第一次作业——报告

邓文 17307130171

系统环境：Windows
编程语言：Python3.7

1. 多维直方图 & 局部直方图

(1). 多维直方图的函数—— evaluate_histogram()
数据主要以 numpy.array 的形式存储，具体函数的 code 见代码 **EvaluateHistogram,py** 文件中。

参数：

- pData: 输入的多维数据，是以 array[D][N]的形式存储的，其中第一个索引代表维度，第二个代表样本的数量
- nData: 输入的样本数量
- dimension: 数据的维度
- bins: 直方图柱子的个数
- intensityMin: 直方图覆盖的范围最低值
- intensityMax: 直方图覆盖的范围最高值
- 返回值: 返回一个 array 统计了各个 bin 的频次分布

具体实现：

```
# bins
try:
    n = len(bins)
    if n != dimension:
        raise ValueError('the dimension of bins must be equal to the dimension of the data')
except TypeError:
    # bins has only one dimension in which means all the dimensions has the same bins
    bins = dimension * [bins]
# intensity
try:
    n = len(intensityMin)
    if n != dimension:
        raise ValueError('the dimension of intensities must  equal to dimensions of data')
except TypeError:
    # intensity has only one dimension in which means all the dimensions has the same intensity range
    intensityMin = dimension * [intensityMin]
try:
    m = len(intensityMax)
    if m != dimension:
        raise ValueError('the dimension of intensities must  equal to dimensions of data')
except TypeError:
    # intensity has only one dimension in which means all the dimensions has the same intensity range
```

首先对参数进行了标准化处理，这里是指:

- bins，intensityMin，intensityMax 的维度必须与输入的维度 dimension 一致。
- 在要求的各个维度的 bins 和直方图范围都一致时，允许 bins， intensityMin，intensityMax 仅输入一个数字以简化输入，如果不一致时，就需要输入包含各个维度的 bins，intensityMin，intensityMax。(列表形式或数组形式均可)。

```
bin_space = [0 for i in range(dimension)]
bin_pos = [0 for i in range(dimension)]
p_histogram = np.zeros(reduce(lambda x, y: x*y, bins), dtype=int)

for i in range(dimension):
    if bins[i] <= 0:
        raise ValueError("bins must be positive!")
    bin_space[i] = (intensityMax[i] - intensityMin[i])/bins[i]

for i in range(nData):
    for j in range(dimension):
        value = pData[j][i]
        bin_pos[j] = int((value - intensityMin[j])/bin_space[j])
        # out-of-boundary detection
        bin_pos[j] = max(bin_pos[j], 0)
        bin_pos[j] = min(bin_pos[j], bins[j]-1)

    index = bin_pos[0]
    for dim in range(1, dimension):
        size = 1
        for idv in range(dim):
            size *= bins[idv]
        index += size*bin_pos[dim]
    p_histogram[index] += 1

return p_histogram.reshape(bins)
```

然后开始统计直方图数据:

- 首先定义了 bin_space，bin_pos，p_histogram 三个变量分别存储每个维度 bin 的宽度，每个样本在各个维度 bin 的位置，以及最终的一维直方图频次数组。
- 然后求出每个维度 bin 的宽度，这里加入了一个检测，即 bins 参数必须为正数，负责会报错。
- 对于每一个样本点，求出他们在各个维度的 bin 的位置，然后再将他们映射到对应的 p_histogram 中。具体的公式为(x, y, z) → index = x+(bins[0])*y+(bins[0]*bins[1])*z 其中(x, y, z)对应 bin_pos
- 然后使 p_histogram[index] += 1
- 最后返回 p_histogram.

