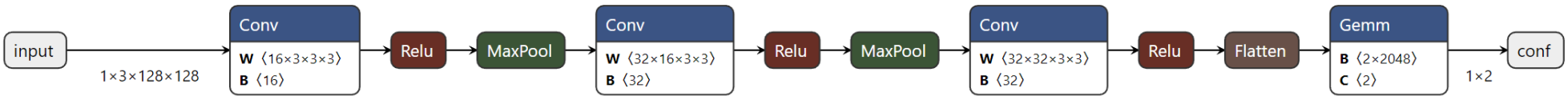
Simple CNN Face Detection

> 姓名：张闻城
>
> 学号：*12010324*

目录

1. 需求分析

项目要求实现建议的神经卷积网络(只需完成前向计算层，反向传输层不做要求)。其神经网络模型如下图。



该模型要求够预测一个尺寸为128×128的图片中含有人脸的概率。

2. 代码及特点

2.1 `macro.hpp`

```
#ifndef CNN_FACEDETECTION_MACRO_HPP
#define CNN_FACEDETECTION_MACRO_HPP

#if defined(NDEBUG)
#ifndef ONLY4DEBUG
#define ONLY4DEBUG(statement)
#endif
#else
```

```
#ifndef ONLY4DEBUG
#define ONLY4DEBUG(statement) statement
#endif
#endif

#if defined(DEBUG)
#ifndef ASSERT
#define ASSERT(expr, error_message) \
if (!(expr)) \
{ \
    fprintf(stderr, "\033[31mfile: %s => line: %d => func: %s => Assertion \'%s\' failed => \'%s\'\033[0m\n", \
    __FILE__, __LINE__, __FUNCTION__, #expr, error_message); \
    exit(EXIT_FAILURE); \
}
#endif
#else
#ifndef ASSERT
#define ASSERT(expr, error_message) \
if (!(expr)) \
    exit(EXIT_FAILURE);
#endif
#endif

#ifndef force_cast
#define force_cast(type) (type)
#endif

#ifndef DECORATOR_REDIRECT
#define DECORATOR_REDIRECT(filename, statement) \
freopen(filename, "w", stdout);        \
statement                                  \
freopen("/dev/tty", "w", stdout);
#endif

#ifndef REDIRECT2
#define REDIRECT2(file) freopen(file, "w", stdout);
#endif

#ifndef REDIRECT2STDOUT
#define REDIRECT2STDOUT freopen("/dev/tty", "w", stdout);
#endif

#ifndef EXIT
#define EXIT exit(EXIT_FAILURE);
#endif

#ifndef SWAP
#define SWAP(a, b) { (a) = (b) - (a); (b) = (b) - (a); (a) = (a) + (b); }
#endif

#endif //CNN_FACEDETECTION_MACRO_HPP
```

`macro.hpp`内主要定义了一些在编写程序及程序运行时常用的宏。

- `ONLY4DEBUG(statement)`：参数`statement`为一语句块。传入该宏的语句只有在DEBUG模式下才会执行；

- `ASSERT(expr, error_message)`：自定义的类似于C++中的断言`assert(expr)`的宏。在`expr`不满足时，将会输出一系列报错信息，并终止当前程序。该宏只有在DEBUG模式下才会启用；
- `force_cast(type)`：用于强制类型转换。虽然C++有四种类型转换运算符(`static_cast`，`dynamic_cast`，`const_cast`，`reinterpret_cast`)，但并没有适用于不同基本数据类型间强制转换的类型转换运算符。因此自定义一个宏用于强制类型转换。这样方便在Debug时直接定位到哪些地方进行了强制类型转换；
- `DECORATOR_REDIRECT(filename, statement)`：该宏实现类类似于Python中的修饰器的功能。该修饰器主要用于将`statement`代码块中输出的语句重定向到文件`filename`中；
- `REDIRECT2(file)`：将输出流重定向到文件`file`中；
- `REDIRECT2STDOUT`：将输出流重定向到标准输出上。

2.2 `facedetection.hpp`

```cpp
#ifndef CNN_FACEDETECTION_FACEDETECTION_HPP
#define CNN_FACEDETECTION_FACEDETECTION_HPP

#include <opencv2/opencv.hpp>

#include "macro.hpp"
#include "facedetection-data.hpp"
#include "mat.hpp"

using namespace std;
using namespace cv;

class CDataBlob
{
public:
    int rows{};
    int cols{};
    int channels{};
    float *data{};

    CDataBlob() = default;
    CDataBlob(const string &srcPath, const Size &size);
    ~CDataBlob();

    void init(int _rows, int _cols, int _channels);
    void set(int rows_, int cols_, int channels_, float *&data_);
    void setnull();
    float operator()(int rowIndex, int colIndex, int channelIndex) const;
    [[nodiscard]] int total() const;
    [[nodiscard]] bool isValid() const;
};

class Filter
{
public:
    int rows{};
    int cols{};
    int channels{};
    int kernels{};
    int padding{};
    int stride{};
    float *const weights{};
    float *const bias{};

    Filter() = default;
```

```
46      explicit Filter(const ConvParam &param);
47      ~Filter();
48
49      void setnull();
50      float operator()(int rowIndex, int colIndex, int channelIndex, int kernelIndex) const;
51      float operator()(int kernelIndex) const;
52      [[nodiscard]] bool isValid() const;
53  };
54
55  namespace cnn
56  {
57      float *allocate(int length, float initialValue);
58      bool linearNormalization(float *data, int dataLength, float lowerBound, float upperBound);
59      void facedetection128x128SimpleConv(const string &srcPath);
60      void facedetection128x128MatMultiplication(const string &srcPath);
61      bool convReLU(CDataBlob &blob, Filter &filter, CDataBlob &resBlob);
62      bool maxPooling(CDataBlob &src, CDataBlob &res);
63      bool im2colConvReLU(const CDataBlob &blob, const Filter &filter, CDataBlob &resBlob);
64  }
65
66  #endif //CNN_FACEDETECTION_FACEDETECTION_HPP
```

