reproducibility_challenge

- error correction code transformer

20212245 김희서

3. Result

3.1) mask

해당 논문은 'Masked Self-Attention'을 사용했습니다. 이때 'Mask'는 무언가를 가린다는 의미로 사용되고, 연구에서 사용된 decoder의 'Self-Attention Layer'는 반드시 자신보다 앞쪽에 포지션에 해당하는 'token'들의 attention score만 볼 수 있습니다.

만일 'output'들이 주어졌을 때, 뒤에 나오는 것은 볼 수 없습니다. 만일 그렇게 된다면, 기계번역을 할 경우를 예를 들면, 답안을 보고 번역하는 경우가 되기에, Masking을 구현할 때 해당 위치의 score 값을 마이너스 무한대 값으로 표기함으로써 구현합니다.

$$A_H(Q,K,V) = Softmax(\frac{QK^T + g(H)}{\sqrt{d}})V,$$

$$g(H) : 0,1^{(n-k)\times k} \rightarrow -\infty, 0^{2n-k\times 2n-k}$$

로 self-attention mechanism을 표현했습니다.

g(H)의 값이 마이너스 무한대이면, 'Softmax' 값이 0이 되고, 0일 경우에는 성능에 어떤 영향을 주지 않기에, 다음처럼 표현하였습니다. 이때 저자가 그림으로 표현한 mask의 경우, parity check matrix를 기반으로 구성하였고, 모든 parity check matrix H의 각각의 row 'i'에 대한 1의 위치를 각각 'unmask'하는 형식으로 구성하였습니다.

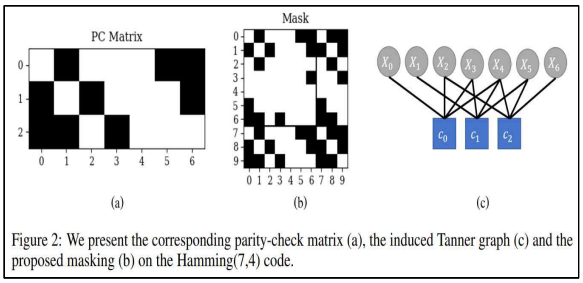


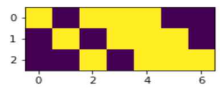
Figure 2: We present the corresponding parity-check matrix (a), the induced Tanner graph (c) and the proposed masking (b) on the Hamming(7,4) code.

위 논문의 그림에서는 '흰'색 부분이 parity check matrix의 '1'의 위치에 해당하는 부분을 의미합니다. 예를 들어, ©의 X_0와 c_0가 연결된 것은 parity check matrix의 1행 1열의 값이 '1'이 되는 것을 나타내는데, 이를 (a)에서 '흰'색으로 표현한 것을 확인할 수 있고, colab 코드에서는 '노란'색으로 표현하였습니다.

```
import numpy as np
H=np.matrix(([1,0,1,1,1,0,0],[0,1,0,1,1,1,0],[0,0,1,0,1,1,1]))
print('matirx H값 \Wn',H)

matirx H값
 [[1 0 1 1 1 0 0]
 [0 1 0 1 1 1 0]
 [0 0 1 0 1 1 1]]
```

```
pyplot.figure(figsize=(3,7))
pyplot.imshow(H)
pyplot.show()
```

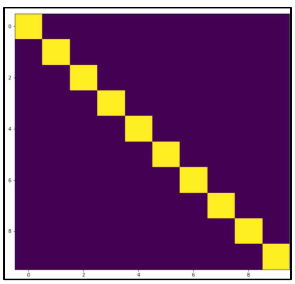


다음은 해당 논문에서 설명한 Algorithm 1로, 사용한 'Mask construction Pseudo Code'입니다.

Algorithm 1: Mask construction Pseudo Code

```
function g(H)
    l,n = H.shape
    k = n-l
    mask = eye(2n-k)
    for i in range(0,n-k) do
        idx = where(H[i]==1)
        for j in idx do
            mask[n+i,j] = mask[j,n+i] = 1
            for k in idx do
                mask[j,k] = mask[k,j] = 1
    return -∞(¬ mask)
```

4 : mask = eye(2n-k)



identity matrix로 '초기화'합니다. 따라서, 그림으로 표현했을 때, 다음처럼 1의 위치에 노란색으로 표현된 것을 볼 수 있습니다.

