DCT 워터마킹 실험 및 분석

20215278 / 정승재

1 통계 분석

1-1 워터마킹 삽입

첫 번째 실험에서는 Lena 이미지를 DCT 변환한 뒤, seed 0부터 5000까지 각각 다른 시드 값을 사용해 워터마크를 삽입하였다. 이때 하나의 DCT coefficient에 5000번 반복 삽입한 것이 아니라, 각 시드마다 독립적으로 한 번씩 워터마킹을 수행하였다. 이번 실험의 목적은 워터마크가 삽입된 이미지를 시각적으로 확인하는 것이 아니라, 워터마크가 삽입된 시퀀스와 삽입되지 않은 시퀀스 간의 통계적 특성을 분석하는 것이다. 따라서 이후 내적 연산을 편리하게 수행하기 위해, DCT 전체 맵 5000개를 그대로 반환하는 대신 중주파수 대역의 16,000개 계수만 선택하여 flatten 한 결과를 반환하였다. 즉, 최종적으로 (5000, 16000) 형태의 데이터를 얻었으며, 이는 seed 0부터 5000까지 삽입된 워터마킹 결과를 의미한다.

```python
def watermarking_detection(img,seed):
    dct_img = dct2(img)
    # 워터마크 심은 이미지에서 DCT 계수 가져옴
    mid_frequency_index = zigzag_choice()
    real_watermarking_sequence = []
    for i,j in mid_frequency_index:
        real_watermarking_sequence.append(dct_img[i][j])
    real_watermarking_sequence = np.array(real_watermarking_sequence)  # 16000,1



    np.random.seed(seed=seed)
    random_seed = np.random.normal(0,1,16000)
    z = np.matmul(random_seed, real_watermarking_sequence) / 16000

    return z
watermarked_correlation = []
for seed in range(5000):
    watermarked_correlation.append(watermarking_detection(idct_image[seed],seed))
watermarked_correlation = np.array(watermarked_correlation)
```

```python
def watermarking_casting(dct_coefficient_map,seed):
    np.random.seed(seed=seed)
    watermark = np.random.normal(0,1,16000)
    mid_frequency_index = zigzag_choice()
    watermark_index_count = 0
    alpha = 0.2
    for i,j in mid_frequency_index:
        dct_coefficient_map[i][j] = dct_coefficient_map[i][j] + \
                                    alpha*abs(dct_coefficient_map[i][j])*watermark[watermark_index_count]
        watermark_index_count = watermark_index_count + 1
    return dct_coefficient_map
# 5000까지 시드 넣어서 각각을 저장. 총 5000개의 시퀀스
lst = []
for i in range(5000):
    dct_copy = np.copy(dct_coefficient_map)
    lst.append(watermarking_casting(dct_copy,i))
lst = np.array(lst) # shape : 5000,512,512
```

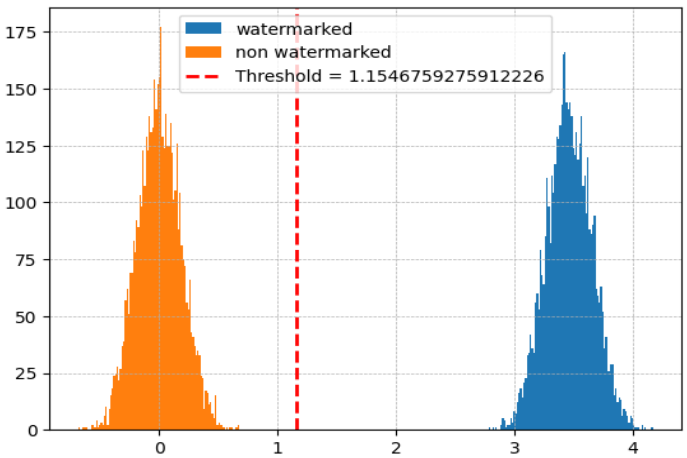
1-2 워터마킹 탐지

탐지 단계에서는 seed 5000부터 10000까지의 구간을 사용하여 워터마킹이 삽입되지 않은 시드 세트를 생성하였다. 함수 내부에서는 두 경우를 구분하여 처리하였으며, 워터마크를 삽입한 경우의 correlation 값 5000개와, 삽입하지 않은 경우의 correlation 값 5000개를 각각 별도로 반환하도록 구성하였다.

```python
non_watermarked_correlation = []
seed = 5000
for i in range(5000):
    non_watermarked_correlation.append(watermarking_detection(idct_image[i],seed))
    seed = seed + 1
non_watermarked_correlation = np.array(non_watermarked_correlation)
```