以上便是函数的主要内容，详细的代码可以看附件~

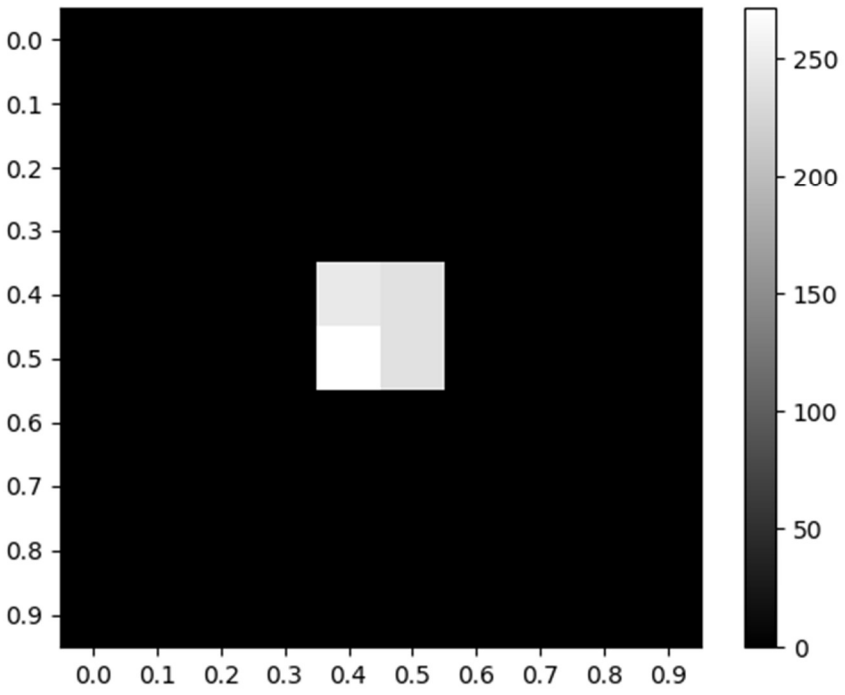
函数测试：

```
if __name__ == "__main__":
    import matplotlib.pyplot as plt
    # randomly generate 1000 two-dimensional sample
    np.random.seed(123)
    data = np.random.normal(0.5, 0.01, (2, 1000))
    histogram = evaluate_histogram(data, 1000, 2, 10, 0, 1)
    pic = plt.imshow(histogram, cmap=plt.get_cmap('gray'))
    plt.yticks(range(10), [i/10 for i in range(11)])
    plt.xticks(range(10), [i / 10 for i in range(11)])
    plt.colorbar(pic)
    plt.savefig("problem1-1.png")
    plt.show()
```

在这里，我测试了一个二维的数据样本：

- 首先设置随机数种子，使结果可以复现
- 然后用 numpy 中自带的生成随机数的方法生成了 1000 个 2 维的数据，每个元素都是服从 Normal(0.5, 0.01)的分布的。
- 对于这样的数据，我用我写好的代码计算他的直方图，并设置 bins 的数量为 10，直方图显示的范围为 0 到 1， 对于我们之前的样本，数据为 Normal（0.5，0.01）的分布，我们可以设想到，直方图大部分的频次都分布在（0.4~0.6，0.4~0.6）的范围内。
- 然后我们用 matplotlib.pyplot 库将我计算的直方图进行可视化展示，并保存图片。

结果展示：



解释：

- 该图像中的像素代表该位置的 bin 的高度，即频次。颜色越白，代表这里的样本越多，反之颜色越黑，代表这里的样本越少。
- 从图中可以看见中间有四块像素点要明显比周围的像素点白很多，这也符合了我之前对这样 1000 个数据样本的猜想。

至此，第一问对**直方图计算**的函数完成！

(2). 局部直方图计算—— local_histograms()

数据同样以 numpy.array 的形式存储，并且在这个函数中，引用了我们之前定义的 evaluate_histogram() 函数。这个函数采用了高效的局部直方图更新算法，节省很多运算。具体代码见附件 **local_update_hist.py** 文件中。

参数：

- pData: 输入的图像数据
- window_size:窗口的大小，一般是奇数，如果是偶数，窗口是偏向右下方的。
- bins: 直方图柱子的个数
- intensityMin:直方图覆盖范围的最小值
- intensityMax:直方图覆盖范围的最大值
- padding: 对边界进行填充的参数，如果不需要填充则设置为 None
- 返回值：一个字典包含了每个像素点对应的邻域直方图

