

CS205 C/C++ Programming - Project_5

Name: 刘乐奇 (Liu Leqi)

SID: 12011327

CS205 C/C++ Programming - Project_5

1 序言

1.1 需求分析

1.2 环境及工具

2 代码

2.1 代码文件

2.2 图片数据

2.2.1 读取和存储

2.2.2 处理

2.3 Matrix类

2.4 CNN实现

2.4.1 卷积

2.4.2 最大池化

2.4.3 全连接

2.4.4 Softmax

3 测试及分析

3.1 x86_64

3.2 ARM64

4 困难及解决

4.1 读入图像的拉伸修正

4.2 BRG转RGB的优化

4.3 动漫人物头像被识别为背景

4.4 计时问题

5 优化

5.1 O3加速

5.2 OpenMP并行优化

5.3 局部变量优化

5.4 OpenCV内置的universal intrinsics

5.5 OpenBLAS优化

6 总结

1 序言

1.1 需求分析

本次Project需要用C/C++实现一个简单是卷积神经网络 (CNN) , 包括各个layer以及将其实现得更普遍。老师提供了一个简单的预训练模型 (<https://github.com/ShiqiYu/SimpleCNNbyCPP>) , 能预测输入图像是人物还是背景。

注意! 禁止使用除了OpenCV以外的第三方库来完成图像数据的读取, 禁止使用C/C++语言以外的语言实现CNN算法。

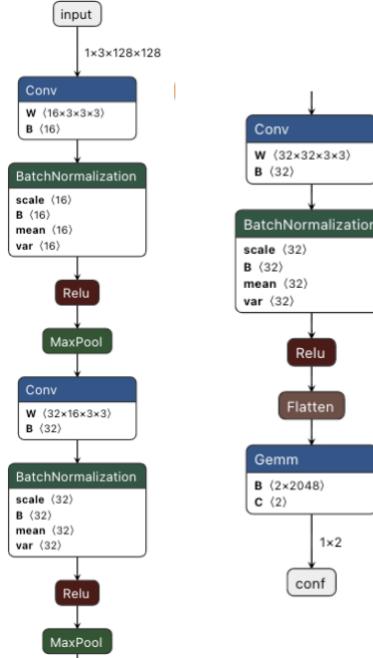
1.2 环境及工具

本程序主要使用x86的Windows系统，利用vscode及wsl编写并编译运行，同时程序也在云端的ARM服务器测试过。

2 代码

- 原理¹：

本次Project中CNN的模型包含三个卷积层 (convolutional layer) 和一个全连接层(fully connected layer)。其中，在卷积层中，需要实现卷积 (conv) , 修正线性单元 (ReLU) , 最大池化 (max pooling) 。



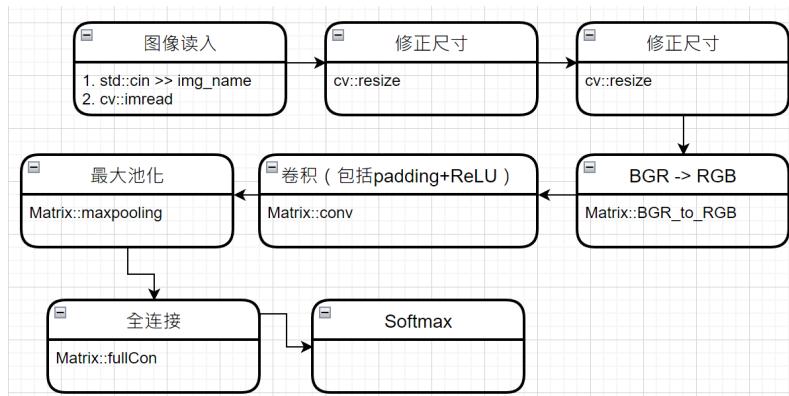
Model

- 3 Convolutional layers (Conv+BN+ReLU)
- 2 MaxPool
- 1 Full connected layer

```
self.backbone = nn.Sequential(
    ConvBNReLU(3, 16, 3, 2, 1), # downsampled by 2, 128 -> 64
    nn.MaxPool2d(2, 2), # downsampled by 2, 64 -> 32
    ConvBNReLU(16, 32, 3, 1), # keep
    nn.MaxPool2d(2, 2), # downsampled by 2, 32 -> 16
    ConvBNReLU(32, 32, 3, 2, 1) # downsampled by 2, 16 -> 8
)

self.classifier = nn.Sequential(
    nn.Linear(in_features=32*8*8,
              out_features=num_cls,
              bias=True)
)
```

- 总体流程图



2.1 代码文件

本程序包含以下三个文件。

- main.cpp 读入图像，调用各函数
- matrix.hpp 矩阵类以及卷积中的各种函数
- face_binary_cls.cpp 老师提供的kernel

如何运行？在终端中输入。

```
g++ main.cpp `pkg-config --cflags --libs opencv4`
```

若要使用OpenBLAS，则输入。

```
g++ main.cpp `pkg-config --cflags --libs opencv4` -I /opt/OpenBLAS/include/ -L/opt/OpenBLAS/lib -lopenblas
```

2.2 图片数据

2.2.1 读取和存储

用OpenCV的`cv::imread()`读取目标图片，并将相关数据存储在`cv::Mat`中。为防止输入图片名字错误，或未成功打开图片，利用`cv::Mat::empty()`来判断，并让用户重新输入图片名字。此处有一个变量`bool che`是因为最外层还有一层`while(true)`来使得程序一次运行能重复输入图片。

```
cv::Mat img;
bool che = false;
do{
    std::cout << "Please input the name of the image to be detected. Q to quit."
<< '\n';
    std::string img_name;
    std::cin >> img_name;
    if(img_name == "Q" || img_name == "q")//退出
    {
        che = true;
        break;
    }
    img_name = "./image/" + img_name;
    //图像读入
    img = cv::imread(img_name);
    if(img.empty())
    {
        std::cout << "Cannot find or open the image! Please try again." << '\n';
    } else break;
}while(true);
if(che) break;
```

