# 图像处理算法设计

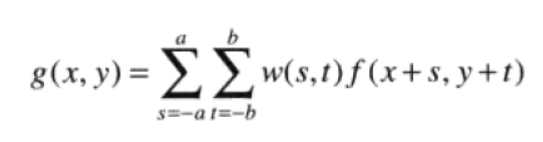
### 实验梗概

利用所学知识基于C++编程实现图像处理，本次选择对图像进行均值滤波以达到降低人脸分辨率的目的（类似于加上马赛克），该功能可以对特殊人员起到保护隐私的目的。在完成作业的基础上，考虑到实现过程中需要手动选定处理区域，系统泛化性能较差，所以我通过查找相关资料，最终实现利用C++自动捕捉人脸，可与摄像头相结合实时获取人脸信息，进而实现信息采集、打击犯罪等功能。

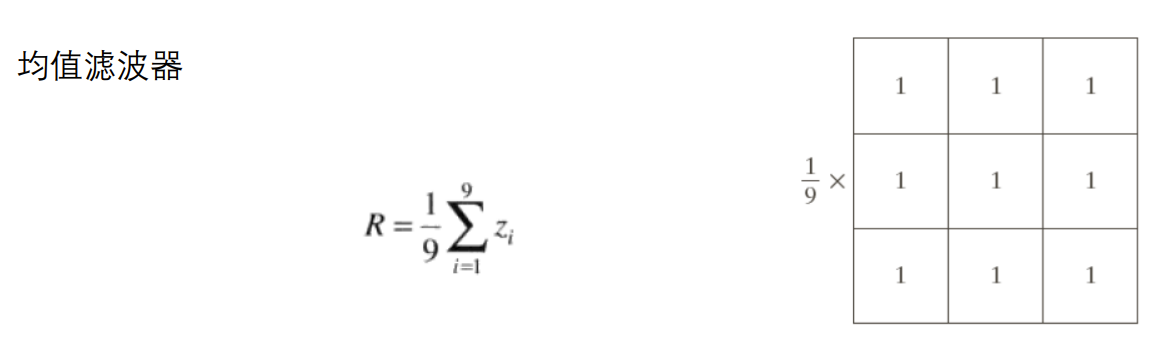
### 实验原理

###### 均值滤波

均值滤波也称为线性滤波，其采用的主要方法为邻域平均法。结合老师教授的PPT，线性滤波的基本原理是用均值代替原图像中的各个像素值，即对待处理的当前像素点（x，y），选择一个窗函数，该窗函数由其相邻的若干像素组成，首先得到窗函数中所有像素的均值，再把该均值赋予当前像素点（x，y），作为处理后图像在该点上的灰度g（x，y），即：



在老师的PPT中对此有很好的解释：



由于利用窗函数处理灰度数据时最外围的像素数据无法访问，此时应该提前将待处理像素数据最外围数据补0，例如若此时模板大小为3\*3，应在像素矩阵最上面，最下面，最左面，最右面补0，以使所有像素数据都可以被处理，这与深度学习中的卷积神经网络十分相似，该操作被称为Padding。然后在与窗函数的对应位置上做逐个元素的相关运算（相关不等于卷积），最后只取原像素数据大小的矩阵输出得到均值滤波后的图片即可。

###### 算法讲解

1. 读取图像数据

//读取待处理图像

Mat My\_image = imread("./Shelby.png"); //利用相对路径进行读取

namedWindow("待处理图像", WINDOW\_AUTOSIZE);

imshow("待处理图像", My\_image);

1. 截取图像处理位置信息ROI(Region of Interest)

// 获取图像处理位置

Rect rect(390, 135, 370, 350);//四个参数分别代表窗口中心的X、Y以及窗口本身的宽与高

Mat roi = My\_image(rect);

imshow("ROI", roi);

1. 对ROI区域的像素数据进行均值滤波

//对人脸进行滤波处理

int kernel = 13; //设置卷积核kernel的大小，此时效果最佳

//特别地，通过实际操作发现：卷积核越小，模糊效果越差，反之，将从模糊效果逐渐变为提取边缘特征

Mat image = roi.clone();

My\_MeanFilter(image, image, kernel); //对roi区域图像数据进行均值滤波

namedWindow("处理后", WINDOW\_AUTOSIZE);

imshow("处理后", image);