具体实现：

```
    x_start, y_start = 0, 0
    x_length, y_length = pData.shape
    if padding is not None:
        radius = int(window_size/2)
        pData = np.c_[pData, padding*np.ones((pData.shape[0], radius))]
        pData = np.c_[padding*np.ones((pData.shape[0], window_size-radius-1)), pData]
        x_start += window_size-radius-1
        pData = np.r_[pData, padding*np.ones((radius, pData.shape[1]))]
        pData = np.r_[padding*np.ones((window_size-radius-1, pData.shape[1])), pData]
        y_start += window_size-radius-1
    histograms = {}
    cur_hist = None
```

首先针对参数做了一些预处理：

- x_start, y_start 代表了开始计算的像素的起始位置，x_length, y_length 代表了图片的长度和宽度。
- 如果 padding 不是 None 值，而是 0，1，2，…… 这样的具体数字，则需要对图像的边界进行扩充，在这里利用了 numpy 包中的 np.c_的方式进行扩展，同时还需要使像素的初始位置，x_start, y_start 各自加上填充的宽度。
- 初始化了一个 histograms 字典 用来存储各个像素点的邻域直方图。
- 初始化了一个 cur_hist 变量 用来存储当前像素点的直方图，初始值为 None.

```
def board_detection(data, pos, size):
    # this is a function which will return the coordinates of window corner
    x_len, y_len = data.shape
    px, py = pos
    r = int(size / 2)
    ex = px + r + 1
    ey = py + r + 1
    while ex > x_len:
        ex -= 1
    while ey > y_len:
        ey -= 1
    l = size - r - 1
    sx = px - l
    sy = py - l
    while sx < 0:
        sx += 1
    while sy < 0:
        sy += 1
    return sx, ex, sy, ey
```

然后定义一个函数用来计算每个像素点邻域左右上下边界，这样在后面的主体函数中调用更加方便。

```
    for x in range(x_start, x_start+x_length):
        for y in range(y_start, y_start+y_length):
            if cur_hist is None:  # initialize where there is not a histogram
                # will call a evaluate_histogram directly
                cur_left, cur_right, cur_up, cur_down = board_detection(pData, (x, y), window_size)
                local_pic = pData[cur_left:cur_right, cur_up:cur_down].reshape((1, -1))
                cur_hist = evaluate_histogram(local_pic, local_pic.size, 1, bins, intensityMin, intensityMax)
```

然后对于每个需要计算邻域直方图的像素点建立一个循环，在这里对于第一个像素点由于 cur_hist 是一个 None 值，所以不能用 efficient 的方法来计算它的直方图，需要在这个点给 cur_hist 进行初始化：

- 首先用上面定义的 boarder_detection() 函数求出这个像素点的领域边界。
- 然后利用求得的边界在将这个邻域从原始图像上取出来，并将它格式化为 evaluate_histogram() 函数的标准输入格式，即 array[D][N] 的形式。
- 然后直接调用之前定义好的 evaluate_histogram() 函数，计算出它的直方图。

```
histograms[(x, y)] = cur_hist
old_left, old_right, old_up, old_down = cur_left, cur_right, cur_up, cur_down
```