2.2.1 类`CDataBlob`

用于表示图像数据的类。图像的数据、卷积后的数据、池化后的数据以及其必要的信息都将由该类存储。

1. 属性：
    - `rows`：每个通道的行数；
    - `cols`：每个通道的列数；
    - `channels`：通道的个数；
    - `data`：指向数据内存的指针。
2. 函数：

- `CDataBlob(const string &srcPath, const Size &size)`：从`srcPath`中读取图像，并将图像拉伸或缩放到由`size`指定的大小。`Size`为`cv`中的类；
- `init(int _rows, int _cols, int _channels)`：将当前`CDataBlob`对象初始化，并为其分配指定大小的动态内存；
- `setnull()`：将当前对象置空(先释放`data`内存，并将所有属性的值置为0)；
- `operator()(int rowIndex, int colIndex, int channelIndex)`：用于获取位于指定行，列，及通道的元素的值；
- `at(int rowIndex, int colIndex, int channelIndex)`：用于修改位于指定行，列，及通道的元素的值(返回引用)；
- `total()`：返回改对象中储存了多少个`float`元素；
- `isValid()`：判断当前矩阵是否是有效的。

2.2.2 类`Filter`

类`Filter`：用来表示再每层卷积操作中所使用到的过滤器。

1. 属性：
    - `kernels`：过滤器所含有的卷积核的个数；
    - `padding`：填充数；
    - `stride`：步长；
    - `weights`：权重矩阵，即卷积核；
    - `bias`：偏差矩阵。
2. 函数：
    - `Filter(const conv_param &param)`：从结构体`conv_param`中读取数据到`Filter`当中；

- `init(int _rows, int _cols, int _channels, int _kernels)`：将当前`Filter`对象初始化，并为其分配指定大小的动态内存；
- `setnull()`：将当前对象置空(先将`weights`和`bias`释放，再将其他属性置零)；
- `operator()(int kernelIndex)`：用于获取过滤器`bias`的指定元素的值；
- `at(int kernelIndex)`：用于修改过滤器`bias`的指定元素的值(返回引用)；
- `operator()(int rowIndex, int colIndex, int channelIndex, int kernelIndex)`：用于获取过滤器`weights`中指定卷积核，行，列，通道的元素的值；
- `at(int rowIndex, int colIndex, int channelIndex, int kernelIndex)`：用于修改过滤器`weights`中指定卷积核，行，列，通道的元素的值(返回引用)；
- `isValid()`：判断当前过滤器是否是有效的。

2.2.3 命名空间`cnn`

内部定义了一些在卷积神经网络中将会使用到的函数

- `facedetection128x128(const string &srcPath)`：计算大小为128×128的图片中是否含有人脸的概率；
- `linearNormalization(float *data, int dataLength, float lowerBound, float upperBound)`，将长度为`dataLength`的`data`数据线性归一化到区间[lowerBound, upperBound]；
- `convBNReLU(CDataBlob &blob, Filter &filter, CDataBlob &resBlob)`：卷积操作，内部同时对数据进行非线性激励(ReLU)。卷积结果储存在参数`resBolob`中；
- `batchNormalization(CDataBlob &blob)`：对`blob`内部存储的数据进行Batch Normalization；
- `maxPooling(CDataBlob &src, CDataBlob &res)`：最大池化操作，结果储存在参数`res`中；
- `im2colConvReLU(CDataBlob &blob, Filter &filter, CDataBlob &resBlob)`：由矩阵乘法实现的卷积，同时对数据进行非线性激励(ReLU)。

2.3 `facedetection.cpp`

2.3.1 基础卷积实现的人脸识别

- `facedetection128x128SimpleConv()`：

```
void cnn::facedetection128x128SimpleConv(const string &srcPath)
{
    TickMeter tick;
    tick.start();

    Filter conv_1_filter(conv_params[0]);
    Filter conv_2_filter(conv_params[1]);
    Filter conv_3_filter(conv_params[2]);
    CDataBlob img(srcPath, Size(128, 128));
    CDataBlob conv_1_blob, conv_2_blob, conv_3_blob;      //CDataBlob before max pooling
    CDataBlob maxPool_1_blob, maxPool_2_blob;             //CDataBlob after max pooling

    /*<------------------Layer 1------------------>*/
    cnn::convReLU(img, conv_1_filter, conv_1_blob);
    cnn::maxPooling(conv_1_blob, maxPool_1_blob);

    /*<------------------Layer 2------------------>*/
    cnn::convReLU(maxPool_1_blob, conv_2_filter, conv_2_blob);
    cnn::maxPooling(conv_2_blob, maxPool_2_blob);

    /*<------------------Layer 3------------------>*/
    cnn::convReLU(maxPool_2_blob, conv_3_filter, conv_3_blob);