2.2.2 处理

图像的尺寸会对卷积的结果产生影响（详见Part4 4.1），因此在读入图像后，要修正其尺寸²。

```
//修正尺寸
cv::Mat res;
cv::resize(img, res, cv::Size(128, 128));
```

```
int main()
{
    cv::Mat img = cv::imread("./image/t1.png");

    cout << img.type() << '\n';

    return 0;
}
```

```
Lynchrocket@LAPTOP-FV00SMG9:/mnt/d/Lynchrocket/大二上/c、c++/Project/Project5/proj5$ ./a.out
16
```

```
[root@ecs001-0021-0022 proj5]# g++ t.cpp `pkg-config --cflags --libs opencv4`  
[root@ecs001-0021-0022 proj5]# ./a.out  
16
```

	C1	C2	C3	C4
CV_8U	0	8	16	24
CV_8S	1	9	17	25
CV_16U	2	10	18	26
CV_16S	3	11	19	27
CV_32S	4	12	20	28
CV_32F	5	13	21	29
CV_64F	6	14	22	30

通过测试和查表，发现无论是x86（从上自下第二幅图）还是ARM（从上至下第三幅图），`cv::imread()`读入的图像默认是 `unsigned char` 类型（当然，这在Project文档里也有说明，不过为以防万一还是测一下）。需要首先将 `cv::Mat` 数据转成 `float` 类型，并规范化其范围为 [0.0, 1.0]。因为 `unsigned char` 类型的取值范围为 [0, 255]，故只需要除以 255.0 就能将其范围限制在 [0.0, 1.0] 中，不过要注意要先用 (`float`) 将数值强制转换为 `float` 类型。

需要注意 `cv::Mat` 中像素点颜色（通道）是 BRG (Blue, Red, Green)，而非 RGB，为适配老师提供的 `weight`，需要转成 RGB。

因为是 3 channel，一次性将三个 channel 转换，并且一次性转四个（128 的因数），能大大提高效率（详见 Part4 4.2）。

参数	解释
<code>cv::Mat& img</code>	传入的图像

```
/**  
*cv::Mat中的通道是[b1, g1, r1, b2, g2, r2, b3, g3, r3...]这样的，  
*需要改成[r1, r2, r3, ... g1, g2, g3, ... b1, b2, b3...]来适配weight  
*/  
float* BGR_to_RGB(cv::Mat& img)  
{  
    size_t row = img.rows;  
    size_t col = img.cols;  
    size_t channel = img.channels();  
  
    size_t di = row * col;  
  
    float* res = new float[row * col * channel];  
  
    for (size_t i = 0; i < row; ++i)  
    {  
        //第i行第一个数据  
        uchar* p = img.ptr<uchar>(i);  
  
        // #pragma omp parallel for  
  
        for (size_t j = 0; j < col - 1; j += 4)  
        {  
            //一次性对每一层执行四个  
            //R  
            size_t l1 = i * col + j;  
            res[l1] = (float)p[3 * j + 2] / 255;  
            res[l1 + 1] = (float)p[3 * j + 5] / 255;  
            res[l1 + 2] = (float)p[3 * j + 8] / 255;
```

```

        res[11 + 3] = (float)p[3 * j + 11] / 255;

    //G
    size_t l2 = di + i * col + j;
    res[12] = (float)p[3 * j + 1] / 255;
    res[12 + 1] = (float)p[3 * j + 4] / 255;
    res[12 + 2] = (float)p[3 * j + 7] / 255;
    res[12 + 3] = (float)p[3 * j + 10] / 255;

    //B
    size_t l3 = 2 * di + i * col + j;
    res[13] = (float)p[3 * j] / 255;
    res[13 + 1] = (float)p[3 * j + 3] / 255;
    res[13 + 2] = (float)p[3 * j + 6] / 255;
    res[13 + 3] = (float)p[3 * j + 9] / 255;
}

return res;
}

```

2.3 Matrix类

```

class Matrix
{
private:
    size_t dimension;
    size_t channel;
    size_t height;
    size_t width;
    float* data;

public:
    //构造器
    Matrix() :dimension(0), channel(0), height(0), width(0), data(nullptr) {};
    Matrix(size_t n, size_t c, size_t h, size_t w, const float* d)
    :dimension(n), channel(c), height(h), width(w)
    {
        size_t len = n * c * h * w;
        data = new float[len]();
        if(!data)
        {
            std::cerr << "Failed to allocate memory for Matrix data!" << '\n';
            exit(0);
        }
        memcpy(data, d, len * sizeof(float));
    }
    //拷贝构造
    Matrix(const Matrix& L) :dimension(L.dimension), channel(L.channel),
    height(L.height), width(L.width)
    {
        size_t len = dimension * channel * height * width;
        data = new float[len]();
        memcpy(data, L.data, len * sizeof(float));
    }
    //析构
    ~Matrix()
    {

