# CS205 C/ C++ Program Design - Project 5

## Background-facedetection

人脸识别技术是一项检测识别人脸进行身份识别的技术，它在现代社会中存在着广泛应用，这项技术已经在公安、金融、医疗、教育等诸多领域发挥了其独特作用。而人脸识别技术的成功与否，关键在于核心算法能否使其识别结果有实用化的识别率和识别速度。

人脸识别作为计算机研究领域的一项热点，其主要实现算法有两种主流，一是采用在LFW数据集上获取最优解的方式，一是通过深度学习的方法。一些商业公司和学者有实现一些辨识度和速度较好的人脸识别的算法和模型，无论是应用于一些实际应用领域，还是对于我们的学习和研究都有比较大的借鉴意义。对比起其他的算法来说，通过CNN实现人脸识别减少了许多图片预处理的工作。

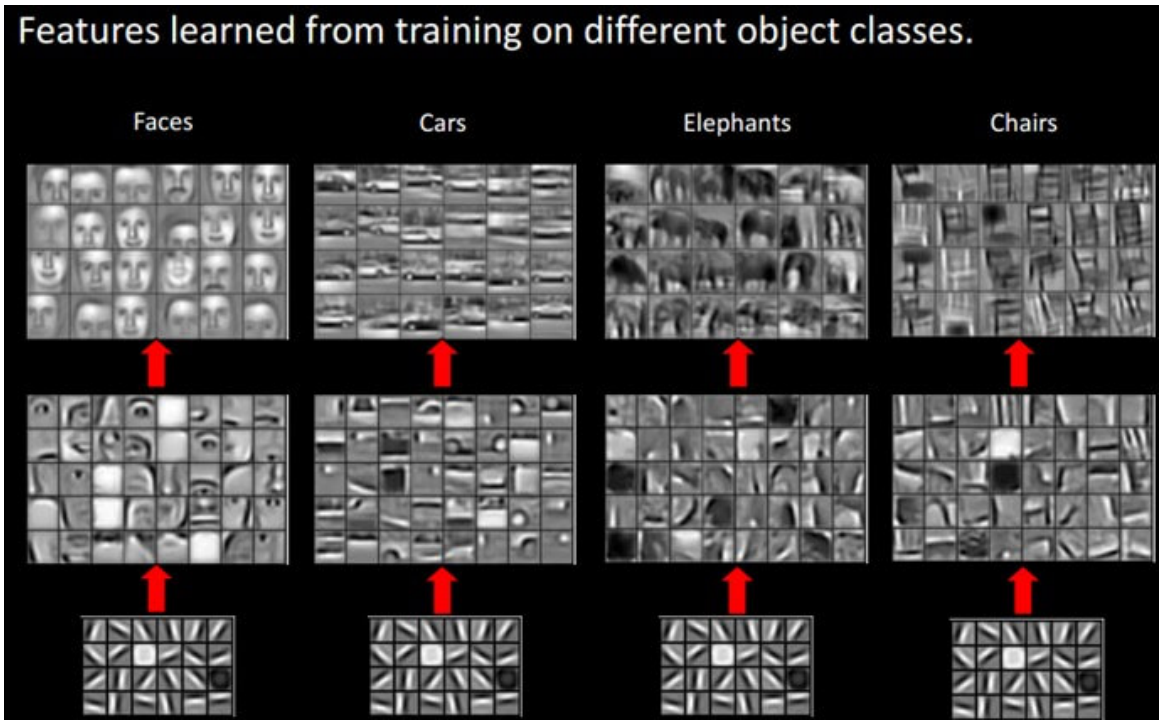
在初级阶段的学习中，此次project，基于卷积神经网络，我们通过三层卷积和一个全连接层实现一个简单的人脸识别的功能。

## SimpleCNN Model

### CNN（convolutional neural networks）概述

神经网络是人工智能研究的重要领域之一，而卷积神经网络（CNN）作为当下最流行的神经网络也被证实在如图像识别、语音识别等诸多领域上有着重要意义。CNN的出现，解决了大量数据量带来的效率低下的问题；以及其具有提取图像特征值的重要性质也解决了图像数据简单处理无法保留特征值的问题。

CNN是一种仿生模型，其灵感来源于人类视觉原理。人的大脑对于图像的处理是是通过逐级分层，不断提取特征，最终实现在最高层达到一个识别的效果。CNN算法就是基于此原理构建的。




