CS205 C/C++ Program Design – Project 2.

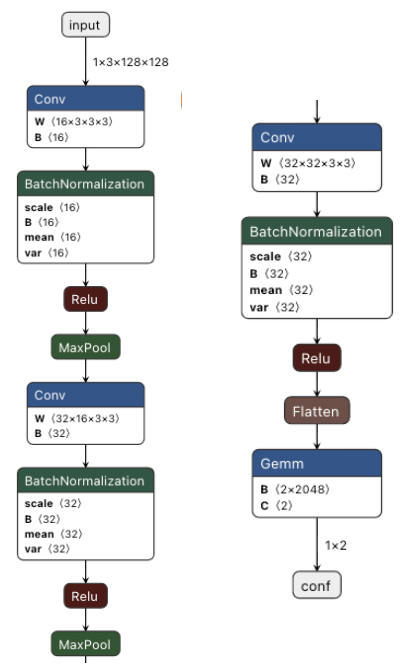
11912306 董叔文

一、目标：实现卷积神经网络

二、实现：

1. 实现原理：

首先，通过于老师的讲解，我明确了实现的具体内容，由三个卷积层（再加三个标准化层，但是已经被老师整合到参数中了），三个激活函数层，三个池化层，以及一个展平面层，一个全连接层。



其中卷积层的作用在于处理信息，找到图片的特征；标准化的作用在于修正数据的均值和方差，使其处于同一特定空间中，方便接下来的处理；激活函数的意义在于破坏数据的线性性质，解决线性不可分问题，同时，该函数也能起到稀疏化的作用，减少计算量；池化的作用也可以对数据进行压缩，减少计算量；展平面层的作用在于将矩阵的多维信息一维化，作为从卷积层到全连接层的过度；最后的全连接层由神经元组成（本模型只有一层神经元），用于判断图片的特征。

1. 各层具体实现
2. 数据读入

为了将图片以矩阵的形式读入到内存中，我使用了OpenCV的Mat类进行读入。



由于本模型是由3\*128\*128的图片学习而来，只能判断该大小图片，因此，当图片的规模不等于128\*128时（读入的通道都是3），便将其进行拉伸或者压缩

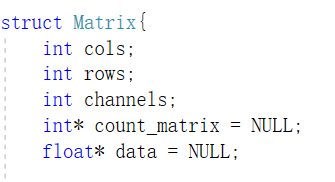


以上是我的代码中所有用到的OpenCV中有关图片处理库函数。

接下来，由于OpenCV默认的读入方式为CV\_8UC3，即三通道，无符号字符，且储存方式为BGR。因此，我首先将其转换到了三通道，有符号浮点数（范围0.0f~1.0f）



该函数以一个Mat类为输入，输出为我的自定矩阵Matrix

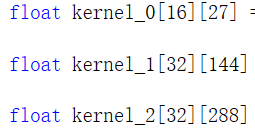


其中包含参数有矩阵的行数、列数、通道数、数据的引用次数（当不同的Matrix类使用相同的数据时，data会指向同一内存以节约空间，他们的count\_matrix会指向同一个整形数），以及一个指向元素内存地址的指针。除外，还有配套的拷贝函数、建构函数、析构函数、重载运算符=等等主要用来支持count\_matrix。

ConvertToFloat函数将会将Mat类中的数据转换为float并除以255以保证其范围在（0，1）之间，从BGR重排为RGB，然后将其保存到输出矩阵中。为了与Mat类对应，我将data的存储格式定为RGBRGBRGB...而不是RRRRR...BBBBB...GGGGG...的形式，即每个元素都由通道个值一同构成。