//直接手撸一个均值滤波函数

void My\_MeanFilter(const Mat& My\_image, Mat& Target, int kernel\_size) // My\_image 为待处理的图像，Target为目标图像，kernel\_size为卷积核的尺寸，卷积核的尺寸必须为奇数，不然无法进行卷积操作

{

int\* kernel = new int[kernel\_size \* kernel\_size]; // 设置卷积核的大小

for (int i = 0; i < kernel\_size \* kernel\_size; i++) // 首先以均值滤波为例，初始化其参数均为1

kernel[i] = 1;

Mat Container; //存放像素数据

int padding\_len = kernel\_size / 2;

//类似于深度学习中的Padding操作，为卷积后的像素数据暂存矩阵添加行与列

Container.create(Size(My\_image.cols + 2 \* padding\_len, My\_image.rows + 2 \* padding\_len), My\_image.type());

Target.create(Size(My\_image.cols, My\_image.rows), My\_image.type());

//由于为彩色图片，所以图片有三个维度，自然需要获取RGB三通道，然后分别进行处理

int channel = My\_image.channels();

//定义两个指针，指向待处理图像和暂存矩阵

uchar\* pM = My\_image.data;

uchar\* pC = Container.data;

//行与列都进行Padding

for (int row = 0; row < Container.rows; row++) //遍历每一行

{

for (int col = 0; col < Container.cols; col++) //遍历每一列

{

for (int num\_channel = 0; num\_channel < channel; num\_channel++) //遍历每一个通道

{

if (row >= padding\_len && row < Container.rows - padding\_len && col >= padding\_len && col < Container.cols - padding\_len)

pC[(Container.cols \* row + col) \* channel + num\_channel] = pM[(My\_image.cols \* (row - padding\_len) + col - padding\_len) \* channel + num\_channel]; //如果均满足，则可按照My\_image的指针对container中的像素数据进行处理

else

pC[(Container.cols \* row + col) \* channel + num\_channel] = 0;//否则就进行补0操作

}

}

}

//定义指针，指向目标图像

uchar\* pd = Target.data;

pC = Container.data;

for (int row = padding\_len; row < Container.rows - padding\_len; row++)//乘积求和

{

for (int col = padding\_len; col < Container.cols - padding\_len; col++)

{

for (int c = 0; c < channel; c++)

{

short t = 0;

for (int x = - padding\_len; x <= padding\_len; x++)

{

for (int y = -padding\_len; y <= padding\_len; y++)

{

t += kernel[(padding\_len + x) \* kernel\_size + y + padding\_len] \* pC[((row + x) \* Container.cols + col + y) \* channel + c]; //从暂存矩阵（已经Padding）中使用均值滤波，得到的结果赋值为t

}

}

//此处注意要防止数据溢出(int是32位数据，ushort是16位数据)

pd[(Target.cols \* (row - padding\_len) + col - padding\_len) \* channel + c] = saturate\_cast<ushort> (t / (kernel\_size \* kernel\_size)); //将t的结果首先做kernel\_size的平均，再通过指针放在目标图像矩阵当中

}

}

}

delete[] kernel; // 将均值滤波定义的kernel删除，节约内存

}

1. 将处理后的ROI数据覆盖在原始图像上，并显示。

//将处理后的图像数据覆盖在ROI上

image.copyTo(roi);

namedWindow("结果");

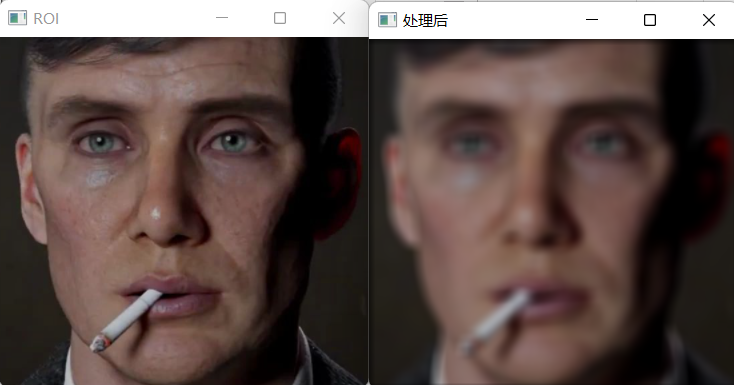
imshow("结果", My\_image);

