> 本节为opencv数字图像处理（1）：灰度变换与空间滤波的第一小节，灰度变换函数，主要包括：图像反转、对数变换、伽马变换、分段线性变换函数（包括对比度拉伸、灰度级分层和比特平面分层）及其C++代码实现。

1.1 图像反转

s和r分别表示处理前后的像素值，则应用反转变换可以得到灰度级范围为$[0, L-1]$的一幅图像的反转图像，由该式给出：$s = L - 1 - r$。　　这种反转图像灰度级的处理适用于增强嵌入在一幅图像的暗区域中的白色或灰色细节，特别是黑色面积在尺寸上占主导地位时。

图像灰度级反转的C++代码如下：

```
#include <iostream>
#include <opencv2\core\core.hpp>
#include <opencv2\highgui\highgui.hpp>
#include <opencv2\imgproc\imgproc.hpp>
using namespace std;
using namespace cv;
/*线性变化之灰度级反转*/
void grayInv()
{
 Mat srcImg = imread("test.PNG", 0);
 if (!srcImg.data)
 {
  cout << "fail to load image" << endl;
  return;
 }
 int k = -1, b = 255;
 int rowNum = srcImg.rows;
 int colNum = srcImg.cols;
 Mat dstImg(srcImg.size(), srcImg.type());
 for (int i = 0; i < rowNum; i++)
 {
  uchar* srcData = srcImg.ptr<uchar>(i);
  for (int j = 0; j < colNum; j++) {
   dstImg.at<uchar>(i, j) = srcData[j] * k + b;
  }
 }
 imshow("original", srcImg);
 imshow("grayInv", dstImg);
```

```
 waitKey(0);
}
```

> opencv4.3.0+Visual Studio 2019环境配置 waitKey的作用

1.2 对数变换

s和r分别表示处理前后的像素值，对数变换的通用形式可以表达为$s = clog(1 + r)$，c是一个常数。该变换的将输入中范围较窄的低灰度值映射为输出中较宽范围的灰度值，对高的输入灰度值也如此。从而实现扩展图像中暗像素的值同时压缩高灰度级的值的目的。一种简易的实现就是将上面给出的代码中像素操作替换为：

```
dstImg.at<uchar>(i, j)=log(double(1 + (double)srcImg.at<uchar>(i, j)));
```

需要注意的是，之后我们需要进行灰度的归一化处理以使得灰度值分布在0-255之间，代码如下：

```
// 归一化处理
 cv::normalize(dstImg, dstImg,
  0, 255, cv::NORM_MINMAX);
 cv::convertScaleAbs(dstImg, dstImg);
```

1.3 幂律（伽马）变换

s和r分别表示处理前后的像素值，幂律变换的基本形式：$s = cr^{\gamma}$。其中c和γ是正常数，有时考虑偏移量（输入为0）幂律变换也可以写为：$s = c(r + \epsilon)^{\gamma}$。 当$\gamma < 1$时，与对数变换类似，幂律变换将较窄范围的暗色输入值映射为较宽范围的输出值，图像整体灰度值增大，用于提高图像暗区域中的对比度，而降低亮区域的对比度；当大于1时则相反，用于提高图像中亮区域的对比度，降低图像中按区域的对比度。 C++代码如下：

```
void GammaCorr()
{
 Mat srcImg = imread("test.PNG");
 if (srcImg.empty())
 {
  cout << "fail to load" << endl;
  return;
 }
 float fGamma = 1 / 3.2;
```

```
//建立待查表文件LUT
unsigned char lut[256];
for (int i = 0; i < 256; i++) {
 //防止彩色溢出，大于255的像素令其为255，小于0的像素令其为0
 lut[i] = saturate_cast<uchar>(pow((float)(i / 255.0), fGamma) * 255.0f);
}
Mat dstImg = srcImg.clone();
const int channels = dstImg.channels();
switch (channels)
{
case 1://灰度图
{
 MatIterator_<uchar> it, end;
 for (it = dstImg.begin<uchar>(), end = dstImg.end<uchar>(); it != end; it++)
  //*it = pow((float)(((*it)) / 255.0), fGamma) * 255.0;
  *it = lut[(*it)];
 break;
}
case 3://彩色图
{
 MatIterator_<Vec3b> it, end;
 for (it = dstImg.begin<Vec3b>(), end = dstImg.end<Vec3b>(); it != end; it++)
 {
  //(*it)[0] = pow((float)(((*it)[0])/255.0), fGamma) * 255.0;
  //(*it)[1] = pow((float)(((*it)[1])/255.0), fGamma) * 255.0;
  //(*it)[2] = pow((float)(((*it)[2])/255.0), fGamma) * 255.0;
  (*it)[0] = lut[((*it)[0])];
  (*it)[1] = lut[((*it)[1])];
  (*it)[2] = lut[((*it)[2])];
 }
 break;
}
}
imshow("ori", srcImg);
imshow("res", dstImg);
waitKey(0);
}
```