由此，我便得到了一个1\*128\*128（float）且范围在（0,1），RGB格式的输入数据。

1. 参数调整

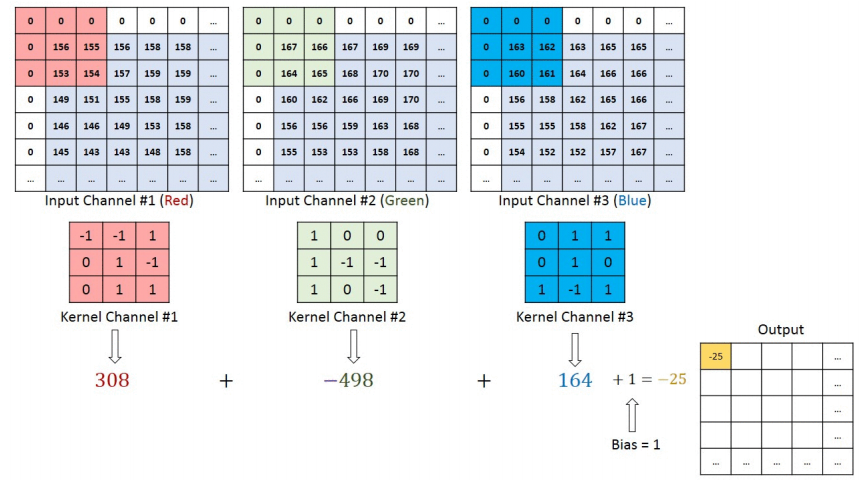


由于我采用的数据储存方式为RGBRGBRGB...，老师给出的参数储存方式为RRRRR...BBBBB...GGGGG...，因此我事先将参数调整为RGBRGBRGB...形式，保存起来。

1. 卷积函数



卷积层将实现最朴素的卷积操作。原理为将参数矩阵与输入矩阵从最左上角开始相乘（从左到右，从上到下，每次移动步长个单位），进行最简单的每对相应元素的数乘再与偏置一同求和，作为输出矩阵的下一个元素的值（从上到下，从左到右）。当然，经过查阅资料，我发现最正规的卷积应当是将参数函数上下左右颠倒后在进行操作，但是这只是定义的操作，不进行颠倒（学习模型中便不进行颠倒）反而更加简便。



为了保持输出矩阵的大小与输入矩阵的大小一致，还需要引入padding操作，具体操作为将原数据的外层加上几层0，在本模型中，由于参数矩阵为3\*3\*3矩阵，因此只要加上一周0即可。由于是中间操作，因此我并未将其模块化输出一个Matrix，而是输出了最简单易拷贝的浮点型指针，减少参数调用中的拷贝时间。



在卷积的具体操作中，我直接抓取了输入矩阵中的相应元素，排成一维向量，和参数矩阵的一维数据向量进行点乘操作。





然后加上偏置，放到计算出的输出矩阵的相应位置即可。

1. 激活函数



该操作十分简单，根据我们采用的函数y = x (x>=0), y = 0 (x<0)，遍历矩阵元素，将小于0的部分设为0，剩余部分保持不变即可。

1. 池化



该操作与卷积操作十分相似，每次取输入数据中的特定区域，然后找到最大值。但是由于我的存储方式为RGBRGBRGB...，因此我必须计算出相应层的位置，遍历该层的特定区域，找到最大值，放入输出矩阵相应位置的相应通道即可。



1. 展平

该操作在本模型中不需要额外的操作，因为我的数据储存方式本身即为一维向量。

1. 全连接



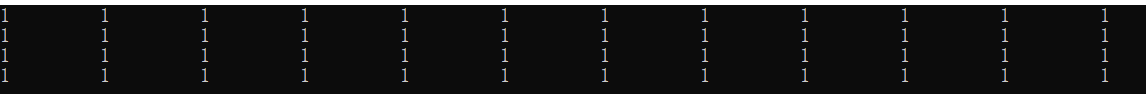
本层的主要实现是让数据通过神经元，不同的神经元对应不同的输入（本模型只有两个神经元，分别用于判断人脸的可能性已经非人脸的可能性）、偏置，不同的输入对应不同的权重。将每一个神经元的输入与其权重相乘，加和（包括偏置），便能得到输出，同样为简单的点乘操作。最后为了标准化输出，对神经元的输出进行softmax,得到了人脸和非人脸的几率。

1. 结果
2. 实现项目要求

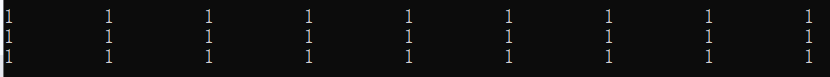
1、(20 points) The convolutional layer for 3x3 kernels is correctly implemented. It should

support stride=1 and stride=2 as well as padding=1.