1-3 Correlation Histogram

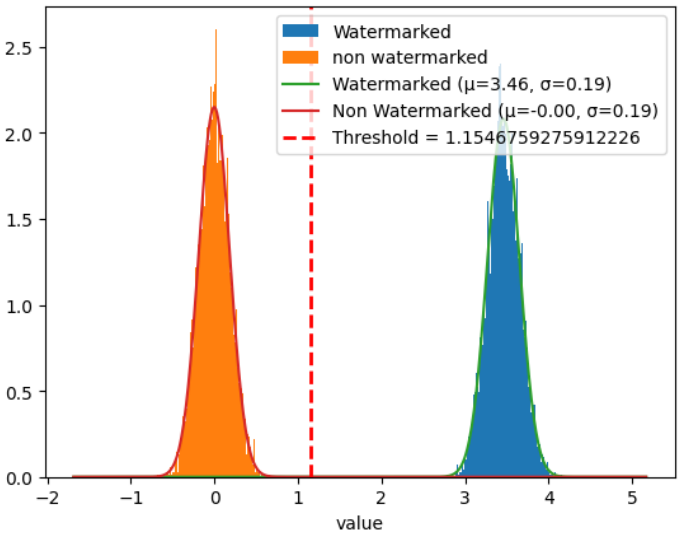
히스토그램은 앞서 언급한 워터마킹 탐지 함수의 반환값을 이용해 그렸다. 주황색 히스토그램은 워터마크가 삽입되지 않은 correlation 값, 파란색 히스토그램은 워터마크가 삽입된 correlation 값을 나타낸다. Threshold는 삽입된 5000개의 각 threshold 값을 모두 더한 뒤, 평균을 내어 하나의 기준값으로 사용하였다.



현재까지는 어떠한 공격도 가하지 않았기에, false alarm 과 missed detection의 경우가 단 하나도 존재하지 않는 분포를 띄고 있다.

2 정규분포 분석

이번 실험에서는 워터마크가 삽입된 5,000개와 삽입되지 않은 5,000개의 샘플 각각에 대해 평균과 분산을 계산하고, 이를 기반으로 가우시안 분포 곡선을 시각화하였다. 워터마크가 삽입된 분포의 평균은 약 3.46으로, 이는 DCT 계수의 절댓값과 삽입된 난수 시드의 제곱 항이 더해지기 때문에 양의 값으로 치우친 결과이다. 반면, 워터마크가 삽입되지 않은 경우에는 서로 독립적인 난수들이 상쇄되면서 평균이 0에 수렴하는 경향을 보였다. Threshold는 워터마크가 삽입된 5,000개의 개별 threshold 값들의 평균을 취하여 하나의 기준 임계값으로 사용하였다.



3 공격 실험

이미지에 공격 후 통계를 비교해보기 위해 워터마크 삽입된 이미지에 추가적으로 가우시안 노이즈를 더해준 다음 분포 비교를 해 보았다. 논문에서는 분산이 4000인것 까지 워터마크 탐지가 robust 하게 탐지 된 것을 확인했기에, 분포 비교를 조금 더 명확히 하기 위해 분산 5000의 가우시안 노이즈를 추가해 주었다.

