图像补全 实验报告

黄立言 2019011329

本次实验中，为了减少计算时间和内存消耗，对图像分辨率做了降低。

算法思路和步骤

本次实验算法主要分为三部分，寻找匹配位置，graph-cut计算边界，泊松融合。下面在分别简述三部分的视线原则，代码分析放在报告的后面。

1. 最佳匹配。

   本次实验的数据分为三部分：原图像，mask图片，候选图像，同时我们需要首先通过mask将原图像上对应的部分涂黑，得到一张待补全的图像。

然后我们通过opencv的dilatation功能，拓展待补全图像，得到的事我们用于匹配的kernel

根据ppt上给出的算法，这里需要将其在候选图像上平移，找到最佳的匹配位置。这个过程非常类似我们所谓的卷积计算，因此这里通过jittor的卷积计算，得到误差矩阵，然后找到误差最小的位置，其坐标就是在候选图像上的offset。这样我们就拿到了最小误差的匹配位置。

2. graph-cut计算融合边界。

   这部分的核心是，将上一部的kernel图像进行分割，得到融合边界，一侧我们仍旧用原始图像，另一侧我们用候选图像填充。如何找出这个边界就是关键。这里根据ppt和课上提到的graph-cut算法实现，即对kernel上的像素进行建图（当然只考虑有效的像素点），每个像素和其上下左右的像素点相连，边权值计算如下：

$$M(s,t,\mathbf{A},\mathbf{B}) = \|\mathbf{A}(s) - \mathbf{B}(s)\| + \|\mathbf{A}(t) - \mathbf{B}(t)\|$$

   同时将原、新图作为source和dest点加入图中，相关的边权设置为INF。对于这个图，我们求解其最小割，得到的划分就是我们需要的解。这里我是通过networkx这个库提供的接口实现的graph-cut，具体实现见后续代码的部分。

   找到了划分边界，就可以将两个图像拼接在一起了：

3. 泊松融合

   可以看到，上图的拼接结果还是有明显的不足，可以看出明显的拼接痕迹，因此需要通过泊松融合将拼接过程变得自然。具体来说就是根据被拼接的部分的像素构建方程Ax=b，然后通过迭代法求解。这里采用的是雅可比迭代法，迭代过程中的矩阵计算则是采用了jsparse库来实现

可以看到融合后效果好了不少。

算法实现

下面分别简述代码实现。

0. 图像分辨率调整

为了避免耗费过大的内存和时间，这里对实验的图像分辨率做了降低调整，采用cv2库的pyrDown方法：

```
import cv2
down_input1 = cv2.imread("../数据/input1.jpg")
down_input1 = cv2.pyrDown(down_input1)
```

1. 匹配计算

这部分代码分为几部分：

首先我们需要将masked图像上涂黑的部分覆盖到原图像上。这部分其实就是将两张图读入，然后将原图对应位置置黑即可

```
def read_and_mask(filepath, maskpath, targetpath):
    src = np.asarray(Image.open(filepath))[...,:3]
    mask = np.asarray(Image.open(maskpath))[...,:3]
    shape = src.shape
```

```
    raw = shape[0]
    col = shape[1]
    output = np.zeros((raw, col, 3))
    for i in range(raw):
        for j in range(col):
            ori_rgb = src[i][j]
            mask_rgb = mask[i][j]
            if mask_rgb[0] < 20 and mask_rgb[1] < 20 and mask_rgb[2] < 20: # 稍微松弛了一
下mask图像上黑色的判定
                output[i][j] = [0, 0, 0]
            else:
                output[i][j] = ori_rgb
    Image.fromarray((output).astype(np.uint8)).save(targetpath)
```