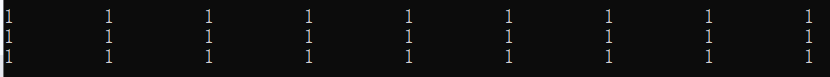
用测试数据进行正确性验证



（测试用矩阵M）



（参数矩阵1）



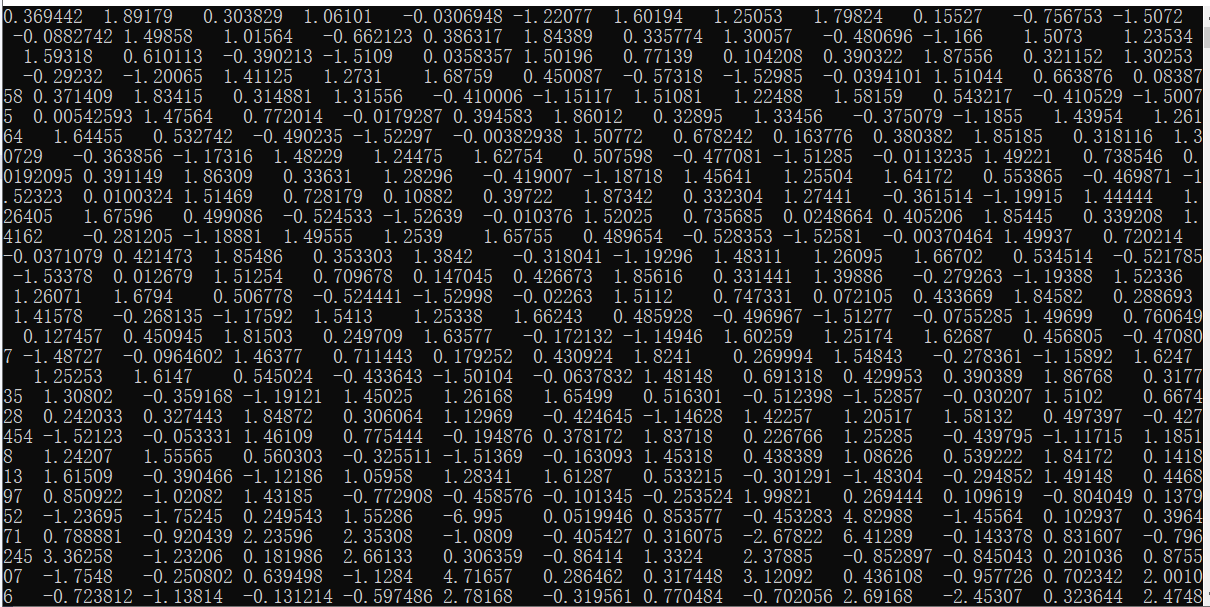
（参数矩阵2）



（偏置1,2）



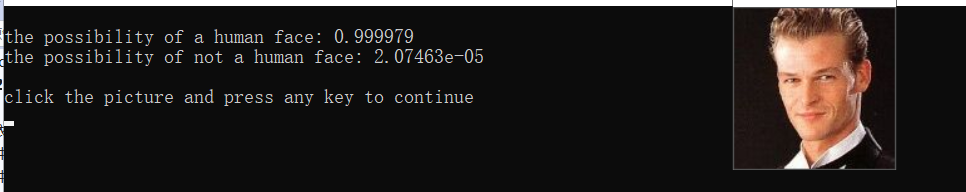
（卷积结果，正确）

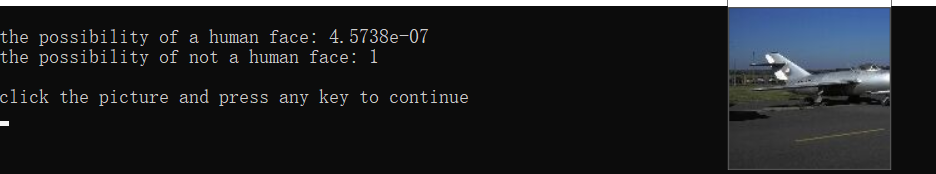


(人脸样例的第一次卷积矩阵的开始部分结果)

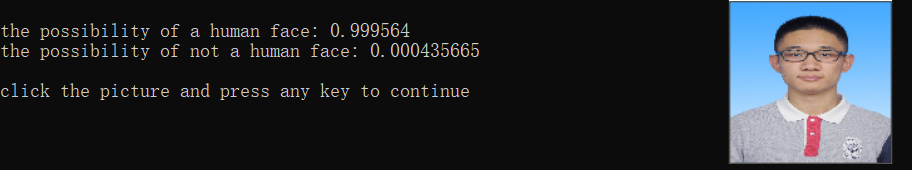
2. (30 points) The program can output the confidence scores correctly for images. You can

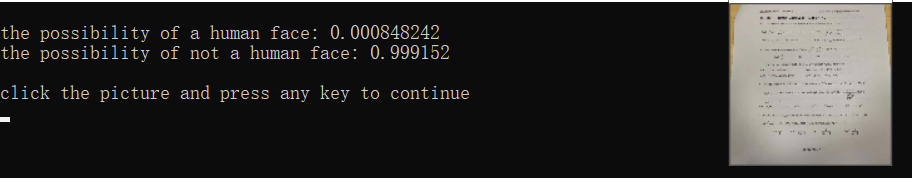
take the sample images at SimpleCNNbyCPP web site to test.