6 : idx=where(H[i]==1)

```
for i in range(0,3):#range(0,n-k):
  idx=np.where(H[i]==1)
  print('idx',idx)
```

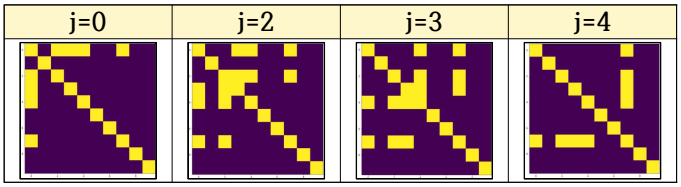
```
idx (array([0, 0, 0, 0]), array([0, 2, 3, 4]))
idx (array([0, 0, 0, 0]), array([1, 3, 4, 5]))
idx (array([0, 0, 0, 0]), array([2, 4, 5, 6]))
```

parity check matrix의 각각의 '1'의 위치를 파악할 수 있습니다. 1행의 경우 1의 위치가 1, 3, 4, 5번째에 존재하는 것을 알 수 있고, 2행의 경우 1의 위치가 2, 4, 5,

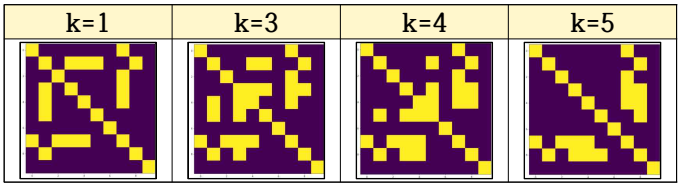
6번째에 존재하는 것을 알 수 있고, 3행의 경우, 1의 위치가 3, 5, 6, 7번째의 열에 존재하는 것을 알 수 있습니다.

전체 과정을 반복하면

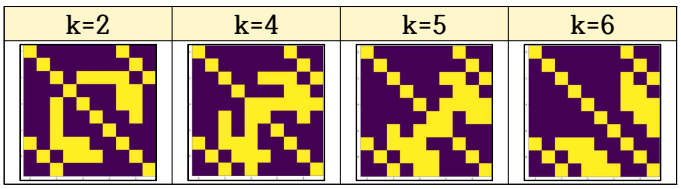
1) $i = 0, idx = 0,2,3,4$



2) $i = 1, idx : 1,3,4,5$



3) $i = 0, idx = 2,4,5,6$



해당 순서로 'Algorithm 1 : Mask construction Pseudo Code'를 통해, mask를 다음처럼 구했습니다.

```
for jj in idx:
    for kk in idx:
        if jj != kk:
            mask[jj, kk] += 1
            mask[kk, jj] += 1
            mask[code.n + ii, jj] += 1
            mask[jj, code.n + ii] += 1
```

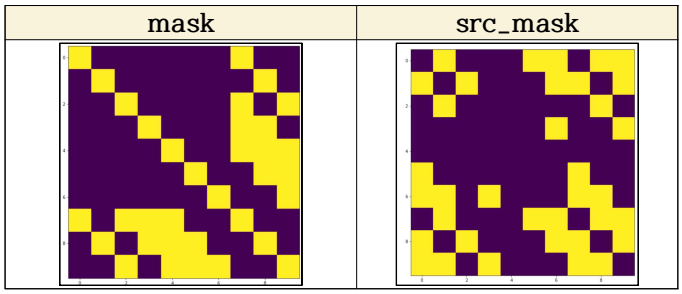
parity check matrix는 binary parity check matrix이고, 코드를 보면, Algorithm 1의 Pseudo Code에서는 'mask[n+i, j]=mask[j, n+i]=1'으로 표현되는데, 코드는 'mask[code.n+ii,jj]+=1'로 표현되고, binary 값이기에, '1+1=0'으로 표현됩니다.

```
mask[j,k]+=1
mask[k,j]+=1

mask=mask%2
print('mask값₩n',mask)

pyplot.figure(figsize=(10,10))
pyplot.imshow(mask)
pyplot.show()
```

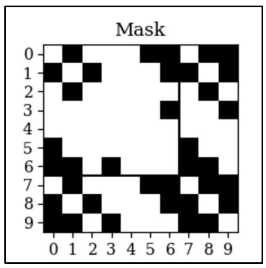
이를 확인하기 위해서, 'colab'에서는 'mask[j,k]+=1'와 'mask=mask%2'로 표현하였습니다. 마지막에 'src_mask = ~ (mask > 0).unsqueeze(0).unsqueeze(0)'를 통해, 논문에서 나타낸 mask가 제대로 표현한 것을 확인하였습니다.