```

```

        if(!this->data) std::cerr << "Cannot delete data[] since it is null!" <<
'\\n';
        else delete[] data;
        if(!this->data) std::cerr << "Fail to delete data[]!" << '\\n';
    }

//override:取值 A(n,c,i,j) 第n个, 第c层, 第i行, 第j列
//由于在循环中调用太费时间, 改成inline
//不检查是否越界, 因为在下方调用该inline函数保证不越界
//return the address
inline float& operator()(size_t n, size_t c, size_t i, size_t j)
{
    size_t index = n * height * width * channel + c * height * width + i *
width + j;
    return data[index];
}

//return the value
inline const float operator()(size_t n, size_t c, size_t i, size_t j) const
{
    size_t index = n * height * width * channel + c * height * width + i *
width + j;
    return data[index];
}

//assignment operator
Matrix& operator = (const Matrix& L)
{
    dimension = L.dimension;
    channel = L.channel;
    height = L.height;
    width = L.width;

    size_t len = dimension * channel * height * width;
    if (data) delete[] data;
    data = new float[len]();
    if(!data)
    {
        std::cerr << "Failed to allocate memory for Matrix data!" << '\\n';
        exit(0);
    }
    memcpy(data, L.data, len*sizeof(float));
}

//BGR->RGB
friend float* BGR_to_RGB(cv::Mat& img);

//卷积操作相关函数
//卷积, 包括padding+ReLU
Matrix& conv(const Matrix & kernel, float * bias, size_t pad, size_t
stride);
//最大池化
Matrix& maxpooling();
//全连接
void fullCon(float* out, float* weightgh, float* bias, size_t N);
};

```

2.4 CNN实现

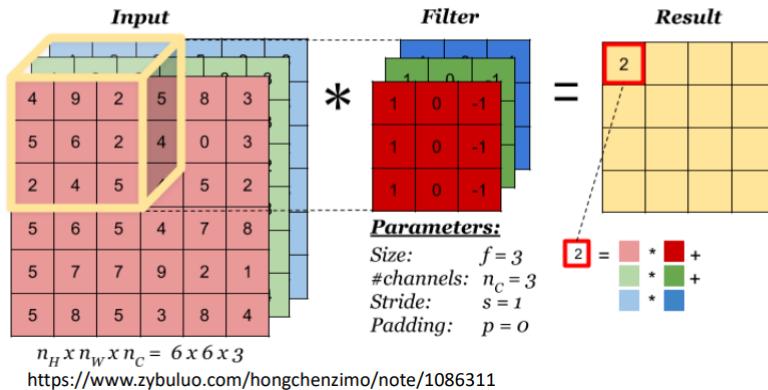
2.4.1 卷积

- 原理¹：

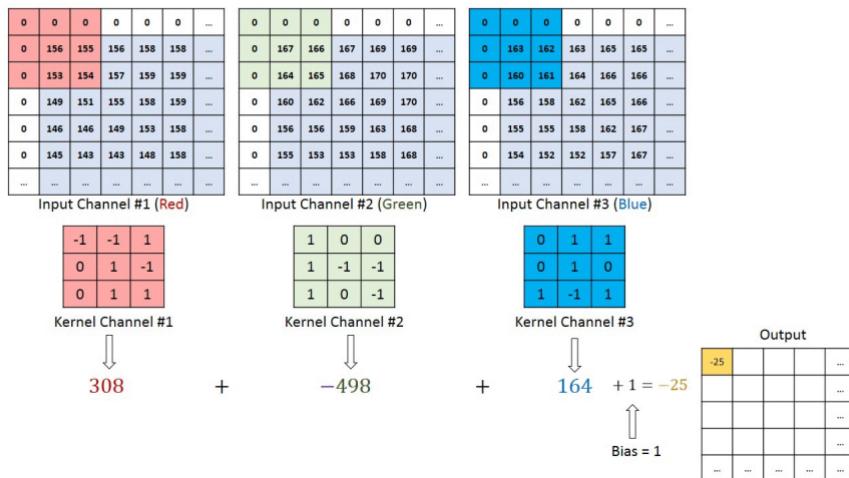
将卷积核（kernel）与输入图片矩阵对齐，对应矩阵位置的数值相乘后再相加，得到新矩阵，这样的过程就叫做卷积。



- Multiple filters (kernels) can create multiple output channels



在卷积之前，可以在输入图片矩阵的四周补上一些0，保证卷积处理完成后图像矩阵的空间尺度大小不变，便于后面的计算。



- 卷积（包含padding和ReLU）

参数	解释
const Matrix& kernel	卷积核，其通道数应与输入数据的通道数一致，卷积核的个数与输出数据的通道数一致
float * bias	偏斜度，长度与卷积核的个数相同
size_t pad	在输入每一层通道填充0的行列数
size_t stride	卷积核移动的步长

```

//conv
Matrix& Matrix::conv(const Matrix & kernel, float * bias, size_t pad, size_t
stride)
{
    if(channel != kernel.channel)
    {
        std::cerr << "conv failed for channel don't match!" << '\n';
        exit(0);
    }
    else
    {
        size_t out_channel = kernel.dimension;
        size_t out_height = (height + 2 * pad - kernel.height) / stride + 1;
        size_t out_width = (width + 2 * pad - kernel.width) / stride + 1;
        float * out_data = new float [1 * out_channel * out_height * out_width]
    }
}

Matrix* tmp = new Matrix(1 , out_channel , out_height , out_width ,
out_data);

//padding
size_t h = height + 2 * pad, w = width + 2 * pad;//padding后输入矩阵的高
= h, 宽=w
size_t hw = h * w;//padding后每一个channel的尺寸
size_t HW = height * width;//padding前每一个channel的尺寸
size_t kn = 0, kc = 0, ki = 0, kj = 0;
size_t KN = 0, KC = 0, KI = 0, KJ = 0;
float* inter = new float[dimension * channel * hw];

for (size_t n = 0; n < dimension; n++)
{
    kn = n * channel * hw;
    KN = n * channel * HW;

    // #pragma omp parallel for

    for (size_t c = 0; c < channel; c++)
    {
        kc = c * hw;
        KC = c * HW;
        for (size_t i = 0; i < height; i++)
        {
            ki = (i + pad) * w + pad;
            KI = i * width;
            kj = kn + kc + ki;
            KJ = KN + KC + KI;
            inter[ki * hw + kj] = 0;
        }
    }
}