（图片来源一文看懂卷积神经网络-CNN（基本原理+独特价值+实际应用）- 产品经理的人工智能学习库 (easyai.tech)）

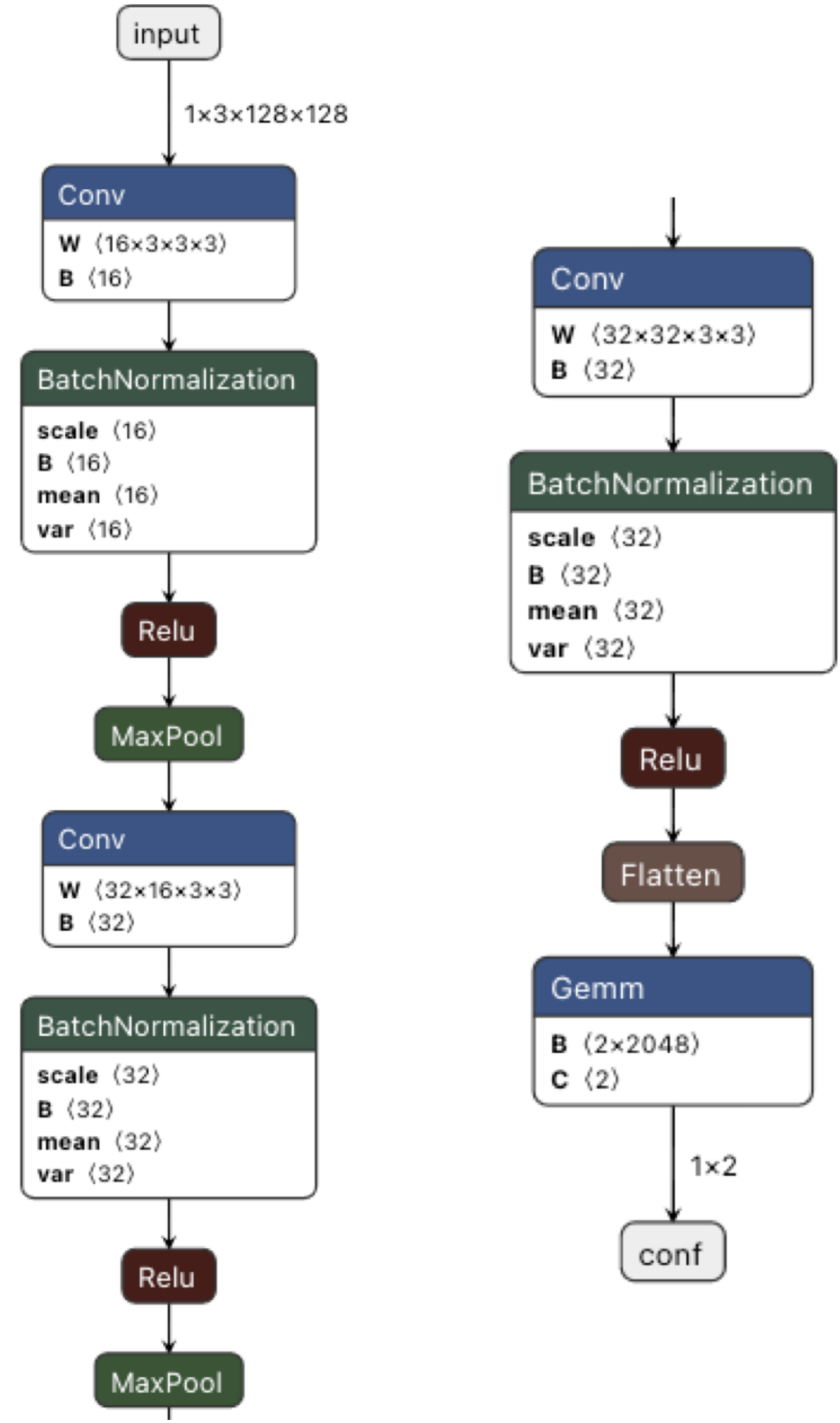
## Model

典型的CNN模型主要由三个部分组成，卷积层、池化层和全连接层组成。

在实现简单卷积神经网络之前，先来概述一下三个部分的主要功能和如何通过这三层达到人脸识别的目的。

卷积层是通过卷积来提取图像特征的，通过卷积核在原矩阵上移动进行点乘。池化层用作降采样，是一个非线性的过程，通过它可以降低数据的维度，从而能够提高全连接层运算的速度。