### 实验结果

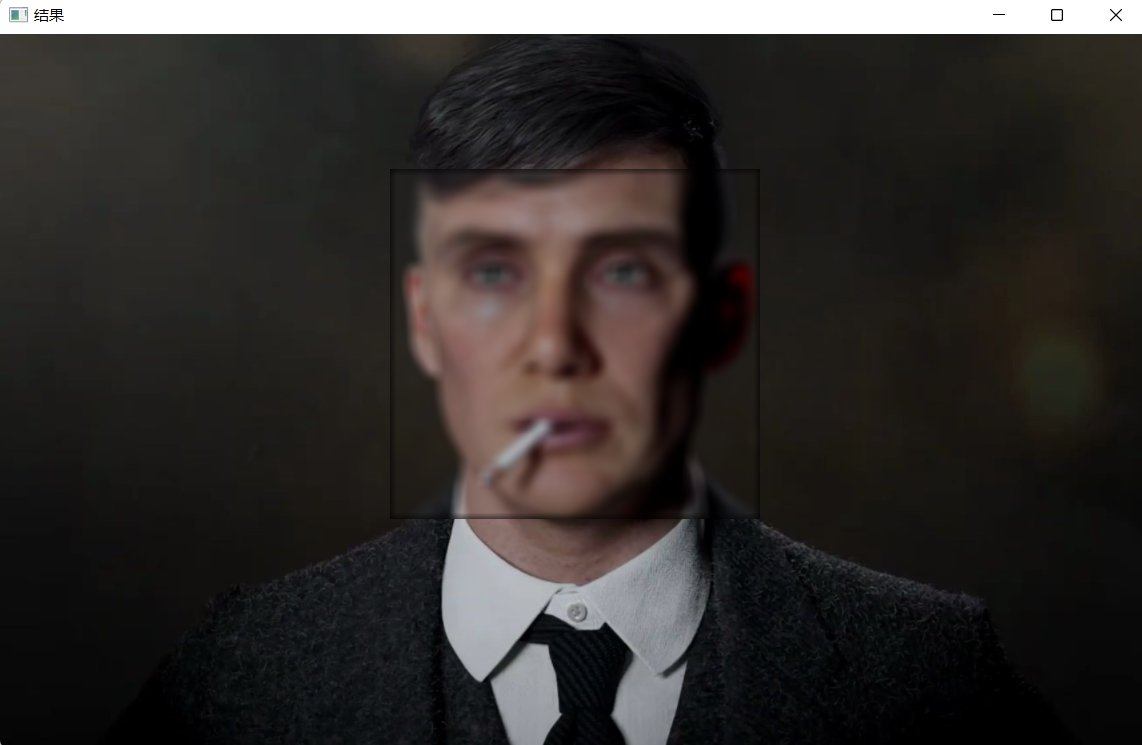
输入待处理图像：



截取待处理位置数据（ROI），并做均值滤波处理，得到如下结果：



通过处理可以看出图像变得非常模糊，接下来再将处理后的图像覆盖到原图像后，得到最终结果：

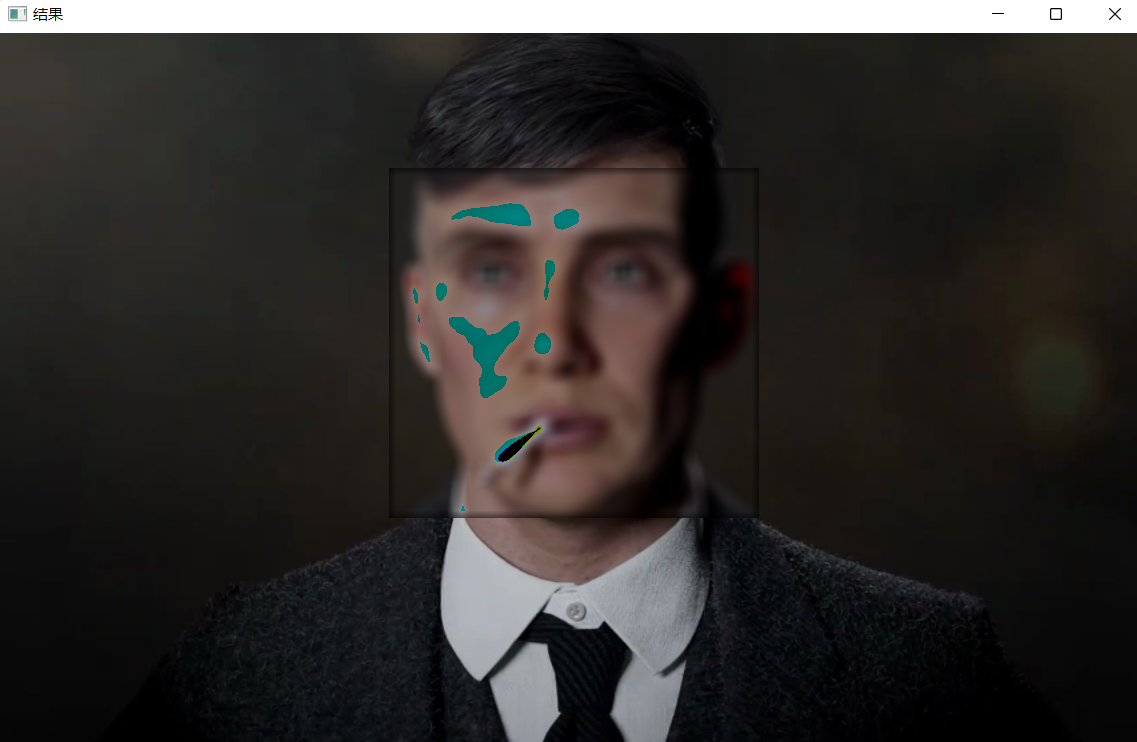


从而完成了降低面部分辨率的目的。

此时如果更改卷积核大小，即：int kernel = 5; 结果如下：



可以发现此时并没有起到很好的模糊效果，这与我们的直观认识一致。接下来不妨增大卷积核大小，即：int kernel = 15;结果如下：



此时可以发现不再进行均值滤波滤波，而是进行高频特征的提取。

### 实验创新

由于上述实验需要手动调节ROI的具体位置，然而不同的图像所对应的ROI位置也是不同的，因此这意味着该系统的捕捉人脸信息泛化能力较差，所以通过查找相关文献，结合OpenCV的人脸识别训练过程，我最终实现了人脸信息的自动捕捉功能。

###### 算法讲解

1.载入待处理图像

//载入待处理图像

Mat My\_image = imread("./Shelby.png");