```

```
    /*<------------------Fully-Connected Layer------------------>*/
    auto conf = new float[2]{0.0f};
    for (int i = 0; i < 2048; ++i)
    {
        conf[0] += fc_params[0].p_weight[i] * conv_3_blob.data[i];
        conf[1] += fc_params[0].p_weight[i + 2048] * conv_3_blob.data[i];
    }
    conf[0] += fc_params[0].p_bias[0];
    conf[1] += fc_params[0].p_bias[1];

    /*<------------------Softmax------------------>*/
    conf[0] = exp(conf[0]) / (exp(conf[0]) + exp(conf[1]));
    conf[1] = exp(conf[1]) / (exp(conf[0]) + exp(conf[1]));


    printf("\033[32mImage <%s>: [Background Confidence = %.6f, ", srcPath.c_str(), conf[0]);
    printf("Human Face Confidence = %.6f] \033[0m", conf[1]);

    tick.stop();
    printf("=> \033[35mTotal time consumption [simple convolution] = %g ms\033[0m\n",
           tick.getTimeMilli());
}
```

在该函数中将对输入图像的数据读入、卷积、激励函数、最大池化、**Softmax**等操作封装起来，以便于计算输入图像是人脸和背景的概率分别是多少。该方法只适用于对于尺寸为128×128的图像进行人脸识别。若图像尺寸不满足该条件，程序会手动将图像拉伸为128×128。由于在设计`CDataBlob`时，其内存存储结构在内存上就是一维连续的，因此不需要Flatten的操作便可以直接进入全连接层。

- `convReLU(CDataBlob &blob, Filter &filter, CDataBlob &resBlob)`(卷积层+激励层)

```
bool cnn::convReLU(CDataBlob &blob, Filter &filter, CDataBlob &resBlob)
{
    ASSERT((blob.isValid() && filter.isValid()), "Invalid input blob or filter")
    ASSERT((blob.channels == filter.channels),
           "Inconsistent channels for input CDataBlob and Filter")

    resBlob.setnull();

    int out_rows = floor((blob.rows + 2 * filter.padding - filter.rows) / filter.stride + 1);
    int out_cols = floor((blob.cols + 2 * filter.padding - filter.cols) / filter.stride + 1);
    resBlob.init(out_rows, out_cols, filter.kernels);

    /*---------------------------Convolution----------------------------*/
    size_t index = 0;
    for (int k = 0; k < filter.kernels; ++k)
    {
        for (int i = -filter.padding, x = 0; x < resBlob.rows; i += filter.stride, ++x)
        {
            for (int j = -filter.padding, y = 0; y < resBlob.cols; j += filter.stride, ++y)
            {
                for (int c = 0; c < filter.channels; ++c)
                {
                    resBlob.data[index] +=
                            blob(i + 0, j + 0, c) * filter(0, 0, c, k) +
                            blob(i + 0, j + 1, c) * filter(0, 1, c, k) +
                            blob(i + 0, j + 2, c) * filter(0, 2, c, k) +
                            blob(i + 1, j + 0, c) * filter(1, 0, c, k) +
```

```
                            blob(i + 1, j + 1, c) * filter(1, 1, c, k) +
                            blob(i + 1, j + 2, c) * filter(1, 2, c, k) +
                            blob(i + 2, j + 0, c) * filter(2, 0, c, k) +
                            blob(i + 2, j + 1, c) * filter(2, 1, c, k) +
                            blob(i + 2, j + 2, c) * filter(2, 2, c, k);
                }
                resBlob.data[index++] += filter(k);
            }
        }
    }
    /*------------------------------------------------------------------*/

    /*-------------------------ReLU Function-------------------------*/
#pragma omp parallel for
    for (int i = 0; i < resBlob.channels * resBlob.rows * resBlob.cols; ++i)
    {
        resBlob.data[i] = 0.0f + force_cast(float) (resBlob.data[i] >= 0) * resBlob.data[i];
    }
    /*------------------------------------------------------------------*/

    ONLY4DEBUG(
            printf("\033[34mConvBNReLU: Batch<%dx%dx%d> * Filter[%d, %d, <%dx%dx%dx%d>] =>
Batch<%dx%dx%d>\033[0m\n",
                   blob.channels, blob.rows, blob.cols, filter.padding, filter.stride,
                   filter.kernels, filter.channels, filter.rows, filter.cols, resBlob.channels,
                   resBlob.rows, resBlob.cols);)

    return true;
}
```

对于一个$n \times n$的batch(矩阵)，使用$f \times f$的卷积核，步长为s，填充为p，对其进行卷积扫描，输出图像的尺寸可通过公式$\lfloor \frac{n+2p-f}{s} + 1 \rfloor$得到。类`CDataBlob`的`at()`函数用于返回对应行、列及通道的元素的引用。同时类`CDataBlob`重载了`operator()`，用于获取对应行，列及通道的元素的值，但若行索引或列索引越界的时候，该函数会返回`0.0f`。类`Filter`重载了`operator()`，用于获取过滤器对应卷积核、行、列及通道的元素的值。在卷积操作中(四重循环)，有最外层到最内层的循环顺序是：
卷积核 $\rightarrow$ 行 $\rightarrow$ 列 $\rightarrow$ 卷积核的某一通道。行循环的行索引从`-filter.padding`开始，列循环的列索引从`filter.padding`开始，循环初始索引值这样设置的原因是：当索引越界时(索引小于0或大于等于`blob`的尺寸)，运算符`blob()`返回的值为0。这样在逻辑上就相当于在`blob`的数据周围填充了`filtet.padding`圈值全为0的元素。这种写法较为优雅的实现了含有padding的卷积操作(不优雅的做法是在**padding**不为0的时候，要根据当前扫描的位置分成不同的case来进行计算)。