```

```

        // #pragma omp parallel for

        for (size_t j = 0; j < width; j++) {
            inter[kj + j] = (*this).data[kj + j];
        }
    }
}

float sum = 0, res = 0;
size_t k_hw = kernel.height * kernel.width;
size_t k_chw = kernel.channel * k_hw;
size_t bc = 0, bi = 0, bj = 0, bik = 0, bck = 0, bjk = 0;//凡带k的，与
kernel有关

#if(BLAS)
float * t_sum = new float[k_hw];
size_t M = kernel.height, N = kernel.width, K = kernel.width;//MxK * KxN
float alpha = 1.0f, beta = 0.0f;
size_t lda = K, ldb = N, ldc = N;

for(size_t c = 0; c < out_channel; ++c)//输出channel
{
    for(size_t i = 0; i < out_height; ++i)//输出height
    {
        for(size_t j = 0; j < out_width; ++j)//输出width
        {
            cblas_sgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
                        M, N, K, alpha, inter,
                        lda, kernel.data, ldb, beta, t_sum, ldc);
            for (size_t i = 0; i < k_hw; ++i)
            {
                sum += t_sum[i];
            }
            // ReLU
            res = sum + bias[c];
            (*tmp).data[c * out_height * out_width + i * out_width + j]
= (res > 0.0f) ? res : 0.0f;
            sum = 0;
        }
    }
}
delete[] t_sum;

#else
// #pragma omp parallel for
for(size_t c = 0; c < out_channel; ++c)//输出channel
{
    for(size_t i = 0; i < out_height; ++i)//输出height
    {
        for(size_t j = 0; j < out_width; ++j)//输出width
        {
            //按着kernel的大小进行矩阵乘法，多用局部变量，减少计算量
            for(size_t k_c = 0; k_c < kernel.channel; ++k_c)//kernel的
channel
            {
                bc = k_c * hw;

```

```

        bck = k_c * kernel.height * kernel.width;
        for(size_t k_i = 0; k_i < kernel.height; ++k_i)//kernel的
height
    {
        //按stride移动kernel做不同的矩阵乘法
        bj = bc + k_i * w + stride * (i * w + j); //对应的是输
入矩阵
        bjk = bck + k_i * kernel.width + c * k_chw; //对应的是
kernel
        #if(CV_SIMD)
            cv::v_float32 v_sum = cv::vx_setzero_f32(); //向量元素
全部初始化为0
            size_t step = sizeof(cv::v_float32)/sizeof(float); //
计算每个向量可以存多少个float
            for (size_t i = step; i < kernel.width; i+=step)
            {
                cv::v_float32 v1 = cv::vx_load(inter + bj +
i); //从内存inter中装载bj+i个float到v1
                cv::v_float32 v2 = cv::vx_load(kernel.data + bjk
+ i); //从内存kernel.data中装载bjk+i个float到v2
                v_sum += v1 * v2; //通过库的内部重载的运算符，迅速算出
            }
            sum += cv::v_reduce_sum(v_sum);
        #else
            for(size_t k_j = 0; k_j < kernel.width;
++k_j)//kernel的width
            {
                sum += inter[bj + k_j] * kernel.data[bjk + k_j];
            }
        #endif
    }
}
// ReLU
res = sum + bias[c];
(*tmp).data[c * out_height * out_width + i * out_width + j]
= (res > 0.0f) ? res : 0.0f;
sum = 0;
}
}
#endif
delete[] inter;
return *tmp;
}
}

```

2.4.2 最大池化

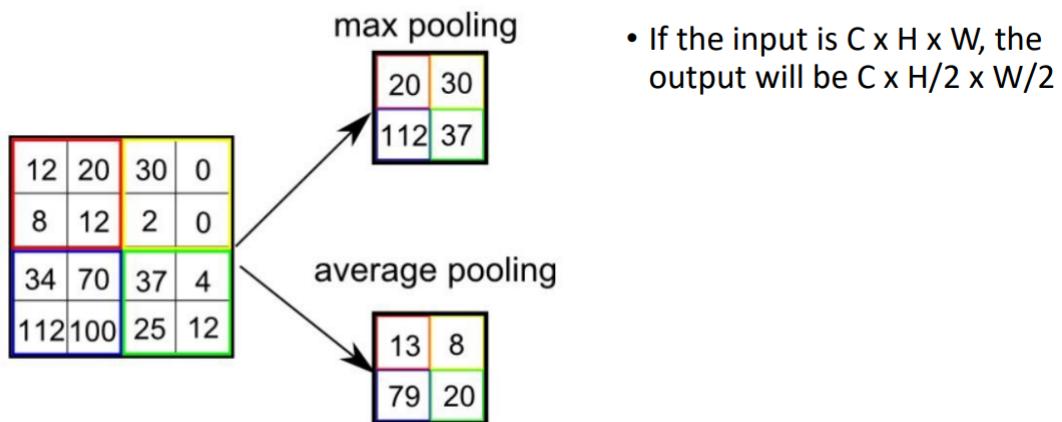
- 原理¹

每个通道，每个限定大小的 $h \times w$ 方格中取最大的数放到输出矩阵中（需保证 h, w 为偶数）。与之对应的还有average pooling。



max pooling

- For each channel



- 最大池化

参数	解释
无	无

```
//maxpooling
Matrix& Matrix::maxpooling()
{
    size_t N = dimension, C = channel, H = height / 2, W = width / 2;//四分矩阵
    float* inter = new float[N * C * H * W];
    Matrix* res = new Matrix(N, C, H, W, inter);
    float max = 0;
    float tmp1 = 0, tmp2 = 0, tmp3 = 0;

    // #pragma omp parallel for
    for (size_t n = 0; n < N; ++n)
    {
        for (size_t c = 0; c < C; ++c)
        {
            for (size_t i = 0; i < H; ++i)
            {
                for (size_t j = 0; j < W; ++j)
                {
                    //取四个格中最大的
                    max = (*this)(n, c, 2 * i, 2 * j);
                    tmp1 = (*this)(n, c, 2 * i, 2 * j + 1);
                    tmp2 = (*this)(n, c, 2 * i + 1, 2 * j);
                    tmp3 = (*this)(n, c, 2 * i + 1, 2 * j + 1);

                    if (tmp1 > max) max = tmp1;
                    if (tmp2 > max) max = tmp2;
                    if (tmp3 > max) max = tmp3;

                    (*res)(n, c, i, j) = max;
                }
            }
        }
    }
}
```

```

    }
    return *res;
}