resize(My\_image, My\_image, Size(1146, 715));

imshow("输入图像", My\_image);

2.载入分类器

//载入检测器

CascadeClassifier detector;

detector.load(xmlPath);

if (!detector.load(xmlPath)) //加载训练文件

{

cout << "不能加载指定的xml文件" << endl;

return -1;

}

3.选取检测部位

//指定检测人脸

vector<Rect> faces;

detector.detectMultiScale(My\_image, faces, 1.1, 3, 0, Size(30, 30));

for (size\_t t = 0; t < faces.size(); t++)

{

rectangle(My\_image, faces[t], Scalar(0, 0, 255), 2, 8); //使用矩形框描绘出检测区域

}

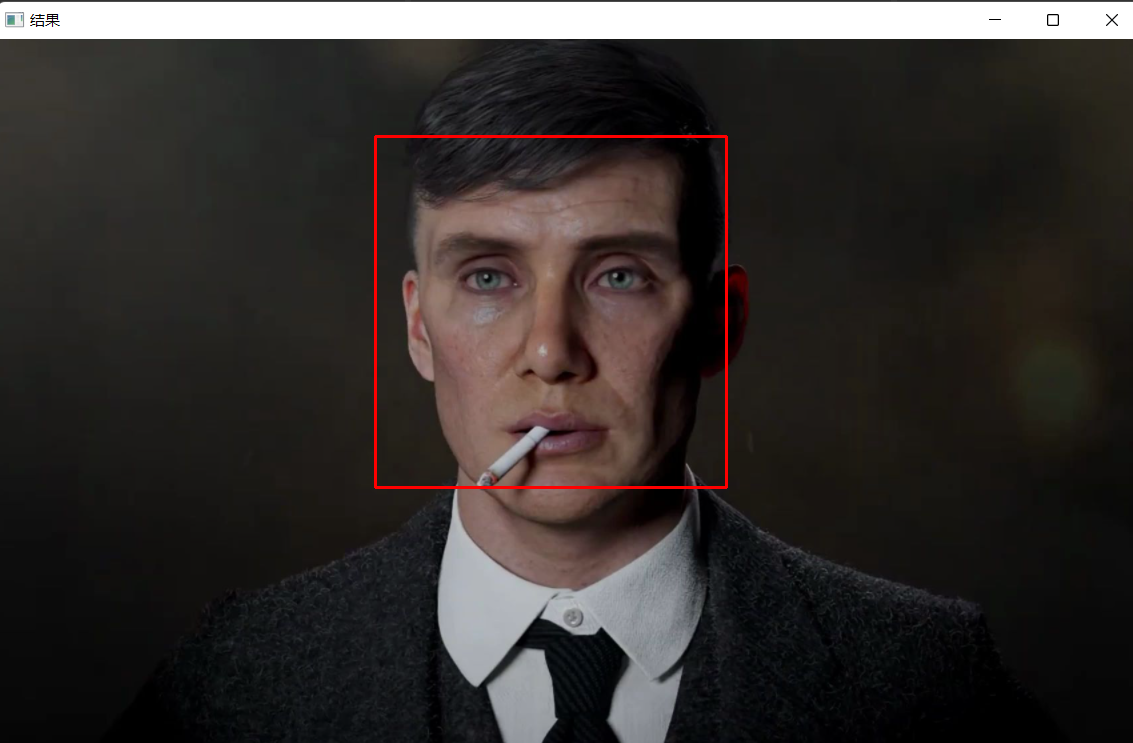
imshow("结果", My\_image);