- 访问`blob`中的元素时，不直接计算下标`index`并用`blob.data[index]`获取元素值，而使用`operator()`的原因是：

  当卷积核`padding`不为*0*时，在逻辑上`blob.data`的四周填充了`padding`圈*0*。要实现这种效果需要对访问`blob.data`时的下标进行判断：若越界，则获取到的值应该是*0*；反之则直接获取`blob.data`相应位置的元素。在这种情况下，将获取`blob`元素的过程封装成一个`inline`函数是比较优雅和高效的做法。通过反汇编的方法[1]可以验证该函数被*inline*成功；
- 访问`filter.weights`中的元素时，不直接计算下标`index`并用`filiter.weights[index]`获取元素值，而使用`operator()`函数的原因是：

  该函数为*inline*函数，通过反汇编可验证该函数被*inline*成功，因此不会产生函数的入栈和出栈而降低运行效率的问题。且这种写法比较优雅，简洁；
- 卷积中未采用*OpenMP*进行加速的原因是：

  所有元素的类型都为*float*，因此在计算的过程中需要保证线程同步，否则计算得到的值很大概率是不正确的。虽然可以通过`#paragma omp critical {\*code to be sychronized*\}`来保证线程同步，但是由于尺寸为128×128的图像卷积的计算量并不算大，使用多线程加速由于线程开辟、通信、销毁也需要一定的时间开销，加速收益并不明显。因此此处未采用*omp*加速。

在卷积完成后，该函数还会对数据进行非线性激励。此处采用的激励函数为最简单的ReLU函数。通常实现ReLU函数的方式是遍历矩阵的元素，遍历的同时判断元素是否小于0：若小于0，则将该元素置位0；否则该元素值不变。其代码实现如下：

```
for (int i = 0; i < resBlob.channels * resBlob.rows * resBlob.cols; ++i)
{
    if (resBlob.data[i] < 0)
        resBlob.data[i] = 0.0f;
}
```

这种实现方式虽然简单，但是并不高效和优雅。其原因是if语句在执行的过程中会进行跳转，破坏了程序执行的连续性，效率因此有一定程度的下降。另一种ReLU函数的实现如下：

```
for (int i = 0; i < resBlob.channels * resBlob.rows * resBlob.cols; ++i)
{
    resBlob.data[i] = 0.0f + force_cast(float) (resBlob.data[i] >= 0) * resBlob.data[i];
}
```

该种实现方法的思路是：将条件判断`resBlob.data[i] >= 0`的值与`resBlob.data[i]`相乘：若条件成立，则条件表达式返回1，`resBlob.data[i]`乘以1仍然为原来的值；反之条件表达式返回0，`resBlob.data[i]`乘以0得到0。这种实现方式不的`resBlob.data[i]`的值一次性计算得出，不存在由于if语句跳转所带来的性能上的损失。同时通过`#pragma omp parallel for`将循环分给多个线程计算，可进一步加速计算过程。

- `maxPooling(CDataBlob &src, CDataBlob &res)`*(池化层)：

```
void cnn::maxPooling(CDataBlob &src, CDataBlob &res)
{
    Assert(src.isValid(), "Parameter [src] is invalid")

    res.setnull();
    res.init(src.rows / 2, src.cols / 2, src.channels);

    for (int k = 0; k < src.channels; ++k)
    {
        for (int i = 0, x = 0; i < src.rows; i += 2, ++x)
        {
            for (int j = 0, y = 0; j < src.cols; j += 2, ++y)
            {
                float max = src.data[k * src.rows * src.cols + i * src.cols + j];
                if (src.data[k * src.rows * src.cols + i * src.cols + (j + 1)] > max)
                    max = src.data[k * src.rows * src.cols + i * src.cols + (j + 1)];
                if (src.data[k * src.rows * src.cols + (i + 1) * src.cols + j] > max)
                    max = src.data[k * src.rows * src.cols + (i + 1) * src.cols + j];
                if (src.data[k * src.rows * src.cols + (i + 1) * src.cols + (j + 1)] > max)
                    max = src.data[k * src.rows * src.cols + (i + 1) * src.cols + (j + 1)];
                res.data[k * res.rows * res.cols + x * res.cols + y] = max;
            }
        }
    }

    DEBUG_CODE(printf("\033[34mMaxPooling Downsampled: <%dx%dx%d> => <%dx%dx%d>\033[0m\n",
                      src.channels, src.rows, src.cols,
                      res.channels, res.rows, res.cols);)
}
```

在该项目的池化层中采用的是最大池化。最大池化的加速策略是：

1. 使用OpenMP加速；
2. 循环展开获取最大值。

其实在循环内部比较并获取最大值的时候，也可以采取类似于ReLU中的写法，将一个条件判断语句写成一个连续的计算。例：

```
void cnn::maxPooling(CDataBlob &src, CDataBlob &res)
{
    ...

    for (int k = 0; k < src.channels; ++k)
    {
        for (int i = 0, x = 0; i < src.rows; i += 2, ++x)
        {
            for (int j = 0, y = 0; j < src.cols; j += 2, ++y)
            {
                float max = src.data[k * src.rows * src.cols + i * src.cols + j];
                max = (src.data[k * src.rows * src.cols + i * src.cols + (j + 1)] > max) *
(src.data[k * src.rows * src.cols + i * src.cols + (j + 1)] - max) + max;
                ...
            }
        }
    }
    ...
}
```