```

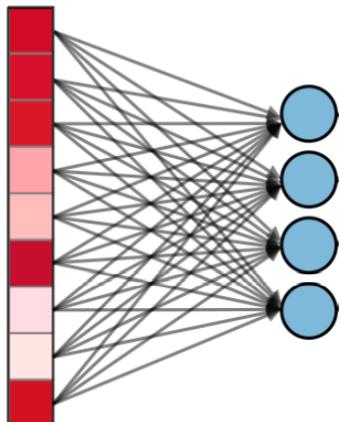
2.4.3 全连接

- 原理¹

矩阵和向量的相乘，按权重和偏斜度投射到结果上。



FC: Fully-Connected Layer



- If the input is L and the output is N (2 in the model, 4 in left figure), the size of weights is NxL (2x2048 in the model)

$$\text{output}_{2 \times 1} = \text{weight}_{2 \times 2048} * \text{input}_{2048 \times 1} + \text{bias}_{2 \times 1}$$

- 全连接

参数	解释
float* out	输出结果，由用户自己传入
float* weight	权重，老师提供
float* bias	偏斜度，老师提供
size_t N	输出数据的长度

```

//full connection
void Matrix::fullCon(float* out, float* weight, float* bias, size_t N)
{
    size_t L = dimension * channel * height * width;
    float sum;

    // #pragma omp parallel for
    for (size_t i = 0; i < N; ++i)
    {
        sum = 0;
        size_t L1 = L / 4, L2 = L / 2, L3 = 3 * L / 4;
        for (size_t j = 0; j < L1; ++j)
        {
            sum += weight[i * L + j] * data[j];
            sum += weight[i * L + L1 + j] * data[L1 + j];
            sum += weight[i * L + L2 + j] * data[L2 + j];
            sum += weight[i * L + L3 + j] * data[L3 + j];
        }
    }
}

```

```
    out[i] = sum + bias[i];
}
}
```

2.4.4 Softmax

- 原理¹

在本Project中保证了输出结果范围在 [0.0, 0.1]



Softmax

- To output the confidence vector ($n=2$ in the model)

for $x \in \mathbb{R}^n$

$$p_i = \frac{e^{x_i}}{\sum_{j=1}^n e^{x_j}} \quad p = \begin{pmatrix} p_1 \\ \vdots \\ p_n \end{pmatrix}$$

- Softmax

```
//Softmax
double p01 = exp(points[0]);
double p02 = exp(points[1]);
double p1 = p01 / (p01 + p02);
double p2 = p02 / (p01 + p02);
```

3 测试及分析

3.1 ×86_64

- 环境: ×86 Windows10的wsl
- 测试用例: 3个background, 7个face

测试用例	说明	Background	Face	用时(μs)	图片
bg.jpg	标准测试用例 background	1	3.55696e-07	22316	
face.jpg	标准测试用例 face	3.12513e-09	1	22693	
t1.png	某外国人脸	1.65381e-07	1	21890	
t2.jfif	于老师 GitHub 主页照片	2.69811e-09	1	22621	
t3.png	本人桌面背景	0.999947	5.34102e-05	21694	
t4.png	斜嘴狗头	0.0907577	0.909242	22134	
t5.png	依蕾娜侧脸	0.958185	0.0418151	22169	
t6.jpg	本人无眼镜正脸	2.13506e-11	1	23906	
t7.jpg	本人有眼镜正脸	0.000444412	0.999556	22541	
t8.jpg	本人有眼镜侧脸	0.000175216	0.999825	21623	
t9.jfif	某漂亮女孩照	0.000367365	0.999633	21814	
t10.jfif	《天气之子》海报	0.977267	0.0227326	22361	

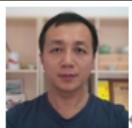
以上10个样例的测试用时都在 [27, 30](ms)之间（未作优化），对于真人 (face, t1, t2, t6, t7, t8, t9)、动漫人物 (t5，尽管在此处分数有些奇怪，详见 Part4 4.3）、抽象人脸 (t4)、背景 (bg, t3, t10) 的识别都有很好的效果。

奇怪的是，t5.png为某动漫女主人公的侧脸照被识别为背景像（即background得分远高于face得分），怀疑是占了图像四分之三的帽子和头发被识别为背景了（详见 Part4 4.3）。

对比t7.jpg和t8.jpg两幅图，区别在于带上了眼镜，但是最终都能识别出来，说明老师提供的kernel能忍受一定程度的干扰。（或许是把眼镜识别成了眼睛？）

3.2 ARM64

- 环境：ARM云服务器
- 测试用例：3个background，7个face，与上例相同

测试用例	说明	Background	Face	用时(μs)	图片
bg.jpg	标准测试用例 background	1	3.44121e-07	57157	
face.jpg	标准测试用例 face	2.32911e-09	1	57221	
t1.png	某外国人脸	2.62951e-07	1	57227	
t2.jfif	于老师 GitHub 主页照片	2.6867e-09	1	57307	
t3.png	本人桌面背景	0.999947	5.34102e-05	57349	
t4.png	斜嘴狗头	0.129898	0.870102	56990	
t5.png	依蕾娜侧脸	0.958185	0.0418151	57312	
t6.jpg	本人无眼镜正脸	2.19628e-11	1	57204	
t7.jpg	本人有眼镜正脸	0.000444412	0.999556	57219	
t8.jpg	本人有眼镜侧脸	0.000175216	0.999825	57238	
t9.jfif	某漂亮女孩照	0.00367365	0.999633	56960	
t10.jfif	《天气之子》海报	0.977267	0.0227326	57127	