得到被mask后的图像，也即待补全图像后，首先需要拓展缺失区域，得到kernel。

```
def get_kernel(masked_path):
    pic = np.asarray(Image.open(masked_path))[...,:3]
    shape = pic.shape
    raw = shape[0]
    col = shape[1]
    output = np.zeros((raw, col, 3))
# 1. 求出黑色masked区域的边界
    black_queue = deque()
    for i in range(raw):
        for j in range(col):
            if pic[i][j][0] == 0 and pic[i][j][1] == 0 and pic[i][j][2] == 0:
                black_queue.append((i, j))
    edge_black_queue = deque()
    for xy in black_queue:
        x, y = xy
        if x + 1 < raw and not (pic[x+1][y][0] == 0 and pic[x+1][y][1] == 0 and
pic[x+1][y][2] == 0):
            edge_black_queue.append((x, y))
        elif x - 1 >= 0 and not (pic[x-1][y][0] == 0 and pic[x-1][y][1] == 0 and pic[x-
1][y][2] == 0):
            edge_black_queue.append((x, y))
        elif y + 1 < col and not (pic[x][y+1][0] == 0 and pic[x][y+1][1] == 0 and
pic[x][y+1][2] == 0):
            edge_black_queue.append((x, y))
        elif y - 1 >= 0 and not (pic[x][y-1][0] == 0 and pic[x][y-1][1] == 0 and pic[x]
[y-1][2] == 0):
            edge_black_queue.append((x, y))

    for xy in edge_black_queue:
        x, y = xy
        output[x][y] = [255, 255, 255]
    Image.fromarray((output).astype(np.uint8)).save("tmp.jpg")
# 2. 对这个边界做dilatation扩展
```

```
    dilatation("tmp.jpg")
# 3. dilatation只得到了对应的区域，这里我们将对应区域在原图上的颜色拿到。
    pic_b = np.zeros((raw, col, 3))
    tmp = np.asarray(Image.open("xxx.jpg"))[...,:3]
    for i in range(raw):
        for j in range(col):
            if (tmp[i][j] >= [200, 200, 200]).all():
                pic_b[i][j] = pic[i][j]
# 4. 得到一个紧的kernel边界。
    up = -1
    down = raw
    left = -1
    right = col
    for r in range(raw):
        if up != -1:
            break
        for c in range(col):
            if pic_b[r][c][0] > 20 and pic_b[r][c][1] > 20 and pic_b[r][c][2] > 20:
                up = max(r-1, 0)
                break
    for r in range(raw):
        if down != raw:
            break
        for c in range(col):
            if pic_b[raw-1-r][c][0] > 20 and pic_b[raw-1-r][c][1] > 20 and pic_b[raw-1-
r][c][2] > 20:
                down = min(raw-r, raw-1)
                break

    for c in range(col):
        if left != -1:
            break
        for r in range(raw):
            if pic_b[r][c][0] > 20 and pic_b[r][c][1] > 20 and pic_b[r][c][2] > 20:
                left = max(c-1, 0)
                break
    for c in range(col):
        if right != col:
            break
        for r in range(raw):
            if pic_b[r][col-1-c][0] > 20 and pic_b[r][col-1-c][1] > 20 and pic_b[r]
[col-1-c][2] > 20:
                right = min(col-c, col-1)
                break
    kernel = np.zeros((down - up + 1, right - left + 1, 3))
    print(kernel.shape)
    for r in range(up, down + 1):
        for c in range(left, right + 1):
            kernel[r - up][c - left] = pic_b[r][c]
```

```
    Image.fromarray((kernel).astype(np.uint8)).save("kernel.jpg")
    return kernel, up, left
```

代码如上所述，有点长，但主要可以分为4部分。其一：求出黑色masked区域的边界上的点，用于后续的膨胀扩展。其二，调用通过opencv实现的dilatation函数，得到扩展后的区域。其三，由于扩展的时候，边界上的点只是白色的，因此还需要对于这部分区域，填充上原图的颜色，其余部分当然是置黑，这样我们其实已经拿到了kernel。其四：为了后续求卷积，我们还需要将上一步得到的结果，划分出来kernel实际的边界，得到一个紧的kernel。这里还返回了上、左两个边界，这是kernel在原图中的offset，在后续的计算中会被用左kernel坐标和原图坐标的转换。

dilatation的实现如下：

```
def dilatation(image):
    src = cv.imread(cv.samples.findFile(image))
    if src is None:
        print('Could not open or find the image: ', image)
        exit(0)
    dilatation_size = 16
    dilation_shape = cv.MORPH_ELLIPSE
    # print("which?{}".format(cv.getTrackbarPos(title_trackbar_element_shape,
title_dilation_window)))
    element = cv.getStructuringElement(dilation_shape, (2 * dilatation_size + 1, 2 *
dilatation_size + 1),
                                       (dilatation_size, dilatation_size))
    dilatation_dst = cv.dilate(src, element)
    # print(dilatation_dst.shape)
    Image.fromarray((dilatation_dst).astype(np.uint8)).save("xxx.jpg")
```

通过opencv的dilatation实现，参考了opencv提供的参考代码。

得到了kernel，下一步就是计算卷积了。这里参考了计图官方给出的卷积计算的代码，实现如下：

```
def calcConvJittor(A, kernel):
    a_r = A.shape[0]
    a_c = A.shape[1]
    k_r = kernel.shape[0]
    k_c = kernel.shape[1]
    offset_r_max = a_r - k_r + 1
    offset_c_max = a_c - k_c + 1
    y = np.zeros([a_r - k_r + 1, a_c - k_c + 1, 1])
    # 直接遍历计算卷积，帮助得到reindex里i的配置。
    # for i0 in range(offset_r_max):
    #     for i1 in range(offset_c_max):
    #         for i2 in range(k_r):
    #             for i3 in range(k_c):
    #                 for i4 in range(3):
    #                     # print(kernel[i2, i3, i4])
    #                     y[i0, i1] += (kernel[i2, i3, i4] - A[i0 + i2, i1 + i3,
i4])**2 # 下面的reindex的配置，参考这里
```

```
    aa = A.reindex([offset_r_max, offset_c_max, k_r, k_c, 3], [
        'i0+i2',
        'i1+i3',
        'i4'
    ])
    kk = kernel.broadcast_var(aa)
    yy = (kk-aa)*(kk-aa)*(kk!=0) # 黑色部分不算
    y = yy.sum([2, 3, 4])
    return y
```