然后我们会把这个直方图保存在 histograms 字典中，并将之前求出的邻域边界存到新的变量 old_left, old_right, old_up, old_down 中。

```
            else:  # there is already a histogram to use local update
                cur_left, cur_right, cur_up, cur_down = board_detection(pData, (x, y), window_size)

                if cur_left > old_left:
                    # do deletion
                    delete_pic = pData[old_left:cur_left, old_up:old_down].reshape((1, -1))
                    cur_hist = cur_hist - evaluate_histogram(delete_pic, delete_pic.size, 1, bins, intensityMin,
                                                             intensityMax)
                if cur_right > old_right:
                    # add to hist
                    add_pic = pData[old_right:cur_right, old_up:old_down].reshape((1, -1))
                    cur_hist = cur_hist + evaluate_histogram(add_pic, add_pic.size, 1, bins, intensityMin,
                                                             intensityMax)

                if cur_up > old_up:
                    # do deletion
                    delete_pic = pData[old_left:old_right, old_up:cur_up].reshape((1, -1))
                    cur_hist = cur_hist - evaluate_histogram(delete_pic, delete_pic.size, 1, bins, intensityMin,
                                                             intensityMax)

                if cur_down > old_down:
                    # add to hist
                    add_pic = pData[old_left:old_right, old_down:cur_down].reshape((1, -1))
                    cur_hist = cur_hist + evaluate_histogram(add_pic, add_pic.size, 1, bins, intensityMin,
                                                             intensityMax)

            histograms[(x, y)] = cur_hist
            old_left, old_right, old_up, old_down = cur_left, cur_right, cur_up, cur_down
        cur_hist = histograms[(x, 0)]
        old_left, old_right, old_up, old_down = board_detection(pData, (x, 0), window_size)
    return histograms
```

然后对于其他的像素点，我们已经有了一个之前的邻域直方图，便可以直接使用直接的直方图，来进行高效的局部更新：

- 备注：为了方便计算，我们的邻域总是先向下移动到底，再向右移动一次，再向下移动到底的，以此循环~
- 我们先计算出当前像素点的邻域边界。
- 在进行直方图像素点频次的增加和删减的时候，我们仍然调用一个 evaluate_histogram() 函数来计算图片中那些需要增加或者减少的像素点的直方图，然后再在 cur_hist 直方图的基础上，加上或减去那些直方图上的数值。

然后进行判断：

- 如果 cur_left > old_left 就代表着邻域的左边界向右移动了一格，这时我们需要在 cur_hist 中删除左边边界之外的像素点的频次。
- 如果 cur_right > old_right 就代表着邻域的右边界向右移动了一格，这时我们需要在 cur_hist 中增加右边边界之内新增加的像素点的频次。
- 如果 cur_up > old_up 就代表着邻域的上边界向下移动了一格，这时我们需要在 cur_hist 中删除上边边界之外的像素点的频次。
- 如果 cur_down > old_down 就代表着邻域的下边界向下移动了一格，这时我们需要在 cur_hist 中增加下边边界之内新增加的像素点的频次。
- 最后需要强调的一点是当邻域移动到最下端时，下一个邻域会重新回到最上端的开始，只是 x 的值会加 1。这时候邻域直方图就很可能没有重合的地方了，针对这里在 x 循环内， 我们会利用 cur_hist = histograms[(x, 0)]将当前邻域直方图变更回最上端的邻域直方图，并更新 old_left, old_right, old_up, old_down
- 最后返回 histograms ， 即每个像素点对应的邻域直方图