1.4 分段线性变换函数

- 对比度拉伸

低对比度图像的成因可能是照明不足、成像传感器动态范围太小或者图像获取过程中镜头光圈设置错误引起，对比度拉伸的作用是扩展图像灰度级动态范围。 下图所示是一个典型的对比度拉伸变换的函数形状，不同的$r1, r2, s1, s2$取值也对应这不同的灰度级变换：

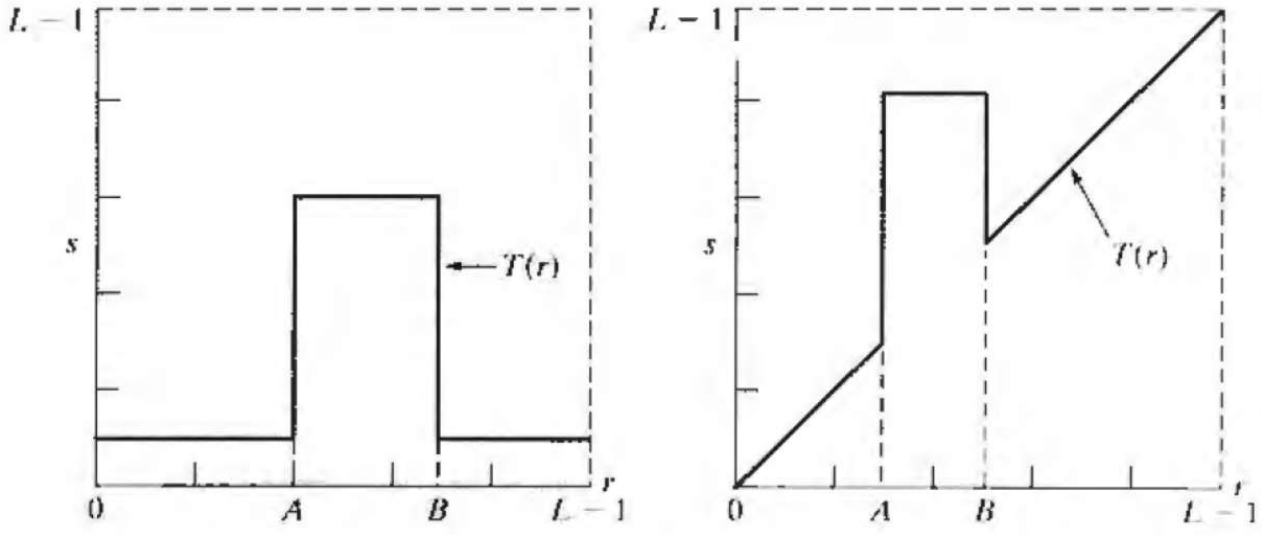
当$r1=s1$且$r2=s2$，则是一个线性变换；当$r1=r2$，$s1=0$，$s2=L-1$（最大灰度级），则是一个阈值处理函数（产生一个二值图像）。　　针对灰度图像的对比度拉伸C++实现如下：

```
void contrastStretch()
{
 Mat srcOri = imread("test1.JPG");
 Mat srcImage;
 cvtColor(srcOri, srcImage, CV_RGB2GRAY);
 //"=";"clone()";"copyTo"三种拷贝方式，前者是浅拷贝，后两者是深拷贝。
 Mat resultImage = srcImage.clone();
 int nRows = resultImage.rows;
 int nCols = resultImage.cols;
 //判断图像存储的连续性，若连续可以得到像素个数
 if (resultImage.isContinuous())
 {
  nCols = nCols * nRows;
  nRows = 1;
 }
 //图像指针操作
 uchar* pDataMat;
 int pixMax = 0, pixMin = 255;
 //计算图像的最大最小值
 for (int j = 0; j < nRows; j++)
 {
  //ptr<>()得到的是一行指针，智能指针+模板类
  pDataMat = resultImage.ptr<uchar>(j);
  for (int i = 0; i < nCols; i++)
  {
   if (pDataMat[i] > pixMax)
    pixMax = pDataMat[i];
   if (pDataMat[i] < pixMin)
    pixMin = pDataMat[i];
  }
 }
 //对比度拉伸映射，从原始范围拉伸到num1~num2
 int num1 = 100, num2 = 200;
 for (int j = 0; j < nRows; j++)
 {
  pDataMat = resultImage.ptr<uchar>(j);
  for (int i = 0; i < nCols; i++)
  {
   pDataMat[i] = (pDataMat[i] - pixMin) * (num2 - num1) / (pixMax - pixMin) + num1;
  }
 }
```

```
    imshow("ori", srcImage);
    imshow("dst", resultImage);
    waitKey(0);
}
```