这里A.reindex里i的设置，是我先通过遍历直接计算卷积后得到的结果，可以从上面代码中的注释部分中看到。

得到的误差矩阵，我们还需要找到最小的位置，这就是kernel在候选图像中的offset。

```
y_row = y.shape[0]
y_col = y.shape[1]
min = 10000000000
offset_r = -1
offset_c = -1
for r in range(y_row):
    for c in range(y_col):
        if y[r][c] < min:
            min = y[r][c]
            offset_r = r
            offset_c = c
```

如此，我们就得到了最佳的匹配位置。

2. graph-cut融合边界

这步就是要对上一步得到的kernel做一个划分，分别分给原图和新图。本算法的核心在于两点：建图+求最小割。

```
def buildGraph(offset_r, offset_c, ori_offset_r, ori_offset_c, input_path, result_path,
kernel, vertical=True):
    input_pic = np.asarray(Image.open(input_path))[...,:3]
    result_pic = np.asarray(Image.open(result_path))[...,:3]
    row = kernel.shape[0]
    col = kernel.shape[1]
    for r in range(row):
        for c in range(col):
            if isBlack(r, c, kernel):
                continue
            graph.add_node((r, c))
            if vertical:
                if (r-1 >=0 and isBlack(r-1, c, kernel)) or r == 0:
                    graph.add_edge((-1, -1), (r, c), capacity=np.inf)
                elif (r + 1 < row and isBlack(r+1, c, kernel)) or r + 1 == row:
                    graph.add_edge((-2, -2), (r, c), capacity=np.inf)
            else:
```

```
                if (c-1 >= 0 and isBlack(r, c-1, kernel)) or c == 0:
                    graph.add_edge((-1, -1), (r, c), capacity=np.inf)
                elif (c+1 < col and isBlack(r, c+1, kernel)) or c + 1 == col:
                    graph.add_edge((-2, -2), (r, c), capacity=np.inf)

    for key in graph:
        if key == (-1, -1) or key == (-2, -2):
            continue
        r, c = key
        if (r+1, c) in graph:
            graph.add_edge((r, c), (r+1, c), capacity=calcM(r, c, r+1, c, ori_offset_r,
ori_offset_c, offset_r, offset_c, input_pic, result_pic))
        if (r-1, c) in graph:
            graph.add_edge((r, c), (r-1, c), capacity=calcM(r, c, r-1, c, ori_offset_r,
ori_offset_c, offset_r, offset_c, input_pic, result_pic))
        if (r, c+1) in graph:
            graph.add_edge((r, c), (r, c+1), capacity=calcM(r, c, r, c+1, ori_offset_r,
ori_offset_c, offset_r, offset_c, input_pic, result_pic))
        if (r, c-1) in graph:
            graph.add_edge((r, c), (r, c-1), capacity=calcM(r, c, r, c-1, ori_offset_r,
ori_offset_c, offset_r, offset_c, input_pic, result_pic))
    cut_value, partition = nx.minimum_cut(graph, (-1, -1), (-2, -2))
    reachable, unreachable = partition
    print(reachable.__len__(), unreachable.__len__())
    return partition
```

这里的图是利用了networkx这个库作为辅助，以每个图的坐标为节点建图，并且规定source、dest节点对应-1，-1和-2，-2。计算边权的方式就是ppt上给出的公式，可以看到这里通过kernel坐标加上在原图/候选图的offset找到对应的坐标。最后通过networkx的minimum_cut方法直接求出最小割，返回的是分别和source、dest对应在一侧的点集，其实就是我们需要分别染成原图和候选图颜色的点集，后面直接用对应的坐标即可。