以上便是函数的主要内容，详细的代码可以看附件~

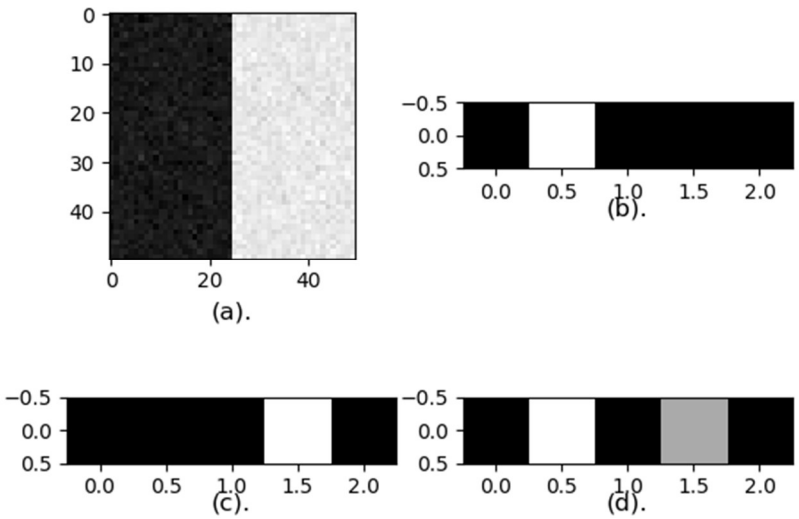
函数测试：

```
if __name__ == "__main__":
    import matplotlib.pyplot as plt
    np.random.seed(123)
    data = np.c_[np.random.normal(0.5, 0.04, (50, 25)), np.random.normal(1.5, 0.04, (50, 25))]
    plt.subplot(221)
    plt.imshow(data, cmap=plt.get_cmap('gray'))
    plt.title("(a).", y=-0.2)
    local_hists = local_histograms(data, 5, 5, 0, 2, None)
    plt.subplot(222)
    plt.imshow(np.array([local_hists[(3, 3)]]), cmap=plt.get_cmap('gray'))
    plt.title("(b).", y=-0.9)
    plt.subplot(223)
    plt.imshow(np.array([local_hists[(40, 40)]]), cmap=plt.get_cmap('gray'))
    plt.title("(c).", y=-0.9)
    plt.subplot(224)
    plt.imshow(np.array([local_hists[(24, 24)]]), cmap=plt.get_cmap('gray'))
    plt.title("(d).", y=-0.9)
    print(local_hists[(3, 3)])
    print(local_hists[(40, 40)])
    print(local_hists[(24, 24)])
    plt.savefig("problem1-3.png")
    plt.show()
```

在对函数的测试中：

- 首先设置随机数种子，使结果可以复现
- 利用 numpy 自带的随机数库生成了一个 50*25 的随机数矩阵，其中每个元素符合 Normal(0.5,0.04)的分布，和另一个 50*25 的随机数矩阵，其中每个元素符合 Normal(1.5,0.04)的分布。
- 然后用 np.c_的方法将这样两个矩阵拼接在一起形成新的图像。我们可以猜到这个图像左半边是比较偏黑色的点，右半边是会稍微偏白色一点。
- 然后我们计算出这个图像每个像素点的邻域直方图，在这里我们选择性的可视化了三个具有代表性的邻域直方图：第一个是在(3,3)的一个邻域直方图，由于这个像素点的邻域完全位于左半边，因此它的邻域直方图会在 0.5 处比较白一点；第二个是在(40，40)的邻域直方图，由于这个像素点的邻域完全位于右半边，因此它的邻域直方图会在 1.5 处比较白一点。第三个是在(24,24)的邻域直方图，由于这个像素点位于图片的中间交界处，因此它的邻域直方图应该会在 0.5 和 1.5 处都会比较白一点。

结果展示：



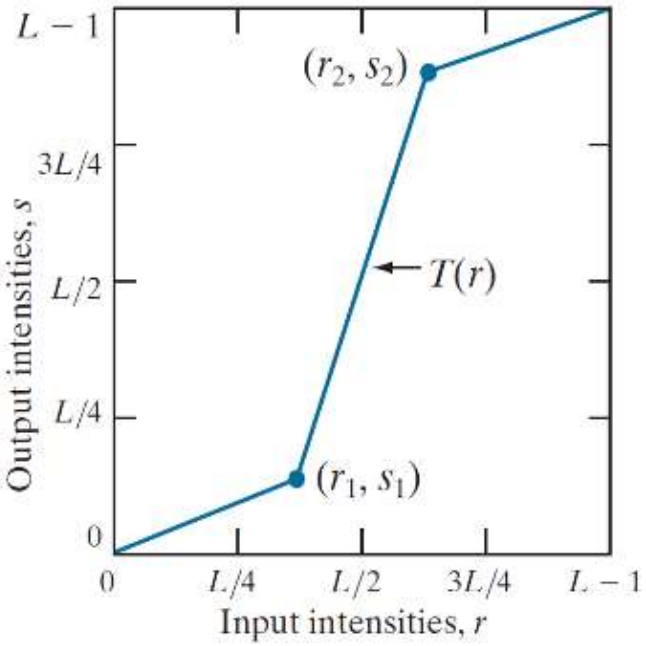
解释：

- 图(a) 是我们随机生成的图片，它的左半边比较黑，右半边偏白。
- 在可视化直方图时，像素点越白，代表该位置的频次越高，反之越黑，代表该位置频次越低，
- 图(b) 是像素点(3，3)的邻域直方图，它在 0.5 处比较白。
- 图(c) 是像素点(40, 40)的邻域直方图，它在 1.5 处比较白。
- 图(d) 是像素点(24,24)的邻域直方图，它在 0.5 和 1.5 处都比较白。
- 以上结果都与我们之前的猜想相对应~