与在x86的系统相比，ARM系统的用时稍微多些，达到了72ms，猜测是因为ARM服务器的内存远小于本机内存（ARM只有不到3GB的内存，如下图）。

```
[root@ecs001-0021-0022 proj5]# free -m
total        used        free      shared  buff/cache   available
Mem:       2977          350         611          13     2015          2275
Swap:          0           0           0
```

统计平均用时如下：

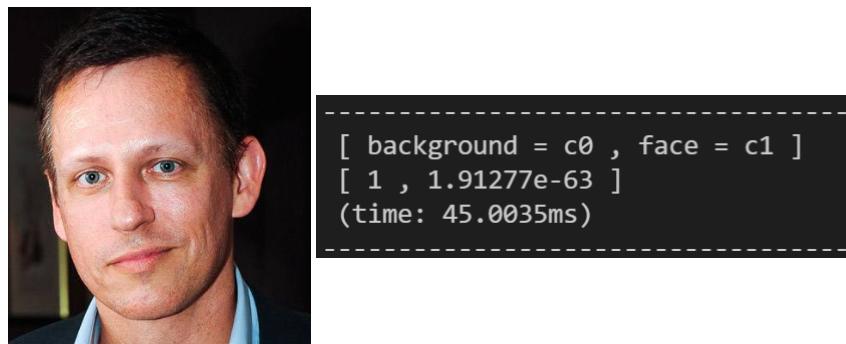
环境	无优化(μs)
x86_64	22326.64
ARM64	57181.73

尽管ARM上用时多于x86 (约61.0%)，但是在准确度方面无大差别，证明了本程序在跨平台运算的正确性。

4 困难及解决

4.1 读入图像的拉伸修正

最开始时，通过cv::Mat读入图像后，立刻进行了后续操作，导致出现了一些匪夷所思的结果。



对左边这幅人脸图像，竟然背景得分为1。但是对老师所给的标准测试样例face.jpg我却仍然有1人脸得分，bg.jpg也同样有1背景得分。

一开始，我怀疑是图片格式问题，因为老师提供的是.jpg格式，而我拿来测试的这个图片是.png格式。可惜由于在网络上难以找到答案，只找到一些其他的知识³，故该问题无法解决。

大小 8.6 KB	大小 221.3 KB
尺寸 128 x 128	尺寸 303 x 341

后来，我发现两幅图片有一个差异，也就是两幅图片的文件大小和图像尺寸是不一样的。由于不同图片的文件大小大部分都是不一样的，所以我怀疑可能是图像尺寸的问题。

◆ resize()

```
void cv::resize ( InputArray  src,
                  OutputArray dst,
                  Size        dsize,
                  double      fx = 0 ,
                  double      fy = 0 ,
                  int         interpolation = INTER_LINEAR
                )
```

```
[ background = c0 , face = c1 ]
[ 2.62951e-07 , 1 ]
(time: 61.2494ms)
```

显然，因为图像的尺寸会影响卷积和最大池化操作，尺寸是非常重要的。老师给的两个图片尺寸都是`128×128`。我在OpenCV的文档⁴里找到了能改变图像尺寸的函数（如左图所示），并将读入的图像立刻改成`128×128`的尺寸，得出了很漂亮的结果（如右图所示）。

4.2 BRG转RGB的优化

在Part1的读入图像处理里，实现了一次性多通道、多元素的批量处理，一定程度上提高了效率（虽然效果不是很明显）。

多通道+单元素：0.227405ms
多通道+多元素：0.177138ms

测试代码主要区别（左图多通道+单元素，右图多通道+多元素）：

```
//R
size_t l1 = i * col + j;
res[l1] = (float)p[3 * j + 2] / 255;
res[l1 + 1] = (float)p[3 * j + 5] / 255;
res[l1 + 2] = (float)p[3 * j + 8] / 255;
res[l1 + 3] = (float)p[3 * j + 11] / 255;

//G
size_t l2 = di + i * col + j;
res[l2] = (float)p[3 * j + 1] / 255;
res[l2 + 1] = (float)p[3 * j + 4] / 255;
res[l2 + 2] = (float)p[3 * j + 7] / 255;
res[l2 + 3] = (float)p[3 * j + 10] / 255;

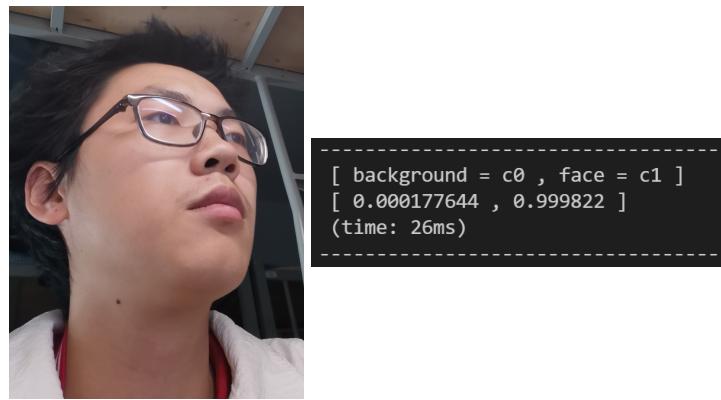
//B
size_t l3 = 2 * di + i * col + j;
res[l3] = (float)p[3 * j] / 255;
res[l3 + 1] = (float)p[3 * j + 3] / 255;
res[l3 + 2] = (float)p[3 * j + 6] / 255;
res[l3 + 3] = (float)p[3 * j + 9] / 255;
```