###### 结果展示

输入图像：



处理结果：



可见相比于手动调节ROI，通过检测器进行检测更加精准、耗时更少，此时再结合之前实现的均值滤波器My\_MeanFilter即可高效地、实时地获取最终输出结果。

### 问题与解决方法

问题1：程序始终无法正确读取输入图像

**方法**：在一次偶然尝试中，发现原来是解决方案配置选择了Debug，从而解决了困扰两天的Bug

问题2：卷积核的尺寸与待处理图像尺寸不知如何匹配

**方法**：结合深度学习基础，需要进行Padding的处理，具体操作为给待处理图像“补0”，通过查找网上相关资料最终解决

### 总结与收获

**创新点：**

1.初步掌握了课程知识，能够将均值滤波原理结合OpenCV用C++实现，降低图像某特定区域的分辨率，从而达到保护隐私等社会实际需求的目的。

2.基于OpenCV的人脸识别训练过程，能够实现输入任意图像，迅速捕捉人脸信息，完成识别功能。

**总结：**

本次基于所学内容利用C++初步完成了图像处理算法设计，在基础实验部分当中，通过手动调节ROI对指定区域进行均值滤波，完成降低分辨率的目的，但其不足之处在于需要依靠手动调节ROI，不够精确，因此在实验创新部分，通过查阅相关资料发现可以通过载入检测器，从而精准地、快速地捕捉人脸等某一特定部位信息。

通过本次的实践操作，更加了解部分图像处理的底层逻辑，虽然自己已学习将近两年深度学习的相关知识，但依然感觉收获满满。

###### 附录：

// SHIJUEMEITI(3).cpp : 此文件包含 "main" 函数。程序执行将在此处开始并结束。

//

//导入工具包

#include<opencv2/opencv.hpp>

#include<iostream>

#include <opencv2/core/core.hpp>

#include <opencv2/highgui/highgui.hpp>

#include<stdlib.h>

using namespace cv;

using namespace std;

//直接手撸一个均值滤波函数

void My\_MeanFilter(const Mat& My\_image, Mat& Target, int kernel\_size) // My\_image 为待处理的图像，Target为目标图像，kernel\_size为卷积核的尺寸，卷积核的尺寸必须为奇数，不然无法进行卷积操作

{

int\* kernel = new int[kernel\_size \* kernel\_size]; // 设置卷积核的大小

for (int i = 0; i < kernel\_size \* kernel\_size; i++) // 首先以均值滤波为例，初始化其参数均为1

kernel[i] = 1;

Mat Container; //存放像素数据

int padding\_len = kernel\_size / 2;

//类似于深度学习中的Padding操作，为卷积后的像素数据暂存矩阵添加行与列

Container.create(Size(My\_image.cols + 2 \* padding\_len, My\_image.rows + 2 \* padding\_len), My\_image.type());

Target.create(Size(My\_image.cols, My\_image.rows), My\_image.type());

//由于为彩色图片，所以图片有三个维度，自然需要获取RGB三通道，然后分别进行处理

int channel = My\_image.channels();

//定义两个指针，指向待处理图像和暂存矩阵

uchar\* pM = My\_image.data;

uchar\* pC = Container.data;