如下图所示，这是实现简单CNN模型的基本流程图：



该模型输入128 * 128的RGB图片，将其交替通过三层卷积和两个池化层，最后送入全连接层得到两个相加为1的正数，第一个代表该图片是背景的概率，第二个参数代表该图片为人脸的概率。

```
    cv::Mat A = cv::imread(argv[1]);
    extern conv_param conv_params[3];
    extern fc_param fc_params[1];
    matrices<float> mat0(128,128,3);
    Convert(A,mat0);
    matrices<float> mat1 = ConvBNReLU(conv_params[0], mat0);
    matrices<float> mat2 = MaxPool(2,mat1);
    matrices<float> mat3 = ConvBNReLU(conv_params[1],mat2);
    matrices<float> mat4 = MaxPool(2,mat3);
    matrices<float> mat5 = ConvBNReLU(conv_params[2],mat4);
    matrices<float> mat6 = FullConnectSoftMax(fc_params[0],mat5);
```

该代码是main方法中的主要流程，先通过opencv将图片读入到cv::Mat中,首先通过`Convert()`函数将cv::Mat类转化为自己定义的matrices类，然后依次调用`ConBNRelu()`,`MaxPool()`,`FullConnectSoftMax()`函数，最后得到的mat6所包含的数据就是两个与背景和人脸相似的概率。

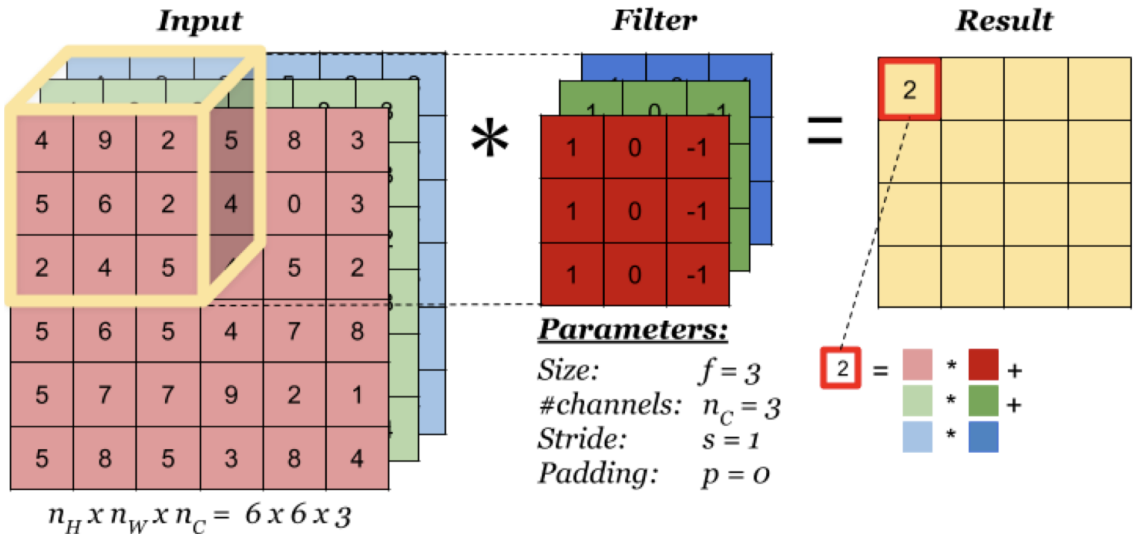
## Convert

该函数用作cv::Mat到自定义的matrices类的转化，由于对于该模型对于图片的大小有严格的限制故该功能的实现还是比较局限的，主要就是将图片像素cv::Mat的存储方式[b1g1r1b2g2r2...]转化为matrices中像素数据的存储方式[r1r2r3...g1g2g3...b1b2b3...]。

```
void Convert(cv::Mat &A,matrices<T> &mat){
    int size = 128*128*3;
    for (int j = 2; j >=0; --j)
    {
        int j_index_mat = 128*128*j;
        for (int i = 2-j, i_index = 0; i < size; i += 3, i_index++)
        {
            mat.data[j_index_mat + i_index] = A.data[i];
        }
    }
    for (int i = 0; i < 3 * 128 * 128; i++)
    {
        mat.data[i] = mat.data[i] / 255.f;
    }
}
```