워터마크 이미지

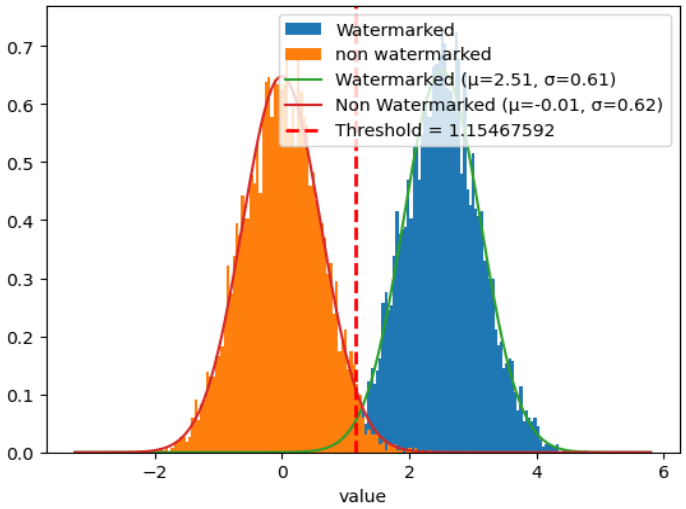
노이즈 추가된 이미지

TABLE I: 원본 이미지와 노이즈 추가 이미지 비교

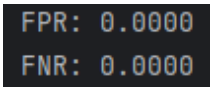
다음으로 FPR과 FNR의 의미는 다음과 같다.

- FPR
  - 워터마크가 일치하다 예측 했지만, 실제론 불일치일 경우
- FNR
  - 워터마크가 불일치하다 예측 했지만, 실제론 일치할 경우

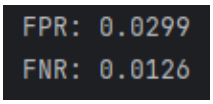
FPR은 False Alarm을 의미하며, Threshold를 기준으로 CDF를 계산한 후 그 값을 1에서 빼주어 얻는다. 이는 실제 불일치(주황색) 분포의 오른쪽 꼬리 영역에 해당한다. 반대로, FNR은 Missed Detection을 의미하며, Threshold의 CDF 값을 계산하여 구한다. 이는 실제 일치(파란색) 분포의 왼쪽 꼬리 영역에 해당한다. 가우시안 노이즈 공격을 가한 뒤, 임계치는 그대로 두고 다시 한번 히스토그램을 찍어보면 다음과 같다.



- 공격 전·후 FPR /FNR비교



Before Noise



After Noise

가우시안 노이즈를 추가한 이후, FPR과 FNR이 모두 증가하는 현상을 확인할 수 있었다. 그 원인은 다음과 같다.

먼저, FNR은 워터마크 일치 분포(파란색)의 왼쪽 꼬리 영역에 해당한다. 공격이 없는 경우, 워터마크된 이미지와 동일한 랜덤 시드를 사용하여 correlation을 계산하면, 상관도가 제곱 형태로 근사되면서 분포가 양의 방향으로 치우치는 경향을 보인다. 그러나 가우시안 노이즈가 추가되면 워터마크 신호가 원래보다 희석되어 전체 correlation이 약화되고, 그 결과 분포의 중심이 0 쪽으로 이동하여 왼쪽 꼬리 영역의 확률이 증가하게 된다. 따라서 FNR이 증가하게 된다.

한편, FPR은 불일치 분포(주황색)의 오른쪽 꼬리 영역에 해당한다. 이 분포는 원래 워터마크 신호와 가우시안 노이즈의 합으로 구성되며, 두 신호가 서로 independent하기 때문에 전체 분산은 각각의 분산의 합으로 표현된다. 따라서 평균은 0을 유지하지만 분산이 커지게 되고, 그 결과 오른쪽 꼬리 영역의 확률이 증가하여 FPR 또한 증가하게 된다.

4 멀티비트 워터마킹

32비트 워터마크를 송·수신 하기 위해 사용한 방법은 CDMA기법을 사용하였다. 구현한 알고리즘의 단계는 총 두 단계로, 비트 삽입 단계와 비트 검출 단계로 이루어 진다.

4-1 bit 삽입 알고리즘

1. 이미지 → DCT 변환

본 멀티비트 워터마킹에서는 Frequency domain의 중 주파수 16000개 영역에 비트를 숨기는 방식을 사용하기 때문에 이미지를 DCT변환하여 주파수 영역으로 변환해 준다. 중주파수 추출 알고리즘은 기존에 사용한 zigzag 알고리즘을 그대로 사용하여 중주파수를 추출 하였다.