结果与样例标准输出几乎无差距。可能造成偏差的原因：为了调整参数，我将其依次从文本读入又输出到文本，在该过程中程序自动优化精度导致微小的损失



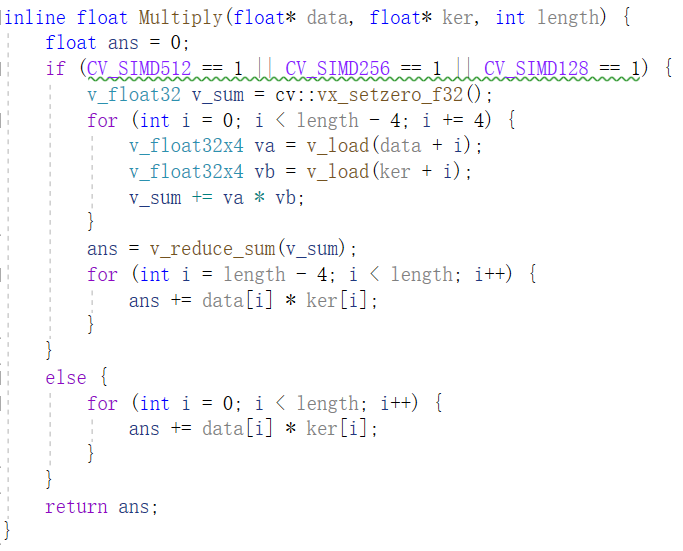


其他图片结果也是正确的

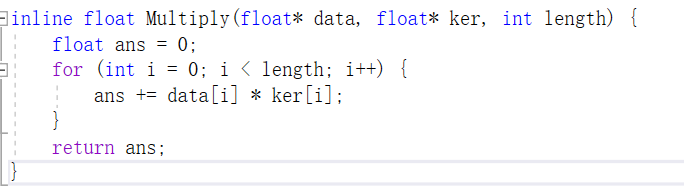
3. (20 points) Optimize your implementation and introduce it in your report. Some

comparisons, analysis and conclusions are welcome.

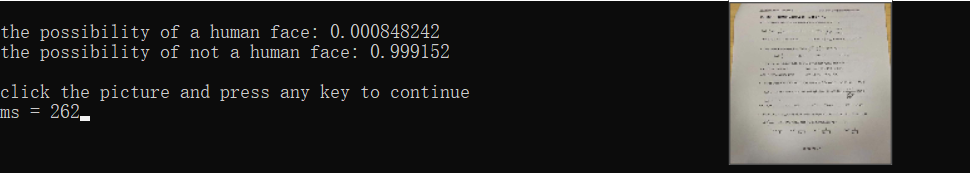
该模型中计算量最大的部分为点乘，为了加速算法，一开始我采用了OpenCV中的指令集对向量乘法进行了优化。但是后来发现采用指令集的效果并不如直接写乘法的效率高。经过思考与查阅资料，认为由于参数函数一般只有27个值，指令集充分利用寄存器的优势无法体现，反而因为指令集操作调用繁复，导致产生了大量额外的开销。相比之下，可能编译器本身对普通乘法也进行了优化，使得普通乘法效率更高。同时，在指令集的调用过程中还会损失一定的精度（编译器在内部运算中float会转化为double）。