这种写法看似可以实现程序运行的连续性，避免条件跳转造成程序效率下降的问题，但是与ReLU中的写法不同的是：ReLU在条件判断以后便可直接计算出最终结果；而在求最大值的条件判断后还需要先计算`src.data[k * src.rows * src.cols + i * src.cols + (j + 1)] - max`的结果，然后才能得出最终答案。这种情况下先进行一系列加减乘运算的耗时显然要比用一个if进行条件跳转的耗时要长。因此此处不适合采用类似ReLU中的写法。

2.3.2 矩阵乘法实现的卷积

- `facedetection128x128UsingSgemm()`：

```
void cnn::facedetection128x128UsingSgemm(const string &srcPath)
{
    TickMeter tick;
    tick.start();

    Filter conv_1_filter(conv_params[0]);
    Filter conv_2_filter(conv_params[1]);
    Filter conv_3_filter(conv_params[2]);
    CDataBlob img(srcPath, Size(128, 128));
    CDataBlob conv_1_blob, conv_2_blob, conv_3_blob;    //CDataBlob before max pooling
    CDataBlob maxPool_1_blob, maxPool_2_blob;           //CDataBlob after max pooling

    /*<------------------Layer 1------------------>*/
    cnn::im2colConvReLU(img, conv_1_filter, conv_1_blob);
    cnn::maxPooling(conv_1_blob, maxPool_1_blob);

    /*<------------------Layer 2------------------>*/
    cnn::im2colConvReLU(maxPool_1_blob, conv_2_filter, conv_2_blob);
    cnn::maxPooling(conv_2_blob, maxPool_2_blob);

    /*<------------------Layer 3------------------>*/
    cnn::im2colConvReLU(maxPool_2_blob, conv_3_filter, conv_3_blob);
```

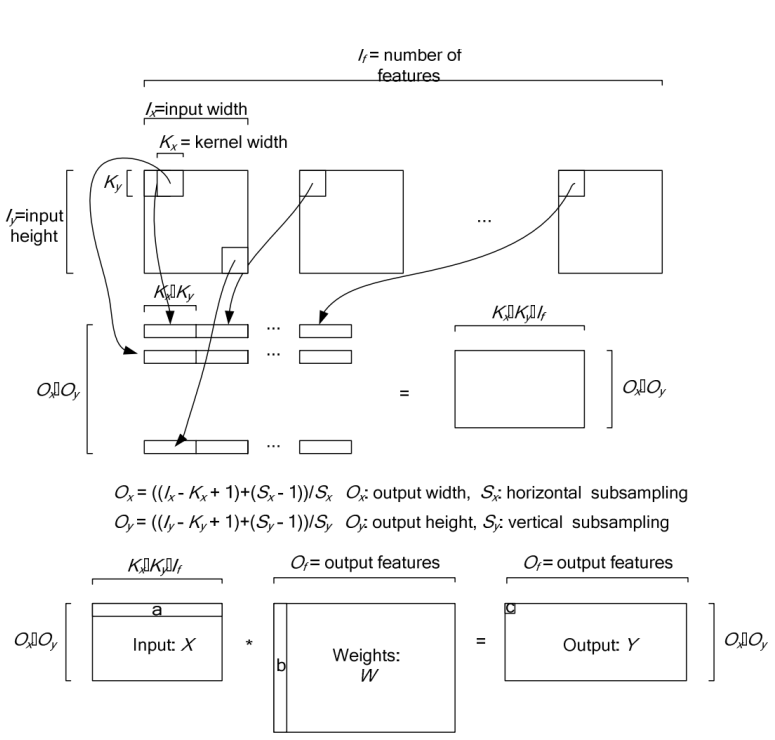
```
    /*<------------------Fully-Connected Layer------------------>*/
    auto conf = new float[2]{0.0f};
    for (int i = 0; i < 2048; ++i)
    {
        conf[0] += fc_params[0].p_weight[i] * conv_3_blob.data[i];
        conf[1] += fc_params[0].p_weight[i + 2048] * conv_3_blob.data[i];
    }
    conf[0] += fc_params[0].p_bias[0];
    conf[1] += fc_params[0].p_bias[1];

    /*<------------------Softmax------------------>*/
    conf[0] = exp(conf[0]) / (exp(conf[0]) + exp(conf[1]));
    conf[1] = exp(conf[1]) / (exp(conf[0]) + exp(conf[1]));


    printf("\033[32mImage <%s>: [Background Confidence = %.6f, ", srcPath.c_str(), conf[0]);
    printf("Human Face Confidence = %.6f] \033[0m", conf[1]);

    tick.stop();
    printf("=> \033[35mTotal time consumption using [matrix sgemm] = %g ms\033[0m\n",
           tick.getTimeMilli());
}
```