## ConvBNRelu

该函数主要实现了卷积和激活函数的效果。在卷积部分，最开始先按照原始的数学运算的逻辑实现了卷积核卷积的过程，其主要思路如图所示，

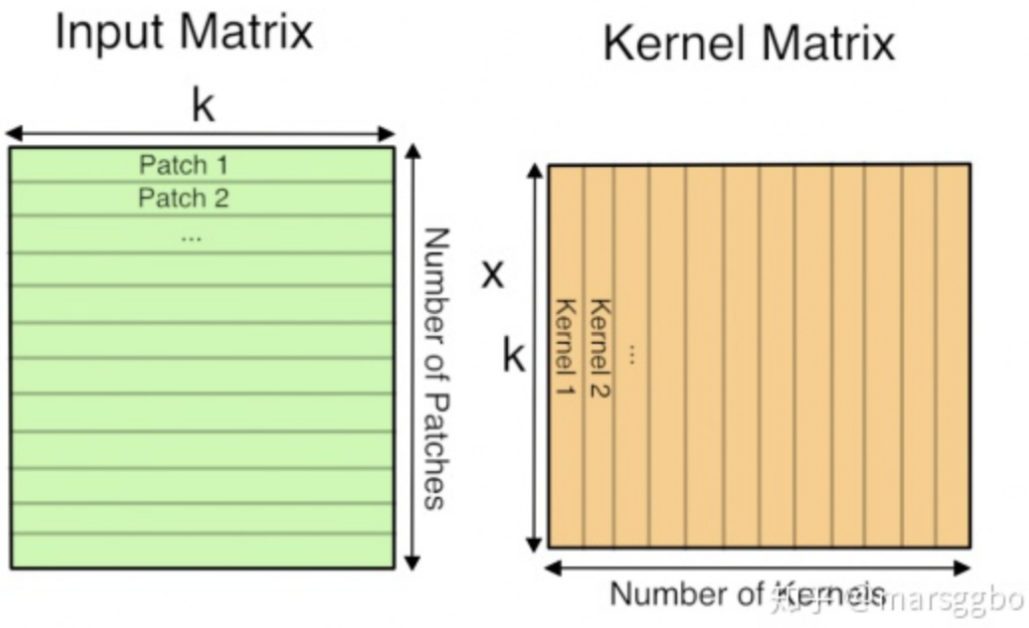



pad代表补零，若pad = 1就在原矩阵的四周补一圈0,stride表示平移卷积核时的步长；对于特定位置上的卷积结果是由对应的卷积核所移动到的所在位置上，卷积核的权重与对应的数据相乘。用代码的方式初步数学化地实现出来是这样的：

```
    //version1
        for (int k = 0; k < channel_in; k++)
        {
            for (int i = -pad, i_index = 0; i <= col - kernel_size + pad; i +=
stride, i_index++)
            {
                for (int j = -pad, j_index = 0; j <= row - kernel_size + pad;
j += stride, j_index++)
                {
                    //产生一个（i,j,k）对应的矩阵所对应的16个值
                    for (int chan = 0; chan < channel_out; chan++)
                    {
                        //产生一个（i,j,k）对应的矩阵与所对应一个out_channel的值
                        for (int u = 0; u < kernel_size; u++)
                        {
                            for (int v = 0; v < kernel_size; v++)
                            {
                                //相对应的矩阵进行点乘
                                if (i + u < 0 || i + u >= row || j + v < 0 ||
j + v >= row)

                                {
                                }
                                else
                                {
```

```
                                    result.data[chan * row_out * col_out +
i_index * col_out + j_index] += mat.data[k * col * row + (i + u) * col + (j +
v)] * param.p_weight[chan * channel_in * kernel_size * kernel_size + k *
kernel_size * kernel_size + u * kernel_size + v];
                                }
                            }
                        }
                    }
                }
            }
        }
    for (int k = 0; k < channel_out; k++)
     {
         float bias = param.p_bias[k];
         int k_index_bais = k * col_out * row_out;
         for (int i = 0; i < col_out * row_out; i++)
         {
             result.data[k_index_bais + i] += bias;
         }
     }
```