//行与列都进行Padding

for (int row = 0; row < Container.rows; row++) //遍历每一行

{

for (int col = 0; col < Container.cols; col++) //遍历每一列

{

for (int num\_channel = 0; num\_channel < channel; num\_channel++) //遍历每一个通道

{

if (row >= padding\_len && row < Container.rows - padding\_len && col >= padding\_len && col < Container.cols - padding\_len)

pC[(Container.cols \* row + col) \* channel + num\_channel] = pM[(My\_image.cols \* (row - padding\_len) + col - padding\_len) \* channel + num\_channel]; //如果均满足，则可按照My\_image的指针对container中的像素数据进行处理

else

pC[(Container.cols \* row + col) \* channel + num\_channel] = 0;//否则就进行补0操作

}

}

}

//定义指针，指向目标图像

uchar\* pd = Target.data;

pC = Container.data;

for (int row = padding\_len; row < Container.rows - padding\_len; row++)//乘积求和

{

for (int col = padding\_len; col < Container.cols - padding\_len; col++)

{

for (int c = 0; c < channel; c++)

{

short t = 0;

for (int x = - padding\_len; x <= padding\_len; x++)

{

for (int y = -padding\_len; y <= padding\_len; y++)

{

t += kernel[(padding\_len + x) \* kernel\_size + y + padding\_len] \* pC[((row + x) \* Container.cols + col + y) \* channel + c]; //从暂存矩阵（已经Padding）中使用均值滤波，得到的结果赋值为t

}

}

//此处注意要防止数据溢出(int是32位数据，ushort是16位数据)

pd[(Target.cols \* (row - padding\_len) + col - padding\_len) \* channel + c] = saturate\_cast<ushort> (t / (kernel\_size \* kernel\_size)); //将t的结果首先做kernel\_size的平均，再通过指针放在目标图像矩阵当中

}

}

}

delete[] kernel; // 将均值滤波定义的kernel删除，节约内存

}

int main(int argc, char\*\* argv)

{

//读取待处理图像

Mat My\_image = imread("./Shelby.png"); //利用相对路径进行读取

namedWindow("待处理图像", WINDOW\_AUTOSIZE);

imshow("待处理图像", My\_image);

// 获取图像处理位置

Rect rect(390, 135, 370, 350);//四个参数分别代表窗口中心的X、Y以及窗口本身的宽与高

Mat roi = My\_image(rect);

imshow("ROI", roi);

//对人脸进行滤波处理

int kernel = 13; //设置卷积核kernel的大小，此时效果最佳

//特别地，通过实际操作发现：卷积核越小，模糊效果越差，反之，将从模糊效果逐渐变为提取边缘特征

Mat image = roi.clone();

My\_MeanFilter(image, image, kernel); //对roi区域图像数据进行均值滤波

namedWindow("处理后", WINDOW\_AUTOSIZE);

imshow("处理后", image);

//将处理后的图像数据覆盖在ROI上

image.copyTo(roi);

namedWindow("结果");

imshow("结果", My\_image);

waitKey(0);//一直等待按键。

return 0;

}

//Detection.cpp : 此文件包含 "main" 函数。程序执行将在此处开始并结束。

//

//导入工具包

#include<opencv2/opencv.hpp>

#include<iostream>

#include <opencv2/core/core.hpp>

#include <opencv2/highgui/highgui.hpp>

#include<stdlib.h>

using namespace cv;

using namespace std;

int main(int argc, char\*\* argv)

{

//选择OpenCV的训练结果路径

string xmlPath = "C:/Users/lenovo/Desktop/C++ PROJECT/opencv/build/etc/haarcascades/haarcascade\_frontalface\_default.xml";

//载入待处理图像

Mat My\_image = imread("./Shelby.png");

resize(My\_image, My\_image, Size(1146, 715));

imshow("输入图像", My\_image);

//载入检测器

CascadeClassifier detector;

detector.load(xmlPath);

if (!detector.load(xmlPath)) //加载训练文件

{

cout << "不能加载指定的xml文件" << endl;

return -1;

}

//指定检测人脸

vector<Rect> faces;

detector.detectMultiScale(My\_image, faces, 1.1, 3, 0, Size(30, 30));

for (size\_t t = 0; t < faces.size(); t++)

{

rectangle(My\_image, faces[t], Scalar(0, 0, 255), 2, 8); //使用矩形框描绘出检测区域

}

imshow("结果", My\_image);

waitKey(0);

return 0;

}