2. bit → Bipolar mapping

```
bit = [1,0,0,1,0,1,0,0,1,1,1,0,1,0,0,0,1,0,1,0,1,1,1,1,0,1,0,1,1,0,1,0]
lst = []
for i in bit:
    if i ==1 :
        lst.append(1)
    else:
        lst.append(-1)
Bipolar_bit = lst
Bipolar_bit
```

CDMA 알고리즘은 특별한 임계치 없이 0 을 기준으로 비트를 검출하기 때문에 비트의 부호가 검출에 사용될 수 있도록 0,1 이진 비트를 -1,+1로 변환하여 주었다.

3. 비트 삽입

```
def watermarking_casting(dct_coefficient_map):
    mid_frequency_index = zigzag_choice()
    P_sequence = []
    # 각 비트별 16000개의 시퀀스 생성 . shpae: 32,16000 (랜덤 +1,-1 이 생성됨)
    for i in range(32):
        np.random.seed(i)
        P_sequence.append(np.random.choice([-1, 1], size=16000))
    P_sequence = np.array(P_sequence) # shape: 32,16000

    # 비트(+1,-1)항 각 시퀀스의 곱으로 삽입할 워터마크 생성
    for i in range(32):
        P_sequence[i, :] = P_sequence[i, :] * Bipolar_bit[i]
    # 각 시퀀스의 합으로 삽입할 워터마크 생성
    watermark = np.sum(P_sequence, axis=0)

    watermark_index_count = 0
    alpha = 0.09
    for i,j in mid_frequency_index:
        dct_coefficient_map[i][j] = dct_coefficient_map[i][j] + \
                                alpha*abs(dct_coefficient_map[i][j])*watermark[watermark_index_count]
        watermark_index_count = watermark_index_count + 1
    return dct_coefficient_map
watermarking_DCT = watermarking_casting(dct_coefficient_map)
```

CDMA에서는 부호를 이용해 비트를 검출하기 때문에, 일반적인 가우시안 노이즈 대신 +1과 -1로 구성된 랜덤 시퀀스를 사용한다. 중주파수 대역의 길이가 16,000이므로, 이에 맞춰 Pseudo Random Number의 길이도 16,000으로 설정한다.

비트의 총 길이는 32이므로 전체 시퀀스의 shape은 32×16,000이 된다. 이후 각 비트마다 해당 의사난수 시퀀스를 곱하고, 각 column의 합을 구하여 최종적으로 삽입할 워터마크를 생성한다. 이후 워터마크 삽입 과정은 이전의 DCT 워터마킹 방식과 동일하게 진행된다. 즉, 각 DCT 계수의 절댓값에 워터마크 세기 조절 파라미터인 alpha와 생성된 워터마크 값을 곱한 항을 추가함으로써 워터마크를 삽입한다. 이 과정을 통해 총 32비트의 워터마크가 삽입된 상태가 된다.

4. PSNR

TABLE II는 원본과 워터마크(비트)가 삽입된 이미지간의

Original image　　Watermarked image

TABLE II: 원본 이미지와 비트 추가된 이미지 비교

비교이다. 워터마크 삽입 강도는 0.09로 했을때가 PSNR 38 dB을 유지 하며 가장 강하게 삽입 할 수 있어서 0.09로 설정하였다.

4-2 비트 검출 알고리즘

```
def watermarking_detection(img):
    # 이미지를 DCT변환
    dct_img = dct2(img)

    # 중주파수 뽑음
    mid_frequency_index = zigzag_choice()
    real_watermarking_sequence = []
    for i,j in mid_frequency_index:
        real_watermarking_sequence.append(dct_img[i][j])
    real_watermarking_sequence = np.array(real_watermarking_sequence)

    # 시드 알고있으니 다시 생성
    P_sequence = [] # 삽입한 시드로 P시퀀스 다시 생성
    for i in range(32):
        np.random.seed(i)
        P_sequence.append(np.random.choice([-1, 1], size=16000))
    P_sequence = np.array(P_sequence)

    # correlation 계산
    detection_bit = np.matmul(P_sequence, real_watermarking_sequence)/16000
    return detection_bit

detection_bit = watermarking_detection(watermarking_image)
detection_bit
```

검출 과정은 다음과 같다. 먼저 워터마크가 삽입된 이미지를 입력받아 DCT 변환을 수행하고, 중주파수 대역의 계수를 추출한다. 이후 검출자는 삽입 시 사용된 의사난수 시드를 알고 있다고 가정하며, 이를 이용해 동일한 의사난수 시퀀스를 생성한다. 비트의 개수가 32비트이므로 총 32개의 의사난수 시퀀스를 생성한다.