但这样的实现方式对应的是内存空间访问的严重不连续性，加上乘法本身的耗时性，会导致代码运行效率低下。为提高性能，增加缓存命中率，提高运算速度，采用将卷积转化为矩阵乘法的形式进行运算，可通过参考下方的示意图：




（图片来源：https:// petewarden.com/2015/04/ 20/why-gemm-is-at-the-heart-of-deep-learning）

该实现方法就是对输入矩阵进行内部的重排列，再通过矩阵乘法的方式实现卷积的效果。其中patch表示的是每个卷积核在移动的过程中与原矩阵对应的数据（其大小为channel_in * kernel_w * kernel_h）,将其展开拼成inputmatrix的每一行，同理对于卷积核做类似操作，对于每一个卷积核，将其展开为kernel matrix的每一列。

虽然这个重新拼凑矩阵时也不可避免内存访问的不连续性，但其不需要同时进行运算速率较慢的乘法运算，故其效率会优于数学逻辑上的简单实现。经过测试，转化为矩阵乘法的速率会比原数学逻辑上的简单实现的快一倍。

```
 // the convolution part
    //version2
matrices<T> kernel(channel_out,channel_in*kernel_size*kernel_size,1);
kernel.data = param.p_weight;
++kernel.refcount;
int H_data = channel_in*kernel_size*kernel_size;
int W_data = row_out*col_out;
matrices<T> matrix(H_data,W_data,1);

    for(int i = -pad,i_index = 0;i <= col-kernel_size+pad;i +=
stride,i_index++){
        for(int j = -pad,j_index = 0;j <= row-kernel_size+pad;j +=
stride,j_index++){
            int i_index_matrix = i_index*col_out+j_index;
            for(int k = 0;k < channel_in;k++){
                int k_index_matrix = k*kernel_size*kernel_size;
                int k_index_mat = k*row*col;
                for(int u = 0;u < kernel_size;u++){
                    int u_index_matrix = u*kernel_size;
                    int u_index_mat = (i+u)*col;
                    for(int v = 0;v < kernel_size;v++){
                    if(i + u < 0 || i + u >= row || j + v < 0 || j + v >= row)
{

                    }
                    else{

 matrix.data[(k_index_matrix+u_index_matrix+v)*W_data+i_index_matrix] =
mat.data[k_index_mat+u_index_mat+(j+v)];
                    }
                    }
                }

            }
        }
    }

result = kernel*matrix;
result.setRow(row_out);
result.setCol(col_out);
result.setChannel(channel_out);
```

```
    for (int k = 0; k < channel_out; k++)
    {
        float bias = param.p_bias[k];
        int k_index_bais = k * col_out * row_out;
        for (int i = 0; i < col_out * row_out; i++)
        {
            result.data[k_index_bais + i] += bias;
        }
    }
```