至此，第二问对**局部直方图**计算的函数完成！第一题也完成。

2. 图像的对比度拉伸——piecewise_linear_transform()

这一道题需要实现图像对比度拉伸的分段线性变换函数（下图）。代码应该读取一个图像；对于所有像素点数值，使用以下函数计算新的灰度值；最后用新的灰度值输出/保存图像。



参数：

- pData: 输入的图片数据
- L: 图片转换时像素范围的参数
- 返回值: 转换后的图片

具体实现：

```python
def piecewise_linear_transform(pData, L):
    """
    :param pData: input pic should be the form of np.array
    :param L: the transformation range parameter
    :return: a new pic after transformation
    """
    def single_op(p):
        if 0 <= p < L*3/8:
            return p/3
        elif L*3/8 <= p < L*5/8:
            return 3*p-L
        elif L*5/8 <= p <= L-1:
            return (p+2*L)/3

    newData = pData.copy()
    x_length, y_length = pData.shape
    for x in range(x_length):
        for y in range(y_length):
            newData[x, y] = single_op(pData[x, y])

    return newData
```

以上是这个函数的主要全部内容，具体解释如下：

- 首先定义了一个 single_op() 函数，用来对每个像素点的灰度值进行转化。
- 然后对原始图片进行 copy，并求出他们的长度和宽度。
- 接下来对每一个像素点，利用之前定义的 single_op()函数，更新他们的灰度值。
- 循环结束后，返回新的图片。

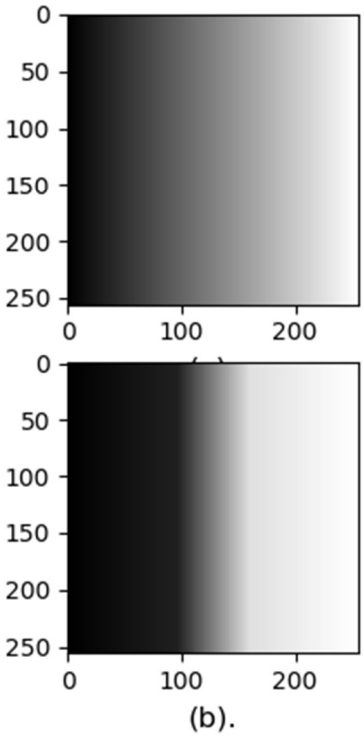
函数测试：

```python
if __name__ == "__main__":
    import matplotlib.pyplot as plt
    data = np.array([np.arange(96, 156) for _ in range(256)])
    data = np.c_[data, np.array([np.arange(156, 256) for _ in range(256)])]
    data = np.c_[np.array([np.arange(0, 96) for _ in range(256)]), data]
    print(data)
    plt.subplot(211)
    plt.imshow(data, cmap=plt.get_cmap('gray'))
    plt.title("(a).", y=-0.3)
    new_data = piecewise_linear_transform(data, 256)
    plt.subplot(212)
    plt.imshow(new_data, cmap=plt.get_cmap('gray'))
    plt.title("(b).", y=-0.3)
    print(new_data)
    plt.savefig("problem2.png")
    plt.show()
```

在以上代码中：

- 首先我们生成了一个 256*256 的纵向了由 0 到 255 逐渐增加的矩阵，因此它的可视化图片应该是渐变的。
- 然后我们计算它进行对比度拉伸后的图片，并展示出来，因为我们定义的对比度拉伸函数，会将 3L/8 到 5L/8 之间的灰度值拉伸至 1L/8 到 7L/8，因此我们会看到中间的渐变层会变窄，两边的渐变层会变宽。

结果展示：



(b).