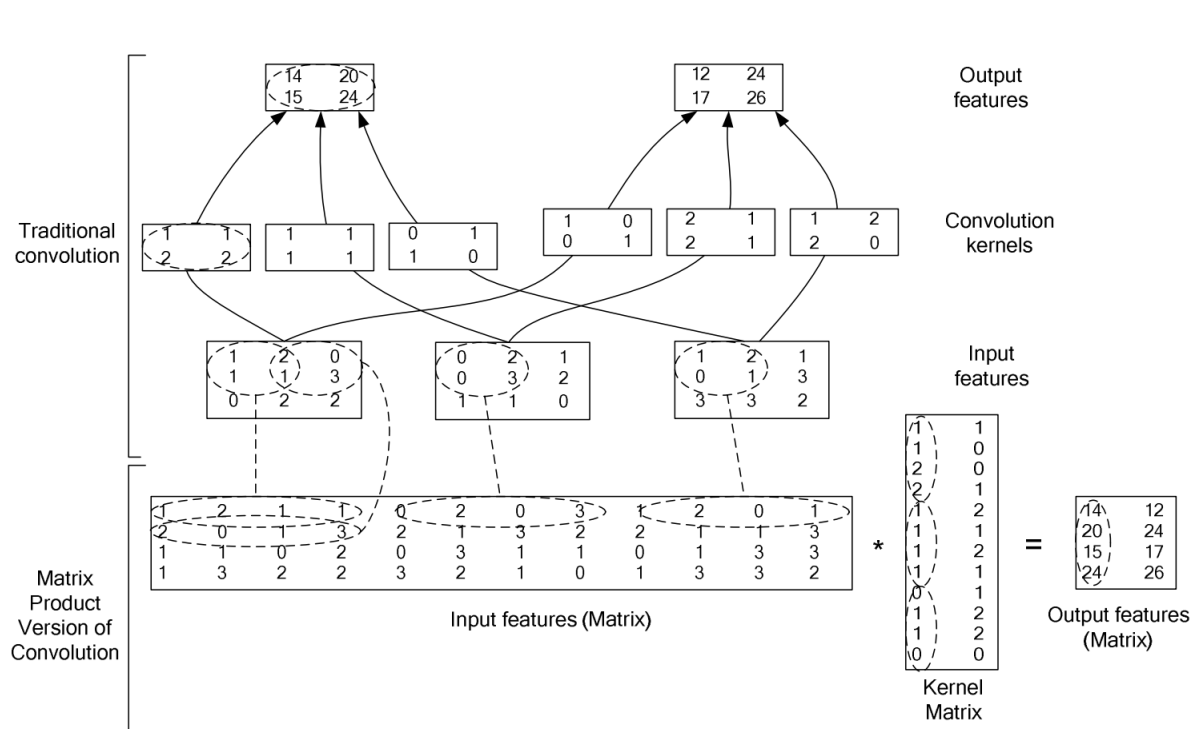
- `im2colConvReLU()`:

```
void cnn::im2colConvReLU(const CDataBlob &blob, const Filter &filter, CDataBlob &resBlob)
{
    ASSERT(blob.isValid() || filter.isValid(), "Invalid blob or filter")
    ASSERT(blob.channels == filter.channels, "Inconsistent channels of blob and filter")

    /*<--------------------Filter Part--------------------->*/
    CMat filterMat(filter.kernels, filter.rows * filter.cols * filter.channels);
    memcpy(filterMat.data, filter.weights, sizeof(float) * filterMat.rows * filterMat.cols);
    filterMat.transfer();

    /*<--------------------CDataBlob Part--------------------->*/
    size_t out_rows = floor((blob.rows + 2 * filter.padding - filter.rows) / filter.stride + 1);
    size_t out_cols = floor((blob.cols + 2 * filter.padding - filter.cols) / filter.stride + 1);
    CMat blobMat(out_rows * out_cols, filterMat.rows);
    size_t index = 0;
    for (int i = -filter.padding, x = 0; x < out_rows; i += filter.stride, ++x)
    {
        for (int j = -filter.padding, y = 0; y < out_cols; j += filter.stride, ++y)
        {
            for (int c = 0; c < filter.channels; ++c)
            {
                blobMat.data[index++] = blob(i + 0, j + 0, c);
                blobMat.data[index++] = blob(i + 0, j + 1, c);
                blobMat.data[index++] = blob(i + 0, j + 2, c);
                blobMat.data[index++] = blob(i + 1, j + 0, c);
                blobMat.data[index++] = blob(i + 1, j + 1, c);
                blobMat.data[index++] = blob(i + 1, j + 2, c);
                blobMat.data[index++] = blob(i + 2, j + 0, c);
                blobMat.data[index++] = blob(i + 2, j + 1, c);
                blobMat.data[index++] = blob(i + 2, j + 2, c);
            }
```

```
        }
    }

    CMat resMat;
    CMat::sgemm(blobMat, filterMat, resMat);

    for (size_t j = 0; j < resMat.cols; ++j)
    {
        for (size_t i = 0; i < resMat.rows; ++i)
        {
            resMat.data[i * resMat.cols + j] += filter.bias[j];
        }
    }

    for (size_t i = 0; i < resMat.rows * resMat.cols; ++i)
    {
        resMat.data[i] = force_cast(float) (resMat.data[i] > 0) * resMat.data[i] + 0.0f;
    }

    resMat.transfer();

    resBlob.set(force_cast(int) out_rows, force_cast(int) out_cols, filter.kernels,
resMat.data);

    ONLY4DEBUG(
            printf("\033[34mConvBNReLU: Batch<%dx%dx%d> * Filter[%d, %d, <%dx%dx%dx%d>] =>
Batch<%dx%dx%d>\033[0m\n",
                   blob.channels, blob.rows, blob.cols, filter.padding, filter.stride,
                   filter.kernels, filter.channels, filter.rows, filter.cols, resBlob.channels,
                   resBlob.rows, resBlob.cols);
    )
}
```