在实现ReLU时，尝试使用两种方式，一是使用std::max的方式实现数据data和0之间的大小比较。

二是采用三目运算符的方式来实现。由于是float数据类型之间的比较，位运算需要为类型转化做额外的开销，故此处并没有实现位运算的方式。

但经过测试发现，数据时间不太稳定，因为使用了clock()函数，这可能和CPU调度任务在不同时刻存在不同有关。但总体而言两种方式的时间开销差距不大。

## MaxPool

该函数主要实现的是降采样的功能，当要降低四分之一的是采样率时，即对每个channel的数据中对大小为4的矩阵中保留一个最大的数，若根据简单数学逻辑进行实现的话，改代码仍然存在大量内存访问不连续的情形，同样，为提高缓存的命中率，采用了先在行方向上进行降维操作然后再在列方向上进行降维操作的方法，虽然该多次循环也会带来多余的开销，但折中考虑，仍然采用了这种方法进行实现。

```
template <class T>
matrices<T> MaxPool(int downsampling, matrices<T> &mat)
{
    // check the arguments
    if (downsampling < 0 || downsampling == 0)
    {
        fprintf(stderr, "downsampling error:negative or zero! %s(%d)-%s\n",
__FILE__, __LINE__, __FUNCTION__);
    }
    if (mat.getCol() % downsampling != 0 || mat.getRow() % downsampling != 0)
    {
        fprintf(stderr, "downsampling error:size do not fit! %s(%d)-%s\n",
__FILE__, __LINE__, __FUNCTION__);
    }
    if (mat.data == NULL)
    {
        fprintf(stderr, "mat_data error:NULL! %s(%d)-%s\n", __FILE__,
__LINE__, __FUNCTION__);
    }
    if (mat.getChannel() == 0)
    {
        fprintf(stderr, "mat_channel error:zero! %s(%d)-%s\n", __FILE__,
__LINE__, __FUNCTION__);
```

```
    }
    if (mat.getRow() == 0)
    {
        fprintf(stderr, "mat_row error:zero! %s(%d)-%s\n", __FILE__, __LINE__,
__FUNCTION__);
    }
    if (mat.getCol() == 0)
    {
        fprintf(stderr, "mat_col error:zero! %s(%d)-%s\n", __FILE__, __LINE__,
__FUNCTION__);
    }
    // some parameters
    int channel = mat.getChannel();
    int row = mat.getRow();
    int col = mat.getCol();
    int row_out = row / downsampling;
    int col_out = col / downsampling;
    matrices<T> result(row_out, col_out, channel);
    matrices<T> temp(row_out, col, channel);

    for (int k = 0; k < channel; ++k)
    {
        int k_index_temp = k * row_out * col;
        int k_index_mat = k * row * col;
        for (int i = 0, i_index = 0; i < row; i += downsampling, ++i_index)
        {
            int i_index_temp = i_index * col;
            int i_index_mat = i * col;
            for (int j = 0; j < col; j += downsampling)
            {
                temp.data[k_index_temp + i_index_temp + j] =
std::max(mat.data[k_index_mat + i_index_mat + j], mat.data[k_index_mat +
i_index_mat + j + 1]);
            }
        }
    }

    for (int k = 0; k < channel; ++k)
    {
        int k_index_temp = k * row_out * col;
        int k_index_mat = k * row * col;
        for (int i = 1, i_index = 0; i < row; i += downsampling, ++i_index)
        {
            int i_index_temp = i_index * col;
            int i_index_mat = i * col;
            for (int j = 0; j < col; j += downsampling)
            {
```

```
                temp.data[k_index_temp + i_index_temp + j + 1] =
std::max(mat.data[k_index_mat + i_index_mat + j], mat.data[k_index_mat +
i_index_mat + j + 1]);
            }
        }
    }

    for (int k = 0; k < channel; ++k)
    {
        int k_index_result = k * row_out * col_out;
        int k_index_temp = k * row_out * col;
        for (int i = 0; i < row_out; ++i)
        {
            int i_index_result = i * col_out;
            int i_index_temp = i * col;
            for (int j = 0, j_index = 0; j < col; j += downsampling,
++j_index)
            {
                result.data[k_index_result + i_index_result + j_index] =
std::max(temp.data[k_index_temp + i_index_temp + j], temp.data[k_index_temp +
i_index_temp + j + 1]);
            }
        }
    }
    return result;
}
```

## FullConnectSoftMax

该函数主要实现的矩阵乘法，达到全连接以及归一化的效果，由于该过程较为简单，此处不做过多赘述。使用SIMD的硬件优化提速矩阵乘法。

```
template <class T>
matrices<T> FullConnectSoftMax(fc_param param, matrices<T> &mat)
{
    // check the paramter!
    if (param.in_features == 0)
    {
        fprintf(stderr, "fc_param_in_feature error:zero! %s(%d)-%s\n",
__FILE__, __LINE__, __FUNCTION__);
    }
    if (param.out_features == 0)
    {
        fprintf(stderr, "fc_param_out_feature error:zero! %s(%d)-%s\n",
__FILE__, __LINE__, __FUNCTION__);
    }
    if (mat.data == NULL)
    {
```

```
        fprintf(stderr, "mat_data error:NULL! %s(%d)-%s\n", __FILE__,
__LINE__, __FUNCTION__);
    }
    if (mat.getChannel() == 0)
    {
        fprintf(stderr, "mat_channel error:zero! %s(%d)-%s\n", __FILE__,
__LINE__, __FUNCTION__);
    }
    if (mat.getRow() == 0)
    {
        fprintf(stderr, "mat_row error:zero! %s(%d)-%s\n", __FILE__, __LINE__,
__FUNCTION__);
    }
    if (mat.getCol() == 0)
    {
        fprintf(stderr, "mat_col error:zero! %s(%d)-%s\n", __FILE__, __LINE__,
__FUNCTION__);
    }

    matrices<T> result(param.out_features,1,1);
    int out_features = param.out_features;
    int in_features = param.in_features;
    int len = in_features /4;
    T buffer[4];
    float expsum = 0;
    for(int i = 0; i < out_features;++i){
        float temp = 0;
        int i_index = i * in_features;
        for(int j = 0; j < len; ++j){
        __m128 m1 =
_mm_set_ps(param.p_weight[i_index+j],param.p_weight[i_index+j+1],param.p_weigh
t[i_index+j+2],param.p_weight[i_index+j+3]);
        __m128 m2 =
_mm_set_ps(mat.data[j],mat.data[j+1],mat.data[j+2],mat.data[j+3]);
        __m128 result = _mm_mul_ps(m1, m2);
        _mm_store_ps(buffer, result);
         temp += buffer[0] + buffer[1] + buffer[2] + buffer[3];
        }
        result.data[i] = temp + param.p_bias[i];
        expsum += exp(result.data[i]);
    }    matrices<T> result(param.out_features, 1, 1);
    float expsum = 0;
    int out_features = param.out_features;
    int in_features = param.in_features;
    for (int i = 0; i < out_features; i++)
    {
        int i_index = i * in_features;
        float temp = 0;
        for (int j = 0; j < in_features; j++)
```