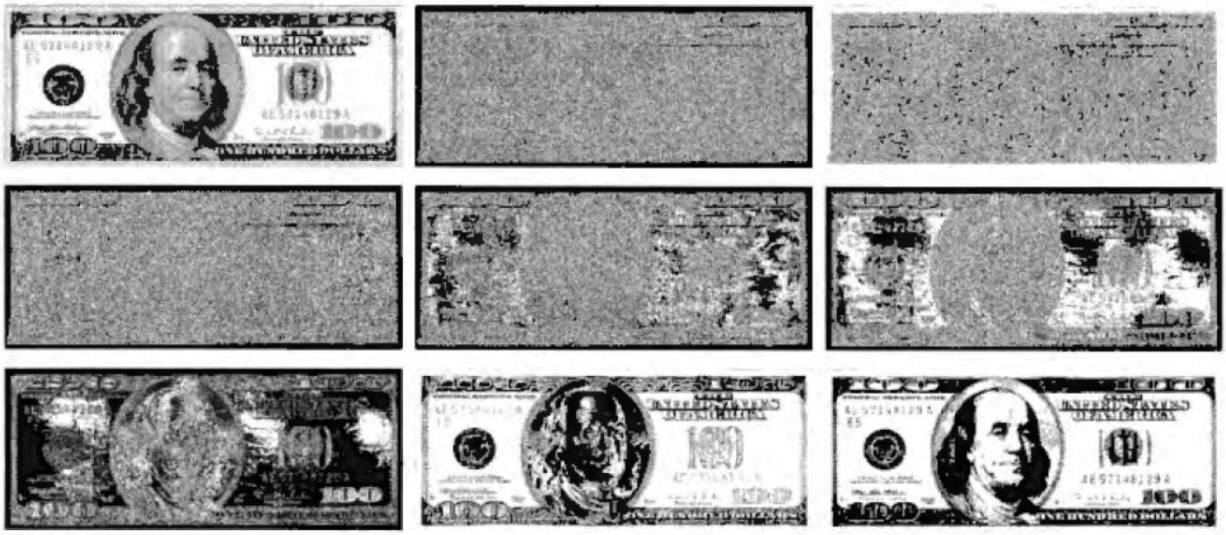
- 灰度级分层

灰度级分层可以突出图像中特定灰度范围的亮度，一种方式即感兴趣范围设为一个值，不感兴趣设置为另一个值；另一种方式是使感兴趣区域变亮或变暗，保持图像中其他的灰度级不变。两种方式的变换函数的形状如下图：



具体实现的话可以对上述对比度拉伸的代码进行简单调整即可。

- **比特平面分层** 下图显示了一个8比特灰度图像（第一幅）以及1~8比特的比特平面（后八幅）。可以直观看出，低阶比特平面在图像中贡献了更精细的灰度细节，高阶比特平面包含了视觉上很重要的大多数数据，



显示一幅8比特图像的第8个比特平面可用阈值灰度变换函数处理输入图像得到二值图像，将0~127之间的灰度映射为0，其他的映射为1。比特平面分层以及使用几个高阶比特平面重构图像的作用在于图像压缩和突出部分比特特征。C++代码如下：

```
void bitLevel()
```

```
{
 Mat srcImage = imread("test2.JPG", 0);
 Mat d[8];
 int b[8];
 for (int k = 0; k < 8; k++)
  //CV_8UC1：其中8代表比特数，0~255；U代表无符号整型，F代表单精度浮点型；
  //C代表通道数；1代表灰度图像即单通道，2代表RGB彩色图像即三通道，3代表
  //带Alpha通道（透明度）的RGB图像，即四通道
  d[k].create(srcImage.size(), CV_8UC1);
 int rowNum = srcImage.rows, colNum = srcImage.cols;
 for (int i = 0; i < rowNum; i++)
  for (int j = 0; j < colNum; j++) {
   int num = srcImage.at<uchar>(i, j);
   //
   for (int p = 0; p < 8; p++)
    b[p] = 0;
   int q = 0;
   while (num != 0)
   {
    b[q] = num % 2;
    num = num / 2;
    q++;
   }
   //
   for (int k = 0; k < 8; k++)
    d[k].at<uchar>(i, j) = b[k] * 255;
  }
 imshow("ori", srcImage);
 for (int k = 0; k < 8; k++)
  imshow("bit" + to_string(1 + k), d[k]);
 waitKey(0);
}
```

欢迎扫描二维码关注微信公众号 深度学习与数学 [每天获取免费的大数据、AI等相关的学习资源、经典和最新的深度学习相关的论文研读，算法和其他互联网技能的学习，概率论、线性代数等高等数学知识的回顾]