在进行卷积时，结果的每一个元素，都是由blob的一小部分与卷积核对应通道对应位置相乘再相加所得到。这实质上就是向量点乘，进行点乘的两个向量分别是：由blob一小部分(尺寸与卷积核相同)的所有通道展开成一个一维行向量；过滤器的一个卷积核的所有通道展开成一个一维列向量。通过这种思想，对于一个$n \times n$的batch与k个$f \times f$的卷积核，通道数都为c，进行步长为s，填充为p的卷积层，可以分别将batch展开成一个$\lfloor \frac{n+2p-f}{s} + 1 \rfloor^2 \times (f * f * c)$的矩阵，过滤器展开成一个$(f * f * c) \times k$的矩阵。两个矩阵相乘后的结果便是卷积的结果。





卷积的本质就是向量点乘，因此可以通过矩阵乘法的方式实现卷积操作。在进行矩阵乘法之前，需要对`blob`和卷积核展开[2]。通过矩阵乘法实现卷积的优势是：普通的卷积由于内存不连续，以及数据排列方式的问题无法对其大幅度的优化加速；转换成矩阵相乘后，可利用访存优化、Strassen算法、SIMD指令、多线程、CUDA等一系列方式进行加速。当图像规模变大时，优化后的矩阵乘法卷积效率要明显优于普通卷积。

> - 本次*Project*中使用的`CMat`类是*Project 4*中`Mat`类的简化版本。详细实现细节请查看附件的代码；
> - 对于如何加速矩阵乘法，在*Project 3*中已有过详细的研究和讨论。由于本次的重点不在于矩阵乘法的加速，因此对这一部分没有过多提及。详细加速方案的细节请查看*Project 3*。

关于为什么矩阵乘法实现的卷积效率要比普通卷积要高，个人认为主要原因是运算的连续性。在普通卷积中，卷积核扫描batch的一子区域，子区域与子区域之间是不内存不连续的，这导致运算时的cache命中率较小，从内存中某一个地址取值的过程需要消耗更多的时间。若先将batch和过滤器展开成矩阵，在进行卷积时，每次点乘过程中的运算是连续的，cache命中率有所提升，在I/O上耗时更少，因此速度更加快。同时转换成矩阵后，在保证了内存的连续性的基础上可以采取更加多样化的加速方式。因此，由矩阵乘法实现的卷积是跟为明智和高效的选择。

2.4 代码特点

类`CDataBlob`和类`Filter`都有其对应的`init()`和`setnull()`函数。`init()`用于对当前对象进行一系列的初始化操作，如：属性的赋值、内存的开辟等等。`setnull()`用于将当前对象的数据进行置空(先将动态内存释放，再将所有属性置0)。如：`Filter`的`init()`和`setnull()`：

```
1  void Filter::init(int _rows, int _cols, int _channels, int _kernels)
2  {
3      Assert(_rows >= 1 && _cols >= 1 && _channels >= 1 && _kernels >= 1, "Invalid parameter for
   initializing filter")
4
5      setnull();      //在初始化之前先对this进行setnull，防止内存泄漏
6      rows = _rows;   //对一系列参数进行赋值
7      cols = _cols;
8      channels = _channels;
9      kernels = _kernels;
10     weights = cnn::allocate(kernels * channels * rows * cols, 0.0f);     //为weights开辟动态内存
11     bias = cnn::allocate(kernels, 0.0f);                                 //为bias开辟动态内存
12 }
13
14 void Filter::setnull()
15 {
16     if (weights != nullptr)
17     {
18         free(weights);
19         weights = nullptr;
20     }
21     if (bias != nullptr)
22     {
23         free(bias);
24         bias = nullptr;
25     }
26     rows = cols = channels = kernels = padding = stride = 0;
27 }
```

在经过Pro. Yu课堂上对于Project的提醒后，此次Project与以往最大的不同在于每个函数在进行主体运算之前都会对传入的参数以及所需要用到的数据进行有效性的检查。例如：

- `convReLu()`卷积函数中先对`blob`，`filter`进行检查：

```
void cnn::convReLU(CDataBlob &blob, Filter &filter, CDataBlob &resBlob)
{
    Assert((blob.isValid() && filter.isValid()), "Invalid input blob or filter")
    Assert((blob.channels == filter.channels), "Inconsistent channels for input CDataBlob and Filter")

    resBlob.setnull();

    ...

}
```

先对`blob`和`filter`的有效性进行检查，检测其相应的数据是否加载正确(`blob`的`data`是否为`nullptr`；`filter`的`weights`和`bias`是否为`nullptr`)，然后再检查`blob`和`filter`的通道数是否一样。

- `linearNormalization()`，在进行线性归一化之前，先检查`data`是否为`nullptr`：

```
bool cnn::linearNormalization(float *data, int dataLength, float lowerBound, float upperBound)
{
    Assert(data != nullptr, "[data] has not been allocated")
    Assert(dataLength > 0, "Invalid size of data for normalization")

    ...

}
```