解释：

- 上面一张图是我们定义的一张纵向的从 0 到 255 渐变的一张图像。
- 下面的这张图则是将第一幅图进行对比度拉伸后生成的图像，它的中间的渐变层变窄了，两边的渐变层变宽了，整个图像的对比度更加明显了。符合我们之前的猜测。

至此，第二题**对比度拉伸**计算的函数完成！

3. 局部直方图均衡化

这道题需要完成两个函数：

(1). 直方图均衡化——histogram_equalization ()

参数：

- pData: 输入的图像数据
- bins：直方图的柱子数量
- intensityMin: 直方图覆盖范围的最小值
- intensityMac: 直方图覆盖范围的最大值
- 返回值: 均衡化后的图像

具体实现：

```
def histogram_equalization(pData, bins=256, intensityMin=0, intensityMax=255):
    flatten_pic = pData.reshape((1, -1))
    histogram = evaluate_histogram(flatten_pic, flatten_pic.size, 1, bins, intensityMin, intensityMax)
    cdf = histogram.cumsum()
    cdf = cdf*255/cdf[-1]
    new_pic = np.interp(flatten_pic, np.linspace(intensityMin, intensityMax, bins), cdf)
    return new_pic.reshape(pData.shape)
```

- 首先将图片转换成 evaluate_histogram() 函数标准输入格式，即 array[D][M]的形式
- 然后用 evaluate_histogram() 函数计算图片的直方图
- 利用 cumsum() 函数计算直方图的累积分布频次
- 然后将累积分布频次映射到[0,255]的范围
- 再利用 numpy 中定义好的 np.interp 函数将之前图片的每个像素点的灰度值根据 cdf 映射得到均衡化后的值。
- 将均衡化后的 array 按照原始图片的 shape 输出。

(2). 高效局部直方图均衡化——efficient_local_hist_eq()

参数：

- pData: 输入的图像数据
- window_size: 局部直方图的窗口大小
- bins：直方图的柱子数量
- intensityMin: 直方图覆盖范围的最小值
- intensityMac: 直方图覆盖范围的最大值
- padding: 对边界进行填充的参数，如果不需要填充则设置为 None
- 返回值: 均衡化后的图像

具体实现：

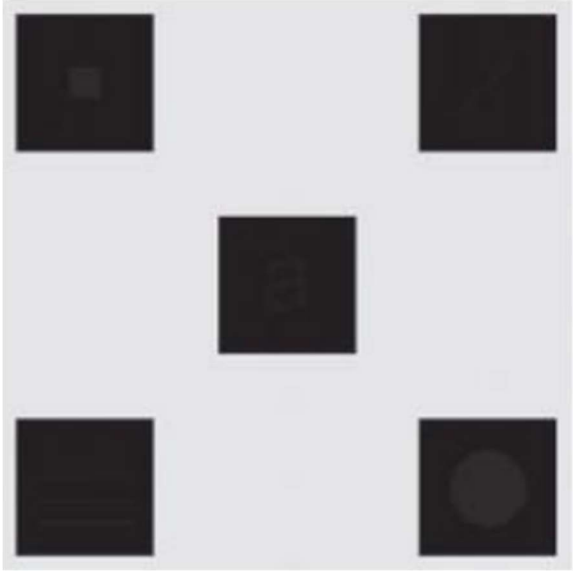
```
def efficient_local_hist_eq(pData, window_size=5, bins=256, intensityMin=0, intensityMax=255, padding=None):
    local_hists = local_histograms(pData, window_size, bins, intensityMin, intensityMax, padding)
    new_pic = np.zeros(pData.shape)
    for pos in local_hists.keys():
        x, y = pos
        hist = local_hists[pos]
        cdf = hist.cumsum()
        cdf = cdf*255/cdf[-1]
        new_pic[x, y] = np.interp(pData[x, y], np.linspace(intensityMin, intensityMax, bins), cdf)

    return new_pic
```