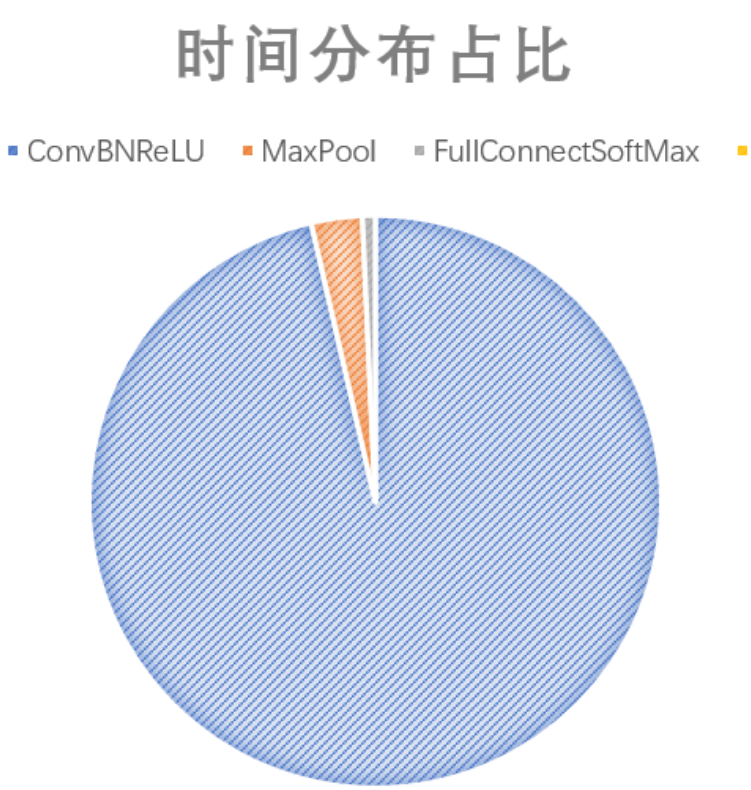
```
        {
            temp += param.p_weight[i_index + j] * mat.data[j];
        }
        result.data[i] = temp + param.p_bias[i];
        expsum += exp(result.data[i]);
    }

    for (int i = 0; i < param.out_features; i++)
    {
        result.data[i] = exp(result.data[i]) / expsum;
    }
    return result;
}
```

# Test and Evaluation

## 算法识别速度

通过测试一些图片，发现该算法可以在20~60ms之间完成运算，输出检测结果。

测试代码所得出的CNN所消耗时间分布图如下所示：




从图中可以看出，对于128*128的RGB图像来说，耗时占比最大的是卷积部分，约占据总时间的95%。故若对此算法进行性能上的优化，可考虑提高卷积层的运算速度。

## Vs openBLAS

在CNN中通过记录调用不同函数所消耗的时间可以得出，卷积部分是耗时最久的，由于在该算法中是通过使用矩阵乘法来实现卷积层，故使用开源OpenBLAS来加速矩阵乘法，从而比较openBLAS与自身算法的差别：