다음으로, 추출한 중주파수 대역과 생성한 의사난수 시퀀스 간의 내적 연산을 수행하여 비트를 검출한다. 이때 탐지 결과는 correlation의 통계적 특성에 의해 삽입된 비트 부호가 유지된 형태로 나타나며, 내적 결과가 양수이면 비트 1, 음수이면 비트 0으로 판별하여 최종적으로 워터마크 비트를 복원한다.

4-3 탐지 결과 (BER)

```
lst = []
for i in range(32):
    if detection_bit[i] >0:
        lst.append(1)
    else:
        lst.append(0)
print(lst,end=" ")
print()
print(bit,end=" ")

✓ [15] < 10 ms

 [1, 0, 0, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 0, 0, 0, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 0, 1, 0]
 [1, 0, 0, 1, 0, 1, 0, 0, 1, 1, 1, 1, 0, 1, 0, 0, 0, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 0, 1, 0]
```

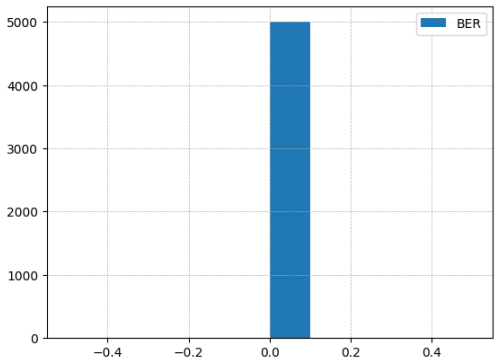
```
error_count = 0
for i in range(32):
    if ss[i] != bit[i]:
        error_count = error_count + 1
print("BER : %f" % (error_count / 32))
✓ [14] < 10 ms

 BER : 0.000000
```

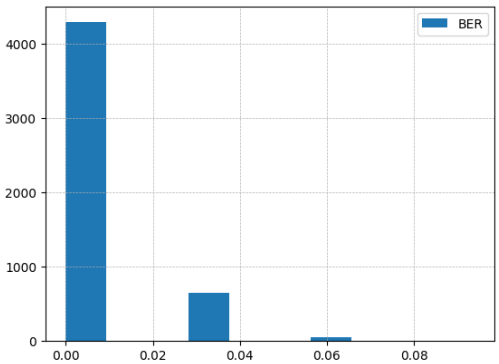
bit는 원래 내가 삽입한 비트 이고, lst는 + 이면 비트 1,- 이면 비트 0으로 다시 비트로 바꾼 list이다. 그 결과 32개 비트 모두 그대로 검출되며 BER은 0으로 높은 탐지율을 달성하였다.

4-4 다수의 샘플 BER변화

이번 실험에서는 멀티비트 워터마킹을 5000번 시뮬레이션 하고 기본 테스트 5000번, 노이즈 추가해서 5000번을 진행 하였다. (a)는 가우시안 노이즈를 추가하기 전의 분포로,



(a) Before attack



(b) After attack

TABLE III: BER 변화량 비교

서로 다른 5000회 실험 모두에서 모든 비트를 100% 정확하게 탐지하였다. 반면 (b)는 워터마크가 삽입된 이미지에 추가로 가우시안 노이즈를 더한 후의 결과를 나타낸다. 대부분의 경우 여전히 100% 탐지가 이루어졌지만, 약 5000회 중 700~800회 정도는 1~2비트의 오탐이 발생하였다. 이는 가우시안 노이즈의 추가로 인해 이미지 내 워터마크 신호가 약화되면서, 숨겨진 비트 정보가 부분적으로 희석되어 검출 정확도가 다소 저하된 것으로 해석된다.