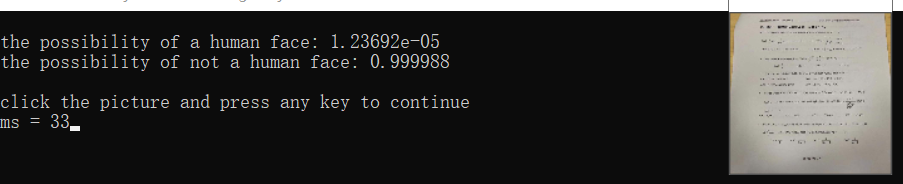
（原算法，充分利用寄存器，本人电脑为128位）



（改进算法，简单易懂又实用）



（使用指令集，vs2019, debug）



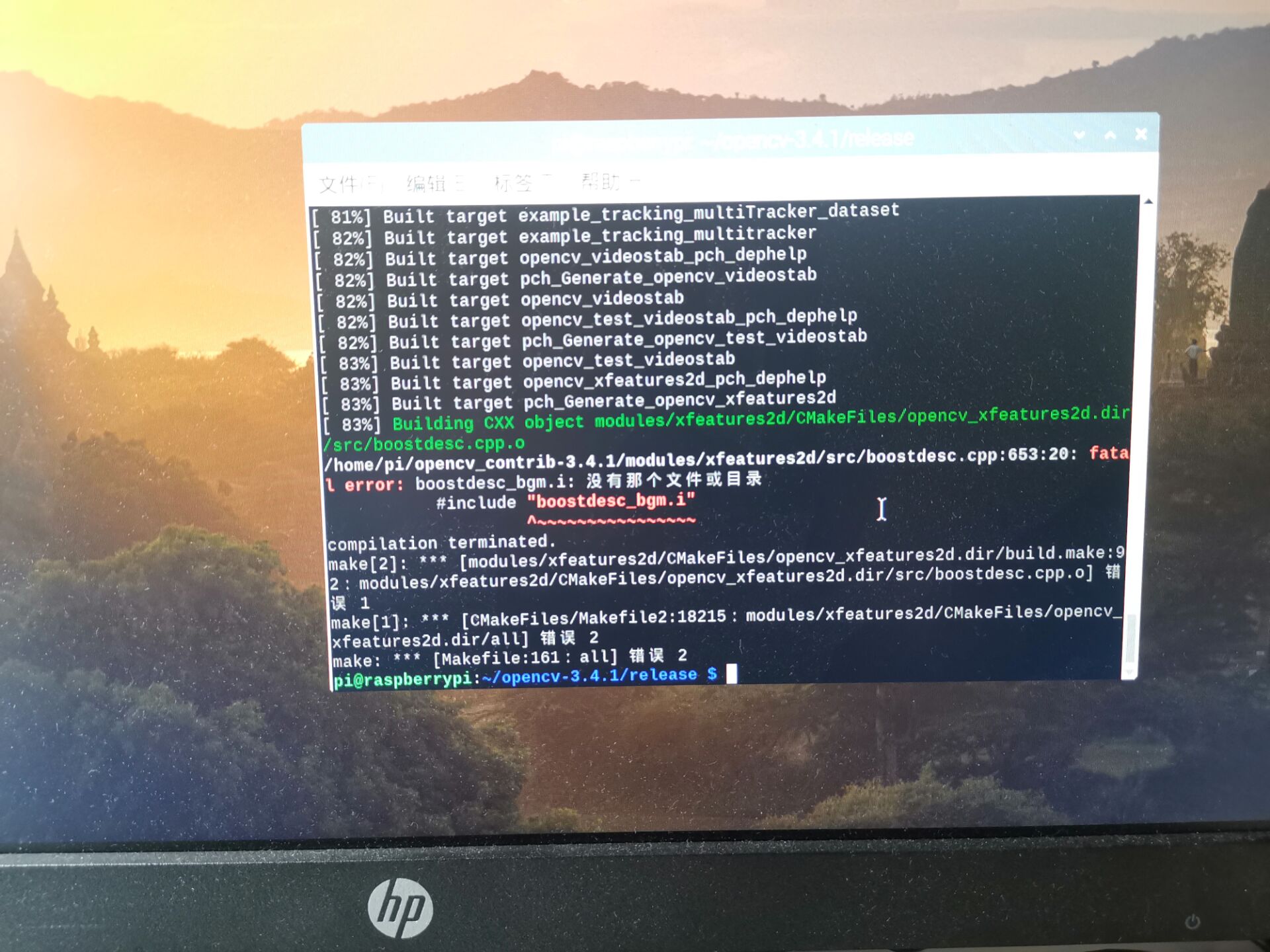
（不使用指令集，vs2019,debug）

效率提高了8倍左右

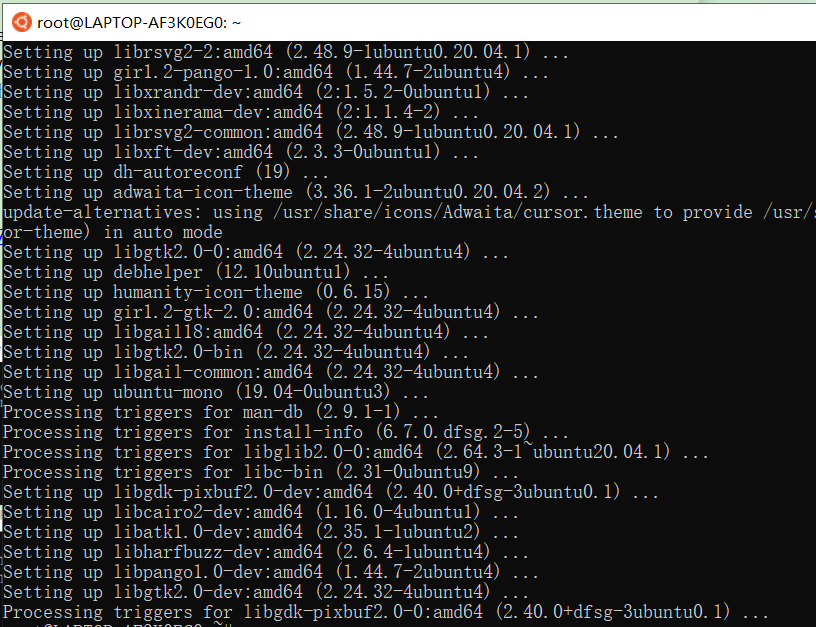
4. (5 points) The program is tested both on X86 and ARM. It can output the same results

for the same inputs on the two platforms.

虽然我尽力尝试着在linux上配置OpenCV，试图直接在arm上运行城市，但是由于各种原因未能成功。



我又尝试了另一种方式，也许如果我可以在Windows系统上生成.out文件，那么直接拷贝到linux系统上应当也可以正常运行，但是由于OpenCV安装的编译时间过长，也并未能实现这一方案。