4.3 动漫人物头像被识别为背景

在测试时，发现有一幅动漫人物头像被识别成了背景（背景得分和脸部得分分别为0.9798和0.0201）（如下图，多次测试仍是如此）。

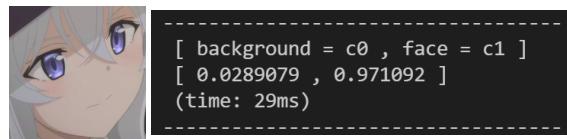


初步怀疑是老师提供的kernel不能识别侧脸，于是我测试了自己的戴眼镜侧脸照，结果如下（背景得分和脸部得分分别为0.0001和0.9998）。



显然，我最初的猜测是错误的，老师提供的kernel能用于识别侧脸。再回看动漫图，猜测是占据了图像四分之三的帽子和头发干扰了识别。

对该图像截图，重新测试，得到如下结果（背景得分和脸部得分分别为0.0289和0.9710）。果然！猜测正确，过多非脸部元素会极大地干扰人脸识别。



4.4 计时问题

一开始，我用来计时的代码如下。（实际上已经改过一次，舍去了小数位）

```

//...
//计时开始
auto start = std::chrono::steady_clock::now();
//...
//计时结束
auto end = std::chrono::steady_clock::now();
auto time = std::chrono::duration_cast<std::chrono::milliseconds>(end -
start).count();
//...

```

注意到计时的单位是毫秒 (milliseconds, ms)，这造成了计时时的严重误差。由于ARM服务器较为稳定，每次计时若用毫秒计，得出的用时是一样的，因此我换用了微秒 `std::chrono::microseconds` 计时，报告的 *Part3* 和 *Part5* 也重测了一遍，但是 *Part4* 由于是叙述困难，与计时关系不大，也就没有重测了。

5 优化

为简化测试，只取上面多个测试用例中的t2.jfif, t3.png, t4.png, t8.jpg, t10.jfif来测试。最终结果如下，详细内容分析可看下文。可以看到，O3优化对ARM效果更明显，而OpenMP对x86更明显。

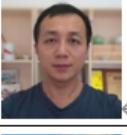
环境	x86_64 (μs)	ARM64 (μs)	提升率(x86_64)	提升率(ARM64)
无优化	22326.64	57181.73		
O3优化	5410.8	10587.4	75.8%	81.5%
OpenMP优化	21728	57129.8	2.7%	0.09%

5.1 O3加速

```
#pragma GCC optimize(3, "ofast", "inline")
```

效果非常惊人！

- x86_64

测试用例	说明	Background	Face	用时(μs)	图片
t2.jfif	于老师 GitHub 主页照片	2.81707e-09	1	5537	
t3.png	本人桌面背景	0.999882	0.000117993	5383	
t4.png	斜嘴狗头	0.0907578	0.909242	6048	
t8.jpg	本人有眼镜侧脸	0.000175216	0.999825	5025	
t10.jfif	《天气之子》海报	0.978251	0.0217494	5061	

- ARM64

测试用例	说明	Background	Face	用时(μs)	图片
t2.jfif	于老师 GitHub 主页照片	2.76405e-09	1	10619	
t3.png	本人桌面背景	0.999946	5.44457e-05	10579	
t4.png	斜嘴狗头	0.0907578	0.909242	10583	
t8.jpg	本人有眼镜侧脸	0.000175215	0.999825	10573	
t10.jfif	《天气之子》海报	0.978251	0.0217494	10583	

在误差允许范围内，保证准确性的情况下，两者的速度有很大提升，平均用时及提升率如下。对ARM的提升更多。

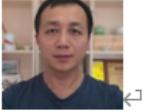
环境	无优化(μs)	O3优化(μs)	提升率
x86_64	22326.64	5410.8	75.8%
ARM64	57181.73	10587.4	81.5%

5.2 OpenMP并行优化

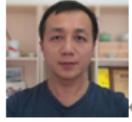
由于机器CPU核数有限，加入过多 `#pragma omp parallel for` 反而会降低效率，故需谨慎添加。

```
#include <omp.h>
//...
#pragma omp parallel for
//...
```

- x86_64

测试用例	说明	Background	Face	用时(μs)	图片
t2.jfif	于老师 GitHub 主页照片	2.81707e-09	1	22160	
t3.png	本人桌面背景	0.999882	0.000117993	21567	
t4.png	斜嘴狗头	0.0907577	0.909242	21749	
t8.jpg	本人有眼镜侧脸	0.000175216	0.999825	21699	
t10.jfif	《天气之子》海报	0.978251	0.0217494	21465	

- ARM64

测试用例	说明	Background	Face	用时(μs)	图片
t2.jfif	于老师 GitHub 主页照片	2.76403e-09	1	57161	
t3.png	本人桌面背景	0.999946	5.44459e-05	57226	
t4.png	斜嘴狗头	0.0907577	0.909242	56993	
t8.jpg	本人有眼镜侧脸	0.000133794	0.999866	57216	
t10.jfif	《天气之子》海报	0.977267	0.0227326	57053	

或许是我加 `#pragma omp parallel for` 的位置不太合适，也有可能是本机和云端服务器配置不高，无论本机还是服务器的增速效果都不明显。对x86提升更多

环境	无优化(μs)	OpenMP优化(μs)	提升率
x86_64	22326.64	21728	2.7%
ARM64	57181.73	57129.8	0.09%

5.3 局部变量优化

显然这个程序中有许多循环，若是能在循环中适当增加局部变量，能减少重复计算，尤其是乘法计算。（只截图了部分）

```
//按着kernel的大小进行矩阵乘法，多用局部变量，减少计算量
for(size_t k_c = 0; k_c < kernel.channel; ++k_c)//kernel的channel
{
    bc = k_c * hw;
    bck = k_c * kernel.height * kernel.width;
    for(size_t k_i = 0; k_i < kernel.height; ++k_i)//kernel的高度
    {
        //按stride移动kernel做不同的矩阵乘法
        bj = bc + k_i * w + stride * (i * w + j); //对应的是输入矩阵
        bjk = bck + k_i * kernel.width + c * k_chw; //对应的是kernel
```