최종 결과가 해당 시뮬레이션의 결과가 다르기에 잘못 표현한 것이라고 결론 내렸습니다.

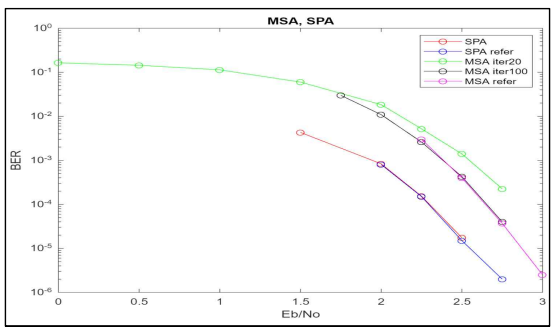
```
src_mask.shape
print(src_mask.long())

tensor([[[0, 1, 1, 1, 1, 1, 1, 0, 1, 1],
         [1, 0, 1, 1, 1, 1, 1, 1, 0, 1],
         [1, 1, 0, 1, 1, 1, 1, 0, 1, 0],
         [1, 1, 1, 0, 1, 1, 1, 0, 0, 1],
         [1, 1, 1, 1, 0, 1, 1, 0, 0, 0],
         [1, 1, 1, 1, 1, 0, 1, 1, 0, 0],
         [1, 1, 1, 1, 1, 1, 0, 1, 1, 0],
         [0, 1, 0, 0, 0, 1, 1, 0, 1, 1],
         [1, 0, 1, 0, 0, 0, 1, 1, 0, 1],
         [1, 1, 0, 1, 0, 0, 0, 1, 1, 0]]])
```



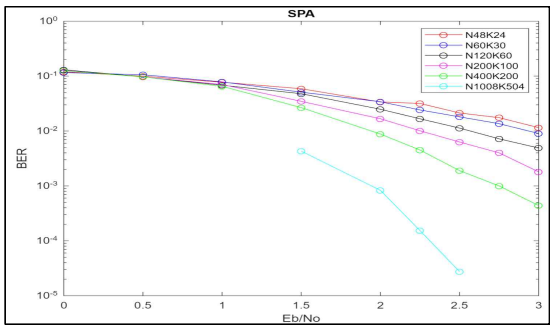
3.2) LDPC : Sum Product Algorithm vs Min-Sum Algorithm vs proposed algorithm

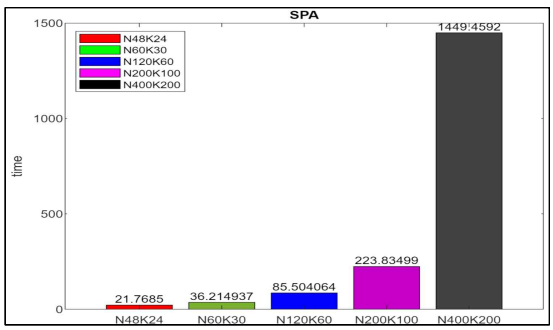
'LDPC'의 decoding algorithm인 Sum Product Algorithm (SPA)와 Min-Sum Algorithm (MSA)와 제안된 알고리즘의 성능을 비교했고, 논문은 LDPC의 성능에 대한 부분을 언급하지 않아서 적용했습니다. 우선, SPA는 Neural Network model로 표현할 때, odd ith layer는 기존의 Variable node update를 나타내며, $\mu_{v,c}^{t} = l_v + \sum_{c' \in N(v) \setminus c} \mu_{c',v}^{t-1}$, even ith layer는 기존의 Check node update를 나타내고, $\mu_{c,v}^{t} = 2\tanh^{-1}\left(\prod_{v' \in M(c) \setminus v} \tanh\left(\frac{\mu_{v',c}^{t}}{2}\right)\right)$, 마지막 network의 vth output은 $s_v^t = l_v + \sum_{c' \in N(v)} \mu_{c',v}^{t}$로 표현됩니다.