3. 自然融合

这里核心在于两点：其一，构建方程中的A和b，其二，解方程。

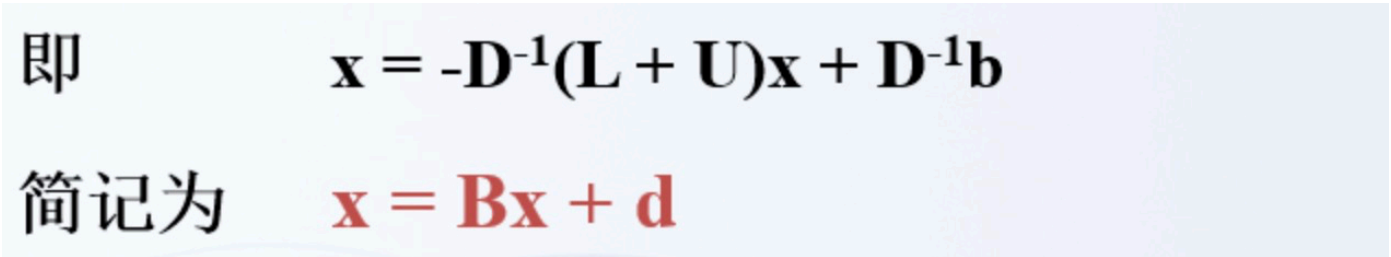
```
# 其中res_list表示原本mask的区域+graph-cut分给候选图像的区域。其实就是理应由候选图像占据的所有像素。
def calcMatrixA(res_list: list):
    N = res_list.__len__()
    MaxtrixA = np.zeros((N, N, ))
    for i in range(N):
        for j in range(N):
            if i == j:
                MaxtrixA[i][j] = 4
            elif adjacent(res_list[i][0], res_list[i][1], res_list[j][0], res_list[j]
[1]):
                MaxtrixA[i][j] = -1
            else:
                MaxtrixA[i][j] = 0
```

```
    return MaxtrixA

def calcB(res_list, source_pic, target_pic, source_offset_r, source_offset_c,
target_offset_r, target_offset_c):
    N = res_list.__len__()
    ret_r = []
    ret_g = []
    ret_b = []
    for i in range(N):
        x, y = res_list[i]
        row = source_pic.shape[0]
        col = source_pic.shape[1]
        rgb = target_pic[x-source_offset_r+target_offset_r][y-
source_offset_c+target_offset_c].astype(np.int32)
        adj_list = []
        if x+1 < row:
            adj_list.append((x+1, y))
        if x-1 >= 0:
            adj_list.append((x-1, y))
        if y+1 < col:
            adj_list.append((x, y+1))
        if y-1 >= 0:
            adj_list.append((x, y-1))
        tmp = 4 * rgb
        for ab in adj_list:
            a, b = ab
            tmp -= target_pic[a-source_offset_r+target_offset_r][b-
source_offset_c+target_offset_c].astype(np.int32)
            if (a, b) not in res_list:
                tmp += source_pic[a][b].astype(np.int32)
        ret_r.append(tmp[0])
        ret_g.append(tmp[1])
        ret_b.append(tmp[2])
    return ret_r, ret_g, ret_b # 返回三个颜色通道的b向量
```

上面的部分即A和b的构建，其实没有太多复杂的地方。个人觉得需要注意的点在于b的构建。这里一开始理解有点问题，当成了在graph-cut得到的图像上直接应用公式4x-adjacent(x)，经过和同学讨论才发现不是这样的，应该是4*target(x) - adjacent(x) + source(x) if x is not in masked，就是对于候选图像计算后，如果有邻居节点不在graph-cut分出的候选图像边界内，则还需要加上原图的像素。

如此我们就得到了A和b，下面只需要解Ax=b即可。这里我采用的是雅可比迭代法，其中涉及矩阵计算的部分通过jsparse实现。雅可比迭代法的表示如下：



其中D、L、U分别为A的对角线、上半三角、下半三角。这里虽然涉及矩阵乘法，但是其实只需要乘一次，对整体时间复杂度没啥影响。

矩阵求解的部分如下，其中jsparse的使用参考给出的jsparse教程，用于计算上面雅可比迭代的Bx这一次乘法：

```
def solveMatrix(A, b):
    row = A.shape[0]
    D_inv = np.zeros((row, row, ))
    L = np.zeros((row, row, ))
    U = np.zeros((row, row, ))
    for i in range(row):
        D_inv[i][i] = 1 / A[i][i]
    for i in range(row):
        for j in range(row):
            if i > j:
                L[i][j] = A[i][j]
            elif i < j:
                U[i][j] = A[i][j]
    B = np.dot(-D_inv, (L + U))
    b = b.reshape(b.shape[0], 1)
    d = np.dot(D_inv, b)
    d = d.reshape(d.shape[0], 1)
    vec = np.random.rand(row, 1).astype(np.float32)
    indices = np.nonzero(B)
    values = B[indices]
    while True:
        output = F.spmm(
            rows=jt.array(indices[0]),
            cols=jt.array(indices[1]),
            vals=jt.array(values),
            size=(row, row),
            mat=jt.array(vec),
            cuda_spmm_alg=1)
        vec1 = output + jt.array(d)
        if ((vec - vec1)**2 == 0).all():
            break
        vec = vec1
    return vec.numpy().astype(np.float32)
```

即根据雅可比迭代的公式循环计算，直到收敛。得到的解就是原区域内融合后的r/g/b值，分别求解后组合，即为我们需要的像素值。