5.4 OpenCV内置的universal intrinsics

受老师文章的启发^{5 6 7}，我尝试使用OpenCV的universal intrinsics来为 conv 加速。首先加入以下头文件并修改 conv 部分代码如下。

```
#include <opencv2/core/simd_intrinsics.hpp>
```

```

for(size_t k_i = 0; k_i < kernel.height; ++k_i)//kernel的高度
{
    //按stride移动kernel做不同的矩阵乘法
    bj = bc + k_i * w + stride * (i * w + j); //对应的是输入矩阵
    bjk = bck + k_i * kernel.width + c * k_chw; //对应的是kernel
#if(CV_SIMD)
    cv::v_float32 v_sum = cv::vx_setzero_f32(); //向量元素全部初始化为0
    size_t step = sizeof(cv::v_float32)/sizeof(float); //计算每个向量可以存多少个float
    for (size_t i = step; i < kernel.width; i+=step)
    {
        cv::v_float32 v1 = cv::vx_load(inter + bj + i); //从内存inter中装载bj+i个float到v1
        cv::v_float32 v2 = cv::vx_load(kernel.data + bjk + i); //从内存kernel.data中装载bjk+i个float到v2
        v_sum += v1 * v2; //通过库的内部重载的运算符，迅速算出
    }
    sum += cv::v_reduce_sum(v_sum);
#else
    for(size_t k_j = 0; k_j < kernel.width; ++k_j)//kernel的width
    {
        sum += inter[bj + k_j] * kernel.data[bjk + k_j];
    }
#endif

```

通过 #if(CV_SIMD) ... #else ... #endif 为用户提供更多的选择，当头文件中 #include <opencv2/core/simd_intrinsics.hpp> 后，就会自动进入 #if(CV_SIMD) 内，其中提供了利用 OpenCV 内置的 universal intrinsics 写的代码，用于加速向量乘法。

可惜的是，不知为何，每次计算出的 sum 都是0（曾在 sum += cv::v_reduce_sum(v_sum); 后作判断，若 sum 不为0，打印 sum 值，但是一直无打印），导致最终的计算结果错误至极。故该优化失败。

5.5 OpenBLAS 优化

```

#include <cblas.h>
#ifndef CBLAS_H //用于后面宏区别不同算法
#define BLAS 1
#else
#define BLAS 0
#endif

#if(BLAS)
    float * t_sum = new float[k_hw];
    size_t M = kernel.height, N = kernel.width, K = kernel.width; //MxK * KxN
    float alpha = 1.0f, beta = 0.0f;
    size_t lda = K, ldb = N, ldc = N;

    for(size_t c = 0; c < out_channel; ++c)//输出channel
    {
        for(size_t i = 0; i < out_height; ++i)//输出height
        {
            for(size_t j = 0; j < out_width; ++j)//输出width
            {
                cblas_sgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
                            M, N, K, alpha, inter,
                            lda, kernel.data, ldb, beta, t_sum, ldc);
                for (size_t i = 0; i < k_hw; ++i)
                {
                    sum += t_sum[i];
                }
                // ReLu
                res = sum + bias[c];
                (*tmp).data[c * out_height * out_width + i * out_width + j] = (res > 0.0f) ? res : 0.0f;
                sum = 0;
            }
        }
        delete[] t_sum;
    }
#endif

```

很可惜，对OpenBLAS的该函数使用不太熟悉，无法成功实现。

6 总结

本次Project是C/C++课程第5个Project，也是最后一个Project，总共的代码行数为321 (matrix.hpp) + 77 (main.cpp) + 41 (老师提供的face_binary_cls.cpp) = 398 + 41 (老师提供的face_binary_cls.cpp)，在一定程度上保持了代码的简洁。

这次Project要求实现一个简单的CNN，即卷积神经网络。虽然最初被这个名字所惊吓，但是一步步将每个功能实现，每次解决棘手的bug时，也非常有成就感。当最终程序能显示出正确的结果时，也能欢呼雀跃。这门课程不仅带给了我知识，也让我对计算机的底层更加好奇。

非常遗憾的是，有许多优化要不是实现代码了却没有成功（详见Part5），有的由于课业和时间原因来不及实现（如调用OpenBLAS，CUDA等），希望后续我能一一实现。

1. 于老师的上课课件 [回](#) [回](#) [回](#) [回](#) [回](#)
2. opencv中的resize()函数的理解以及引申 <https://www.cnblogs.com/zvmxvm1991/p/7891394.html> [回](#)
3. OpenCV imread读取jpg图像的一个大坑 <https://www.cnblogs.com/zjutzz/p/10543935.html> [回](#)
4. OpenCV文档 https://docs.opencv.org/4.x/da/d54/group__imgproc__transform.html#ga47a974309e9102f5f08231edc7e7529d [回](#)
5. 使用OpenCV中的universal intrinsics为算法提速 (1) https://mp.weixin.qq.com/s?_biz=MjM5NTE3NjY5MA%3D%3D&mid=2247484025&idx=1&sn=132d0fc0a242df11bd5b59cd22eaad99&scene=45#wechat_redirect [回](#)
6. 使用OpenCV中的universal intrinsics为算法提速 (2) https://mp.weixin.qq.com/s?_biz=MjM5NTE3NjY5MA%3D%3D&mid=2247484072&idx=1&sn=e04b079225776cfde7c400d319f58448&scene=45#wechat_redirect [回](#)
7. 使用OpenCV中的universal intrinsics为算法提速 (3) <https://mp.weixin.qq.com/s/XtV2ZUwDq8sZ8HlzGDRaWA> [回](#)