

수치해석

HW #9

FDCT/IDCT

2018007956 김채아

[DCT/IDCT 과정]

1. 이미지를 R,G,B 세 개의 채널로 분리한다
2. 16x16사이즈의 블록들로 나누고
3. 각 블록들마다 DCT를 한다
4. 값이 큰 16개를 고른다 (양자화 과정을 간단하게 구현)
5. linear combination(IDCT)를 한다
6. 16x16 이미지들을 조합하고
전체 원본 이미지를 만든다
7. R,G,B를 병합한다

```
def select(orig):
    s = orig.shape
    orig = orig.flatten()
    absorig = np.abs(orig)
    # 절댓값이 최대인 값 16개 찾기 -> 16개 추출 후 새로 생성
    select_orig = np.zeros(orig.shape)
    for _ in range(16):
        m=absorig.argmax()
        select_orig[m]=orig[m] : energy compaction
        absorig[m]=0

    select_orig = select_orig.reshape(s)
    return select_orig

def blocking(orig,x,y):
    return orig[16*x: 16*(x+1),16*y: 16*(y+1)]
```

```
img = cv2.imread("C:\\Users\\LG\\Desktop\\1.jpg").astype('float32')
r,g,b=cv2.split(img)

A=[r,g,b]
B=[]
for i in range(3):
    c = []
    for y in range(b.shape[1] // 16):
        r = []
        for x in range(b.shape[0] // 16):
            block = blocking(A[i], x, y)
            bdct = cv2.dct(block)
            select_dct=select(bdct)
            idct = cv2.idct(select_dct).astype('int')
            if len(r)==0:
                r=idct
            else:
                r=np.vstack([r,idct])
            if len(c)==0:
                c=r
            else:
                c=np.hstack([c,r])
        B.append(c) # r,g,b
im = cv2.merge((B[0],B[1],B[2]))
print(img-im)
cv2.imwrite("C:\\Users\\LG\\Desktop\\11.png",im)
```

이미지를 블록들로 나눔
각 블록들마다 DCT
16개를 고른다
적은 coefficients로 IDCT
전체 이미지로 합침
RGB 색 합침
원본과의 차이를 보여줌 (이걸 표현하기 위해 원본 이미지 사이즈를 16의 배수로 맞췄다)

[image 1] 인물 사진

원본



16개의 coefficients로만 구성된 이미지



오른쪽 사진의 모자와 눈 부분을 자세히 보면 16x16 block들이 보인다.
원본과의 차이를 크게 느낄 수 없다

`print(img-im)` 의 일부

(두 이미지의 차이)

```
[[[-2.  0. -5.]  
  [-1.  1. -4.]  
  [ 0.  2. -3.]  
  ...  
  [ 1.  1.  1.]  
  [ 1.  1.  1.]  
  [ 1.  1.  1.]]
```

```
[[[-2.  0. -5.]  
  [ 0.  2. -3.]  
  [ 1.  3. -2.]  
  ...  
  [ 1.  1.  1.]  
  [ 1.  1.  1.]  
  [ 1.  1.  1.]]
```

```
[[[-1.  0. -2.]  
  [ 0.  1. -1.]  
  [ 1.  2.  0.]  
  ...  
  [ 1.  1.  1.]  
  [ 1.  1.  1.]  
  [ 1.  1.  1.]]
```

ideal하게 모두 0이진 않지만
이런 무작위한 랜덤 노이즈는
육안으로 볼 수 없다

[image 2] 복잡한 영상 구조를 가진 풍경 사진



가로등, 뒤쪽의 건물, 작은 차들까지는 구현하지 못한다
DCT를 하고 큰 값 16개를 선택하게 되면 낮은 frequency에 해당하는 계수들이 남게 되므로
high frequency 계수를 많이 가진 복잡한 영상구조는 정확히 복원해내지 못한다

[image 3] 간단한 영상 구조를 가진 사진



평평한 면이 많고, 복잡한 구조를 가지지 않은 간단한 이미지이다
앞 두 사진보다 원본 사진과의 차이가 더 작음을 볼 수 있다
간단한 사진은 적은 coefficients로도 거의 완벽하게 구현해 낼 수 있다

[FDCT/IDCT 직접 구현]

```
def C(u):
    if u==0:
        return 1/np.sqrt(2)
    else:
        return 1

def DCT(S):
    F = np.zeros((16,16))
    for v in range(16):
        for u in range(16):
            sum=0
            for y in range(16):
                for x in range(16):
                    sum += S[y][x] * np.cos((2*y+1) * v * np.pi / 32) * np.cos((2*x+1) * u * np.pi / 32)
            F[v][u] = 1/8 * C(v) * C(u) * sum
    return F

def IDCT(F):
    S = np.zeros((16,16))
    for y in range(16):
        for x in range(16):
            sum=0
            for v in range(16):
                for u in range(16):
                    sum += C(u) * C(v) * F[v][u] * np.cos((2*y+1) * v * np.pi / 32) * np.cos((2*x+1) * u * np.pi / 32)
            S[y][x] = 1/8 * sum
    return S.astype(np.int)
```

■ 8x8 block 2D DCT (64 coefficients)

16x16

- Efficient energy compaction
- Blocking artifact at low bit rates
- Fast FDCT/IDCT algorithms exist

$$F_{vu} = \frac{1}{8} C_v C_u \sum_{y=0}^{N-1} \sum_{x=0}^{N-1} S_{yx} \cos\left(v\pi \frac{2y+1}{2N}\right) \cos\left(u\pi \frac{2x+1}{2N}\right)$$

$$S_{yx} = \frac{1}{8} \sum_{v=0}^{N-1} \sum_{u=0}^{N-1} C_v C_u F_{vu} \cos\left(v\pi \frac{2y+1}{2N}\right) \cos\left(u\pi \frac{2x+1}{2N}\right)$$

$$C_u = \begin{cases} \frac{1}{\sqrt{2}} & \text{if } u = 0 \\ 1 & \text{else} \end{cases} \quad C_v = \begin{cases} \frac{1}{\sqrt{2}} & \text{if } v = 0 \\ 1 & \text{else} \end{cases}$$

[FDCT/IDCT 직접 구현 - image 1]

원본



16개의 coefficients로만 구성된 이미지



`print(img-im)` 의 일부

(두 이미지의 차이)

```
[[[ 0.  0.  0.]  
   [ 0.  0.  0.]  
   [ 0.  0.  1.]  
   ...  
   [ 1.  1.  1.]  
   [ 1.  1.  1.]  
   [ 1.  1.  1.]]  
  
[[[ 0.  0. -1.]  
   [ 1.  1.  0.]  
   [ 1.  1.  1.]  
   ...  
   [ 1.  1.  1.]  
   [ 1.  1.  1.]  
   [ 1.  1.  1.]]  
  
[[[ 1.  0.  1.]  
   [ 0.  0.  1.]  
   [ 0.  0.  1.]  
   ...  
   [ 0.  0.  0.]  
   [ 0.  0.  0.]  
   [ 0.  0.  0.]]
```

FDCT/IDCT 각각 6중 for문을 사용하였고 RGB에 해당하는 3중 for문까지 해서, 사이즈가 큰 이미지는 코드 작동 시간이 너무 오래 걸렸다
(이 이미지는 4191.383754730225초(약 69분) 걸림)

라이브러리를 사용했을 때와 거의 같은 결과를 보여준다