- 首先直接利用此前定义的 local_histograms() 函数，生成图片中每个像素点对应的邻域直方图
- 然后创建一个与原图相同大小的空矩阵
- 然后对于每一个像素点
- 获得他们邻域直方图
- 利用 histogram_equalization() 函数相同的方式计算直方图累积频次分布，并映射到[0,255]的范围，再将该像素点的灰度值利用 np.interp 函数根据 cdf 映射得到均衡化后的灰度值
- 循环结束后，输出新的图片。

函数测试：

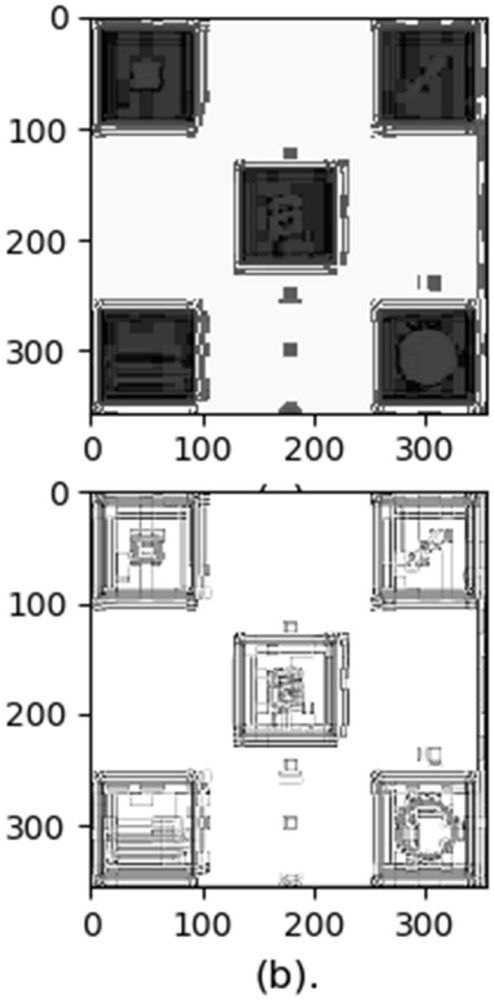
```
if __name__ == "__main__":
    import matplotlib.pyplot as plt
    from PIL import Image
    img = np.array(Image.open("problem3.png").convert("L"))
    img1 = histogram_equalization(img)
    plt.subplot(211)
    plt.imshow(img1, cmap=plt.get_cmap('gray'))
    plt.title("(a).", y=-0.3)
    img2 = efficient_local_hist_eq(img)
    plt.subplot(212)
    plt.imshow(img2, cmap=plt.get_cmap('gray'))
    plt.title("(b).", y=-0.3)
    plt.savefig("problem3-1.png")
    plt.show()
```

- 在这里，我选择截取了课件 ppt 上的图片作为这道题的数据，因为是直接从 ppt 上截的图，所以图片可能会比较模糊。图片如下：



- 我们用 PIL 库将这幅图片读入到 img 中，并转化为灰度图。
- 然后将它进行完全直方图均衡化，并可视化展示出来。
- 后面又将它进行局部直方图均衡化，并可视化出来。
- 由于这幅图片整体像素灰度值整体比较均衡，所以完全直方图均衡化可能效果并不明显，而局部直方图均衡化效果应该更加明显。

结果展示：



解释:

- 由于这个图片是直接在 ppt 上截下来的，因此上面会有很多噪音造成图案不清楚。
- 但是我们可以看到，在第一幅图完全直方图均衡化中，4 个角落的正方形整体仍然是偏黑色的，内部的图形仍然不清晰。
- 而在第二幅图局部直方图均衡化中，4 个角落的正方形的线条与第一幅图相比已经更加明显，说明对比度增强。

至此，第三题的**基于高效局部更新的局部直方图均衡化**的函数代码已经完成!
第一次作业也顺利完成!