跨平台操作十分繁琐，如果今后能开发一种平滑的转换器，将PE格式的.exe文件与elf格式的.out文件随意切换就好了。

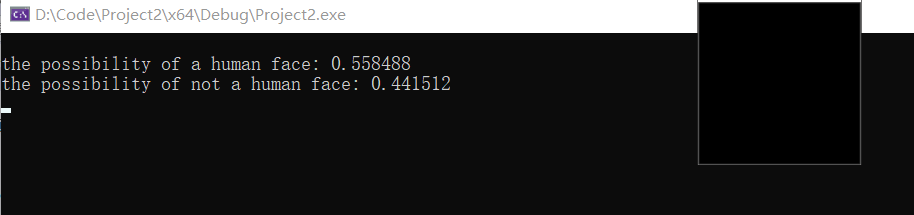
5. (5 points) Please host your source code at GitHub.com. you can just put a link in the

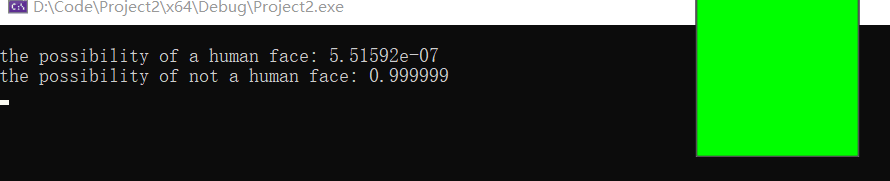
report.

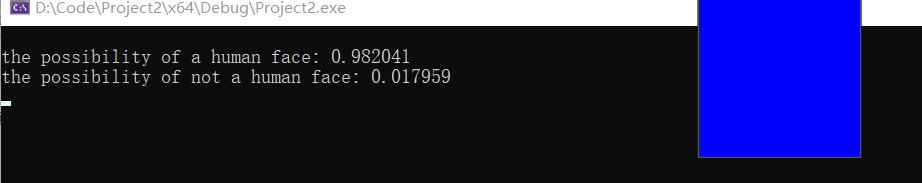
<https://github.com/13439797368/project_2.git>

1. 问题

我发现这组参数虽然可以很好的对正常图片进行识别，但是对纯色图片却不能进行识别







猜想是当时学习的模型中并没有纯色的样本。在学习的样本中加入纯色样本也许可以让这组参数的准确度更高一些。