在对参数进行检查后，程序会通过一些输出语句反馈给程序员，便于程序员及时发现错误和Debug。

除了增加了对参数的检查之外，本程序还禁止函数返回非基本数据类型。一是若直接将这些非基本数据变量返回，需要进行内存拷贝的操作，造成了不必要的时间消耗；二是若将函数内部创建的非基本数据类型变量直接返回，有可能造成内存管理方面的问题(如：内存泄漏、内存被提前释放、内存多次释放等)。因此本程序中对于该类函数的处理是：在函数外部由用户自己创建用于一个储存函数结果的对象并当作参数传给函数。函数在内部会先对该参数进行`setnull()`以防止内存泄漏。如：

```
void cnn::convReLU(CDataBlob &blob, Filter &filter, CDataBlob &resBlob)
{
    ... //参数检查

    resBlob.setnull();

    ...

}
```

```
void cnn::maxPooling(CDataBlob &src, CDataBlob &res)
{
    ... //参数检查

    res.setnull();

    ...
}
```

3. 测试结果及实验验证

> 本地测试环境：
>
> *OS：Ubuntu 20.04.1*
>
> *CPU：AMD Ryzen 9 5900HS with Radeon Graphics @ 3.30 GHz*
>
> *RAM：16.0 GB (15.4 GB availiable)*
>
> *GPU：NVIDIA GeForce RTX 3060 Laptop GPU*
>
> 远程测试环境：
>
> *OS：Linux*
>
> *CPU Architecture：aarch64 64-bit*
>
> *RAM：2.9 GB*

图片说明	人脸概率	背景概率	附图
标准人脸测试用例	0.999905	0.000000	
标准背景测试用例	0.000211	1.000000	
蒂法	0.994583	0.000027	
海滩	0.000018	1.000000	
本人正脸照	0.954361	0.002151	
银河系	0.054717	0.978368	

图片说明	人脸概率	背景概率	附图
“双料”高级特工——“穿山甲”	0.923201	0.008270	
日落	0.016847	0.998199	

由测试结果大体可得知本程序能够较为准确的识别出输入的图片是否含有人脸。

图片说明	**X86**平台下运行速度**(SIMPLE CONVOLUTION)**/ms	**X86**平台下运行速度**(SGEMM)**/ms	**ARM**平台下运行速度**(SIMPLE CONVOLUTION)**/ms	**ARM**平台下运行速度**(SGEMM)**/ms
标准人脸测试用例	6.78891	4.13822	12.2976	8.74039
标准背景测试用例	7.08234	1.99382	11.9949	8.4658
蒂法	7.34527	2.04712	12.5262	8.86585
海滩	7.31721	2.18162	12.2978	8.96432
本人正脸照	7.33648	2.13352	12.0897	8.72683
银河系	7.38558	1.88146	11.8746	8.6116
“双料”高级特工——“穿山甲”	7.38696	1.83055	11.9587	8.55833
日落	8.98457	1.99997	11.8659	8.52646

(以上结果为10次独立重复实验的均值)

由表格中的结果对比可得知：对于尺寸为128×128的人脸识别运算，不论是在x86还是arm平台下，矩阵乘法实现的卷积运算的效率相较于普通卷积运算有明显的提升。由此可合理推断，当图像规模变大时，矩阵乘法实现的卷积将会更加具有运算效率上的优势。这也是许多数据处理库(caffe, pytorch等)在实现卷积时是借助矩阵乘法实现的原因。

> 本次项目中的矩阵乘法只采用了访存优化*(ikj)*的方式进行加速。原因是输入图像的规模较小，其他加速方案的加速效果并不明显。测试的过程中曾对比使用*OpenBLAS*加速矩阵乘法，但最终的时间仍然比只使用访存优化所用的时间要长。当输入图像规模增大后，再考虑采取多种方式*(Strassen*算法、多线程、*SIMD*、*CUDA*等*)*加速矩阵乘法*(*详细加速方式请看*Project 3*报告*)*。

至于arm平台与x86平台运算速度的差距，这是由于本地与远程服务器的CPU性能以及RAM大小不同，导致运算性能上有所差距。

4. 困难及解决

4.1 动态开辟内存未初始化

本次Project的最初几个版本中，运算结果总是会出现`nan`这个结果。在网上查询后得知`nan`表示“not a number”，其产生原因一般是进行了一些不合法的数学运算，如：对负数开方、对负数求对数、$\frac{0.0}{0.0}$等。但在认真检查代码过后，我并没有发现类似可能产生错误的运算。之后在同学的提醒下，发现了真正导致这样结果的原因是程序在开辟了动态的内存后，没有对其中的值进行初始化。原代码如下：

```
inline float *cnn::allocate(int length, float initialValue)
{
    return static_cast<float *>(aligned_alloc(256, length * sizeof(float)));
}
```

修改后的代码如下：

```
inline float *cnn::allocate(int length, float initialValue)
{
    auto res = static_cast<float *>(aligned_alloc(256, length * sizeof(float)));
#pragma omp parallel for
    for (int i = 0; i < length; ++i)
    {
        res[i] = initialValue;
    }
    return res;
}
```

4.2 计算结果不正确

最初完成卷积操作后，算出来的结果总是与预想的相差很大。之后经过很长时间的Debug之后，发现错误的原因主要出现在在卷积操作中访问元素时的下标不正确。改正后便可计算得到正确的结果。

参考文献

1. 编译后的c/c++ 程序，如何判断一个函数是否真的被inline了？ ↩

2. im2col Convolution (opengenus.org) ↩