对输入同一张图片进行多次测试，得到以下数值。

| Function | Time |
| --- | --- |
| OpenBLAS | 9412us |
| 未使用openBLAS加速前 | 48461us |

说明使用openBLAS加速使得该性能提高五倍左右。

## 算法识别正确率

在测试算法结果时，采用测试样例进行调试的结果如下：

| 图片 | 像人脸的概率 | 像背景的概率 |
| --- | --- | --- |
| face.jpg | 1 | 3.76485e-09 |
| bg.jpg | 3.55695e-07 | 1 |

为更好地测试算法的可靠性，我增加了其他图片的测试样例，这些图片可以大致分成三类，一类是证件照类型图片，一类是动漫人物的图片，一类是普通的生活照。发现算法对于动漫人物图片识别效果较好，对于证件照类型的图片识别效果不稳定，和画面的色彩有一定关系。对于普通的生活照的识别效果不太好。

产生该结果的原因可能是因为卷积的深度不够等导致的，可以尝试增加卷积的次数，检测识别的效果有无比较明显的变化。但是由于缺少训练参数，此处并未实现多层卷积。

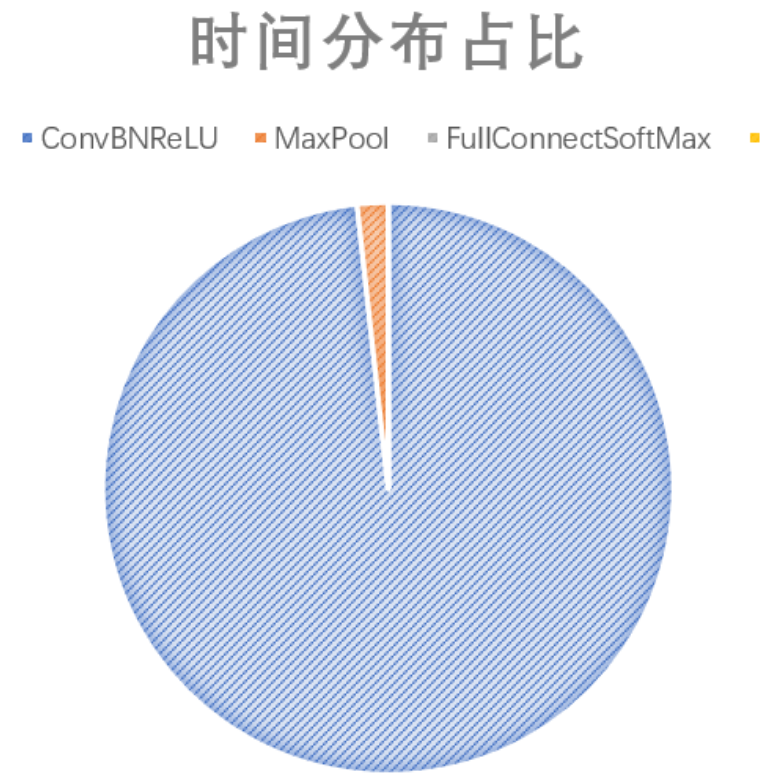
## Comparison of different Platform in X86 and ARM

在ARM系统上运行程序，使用测试样例进行检测，发现算法识别结果并没有发生变化。

但是在时间的比对上可以参考如下的表格：

| Platform | Time |
| --- | --- |
| X86 | 48.461ms |
| ARM | 59.162ms |

对于在ARM平台上程序运行内部层之间的时间开销占比也相应地制作表格：




对比起在X86平台上在FullConnectSoftMax函数时间调用具有优势，但是总体而言，在ARM平台上运行的卷积层的用度开销会大于在X86平台上，且其占比提高到98%。

这些计算上的差异，体现了ARM处理器和X86处理器在性能上的差别，X86处理器专注于性能和速度，而ARM处理器则以其精简计算机指令，在一些硬件路线上可以达到一个比较好的效果。其中具体的差别在Project4之中已经做了详细的介绍，故此处也不多赘述。

# Reference

**1.基于深度学习的人脸识别技术综述 - 知乎 (zhihu.com)**

**2.一文看懂卷积神经网络-CNN（基本原理+独特价值+实际应用）- 产品经理的人工智能学习库 (easyai.tech)**