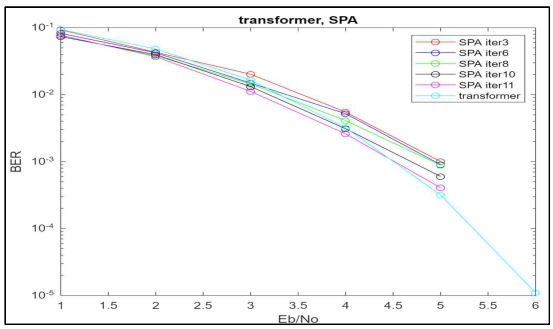
우선, 비교를 위해 사용한 SPA, MSA의 성능의 평가를 위해 [1]의 논문과 성능을 비교하였습니다. 이를 통해, 기존의 decoding 방법과의 제안한 방법의 성능 비교와 code word에 따라 성능과 소요 시간(연산시간, 복잡도)을 비교하겠습니다.

1) Sum-Product Algorithm (SPA)





우선 1번째 그래프는 'SPA'의 성능을 보여주고, 2번째 그래프는 '매트랩'의 'tic', 'toc'을 통해 작동 시간으로 복잡도를 확인하였습니다. 따라서, codeword의 **'길이'** 값인 N의 값이 증가함에 따라 성능이 개선되고 그만큼 복잡도도 증가하는 것을 확인할 수 있습니다.

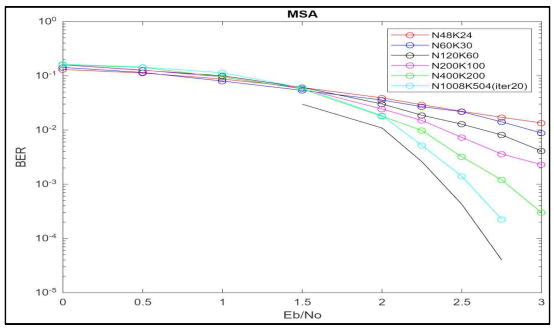


해당 그래프는 parity check matrix에서 iteration을 다르게 설정하였고, iteration이 증가함에 따라 계산량이 많아져서, complexity는 늘어나지만, 그에 따라 성능이 개선되는 trade off가 존재합니다. 따라서, 새로운 decoding 연구는 complexity를 줄이면서, 기존의 연구와 성능을 비슷하게 가져가거나, complexity를 일부 늘리더라도, 성능 개선이 많이 되는 것에 집중합니다.

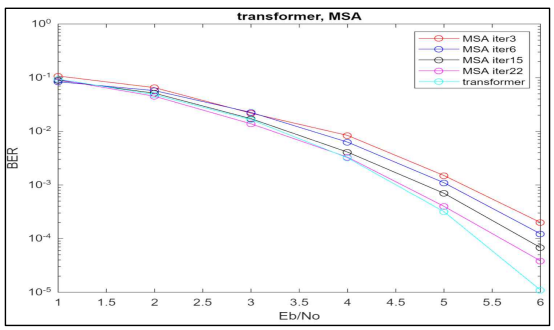
SPA와 비교했을 때 10회 이상의 iteration을 했을 때와 유사한 성능을 가졌다는 것을 확인했습니다. 'code'에서는 "parser.add_argument ('--N_dec', type=int, default=6)", 이때 N은 layer의 수를 나타내고. neural network 표현으로 변형하면, 1번의 iteration은 check node layer, variable node layer에서 각각 한 번씩 decoding이 진행되기에, @번의 iteration은 2@개의 layer가 존재합니다. 따라서, 'N=6'의 default 값은 iteration 3임을 의미한다는 것을 알 수 있고, 제안한 방법은 SPA에 비해 더 적은 iteration을 사용하였음에도 비슷한 성능을 가진다는 것을 확인할 수 있습니다.

2) Min-Sum Algorithm(MSA)

MSA의 경우, SPA의 CN update 식에 존재하는 tanh와 같은 식에 의해 계산의 복잡도가 존재하기에, 이를 개선하기 위해서 고안된 방법입니다. 즉, 다음과 같이 표현됩니다. $\mu_{c,v}^{t} = \min_{v' \in M(c) \setminus v}(|\mu_{v',c}^{t}|) \prod_{v' \in M(c) \setminus v} sign(\mu_{v',c}^{t})$

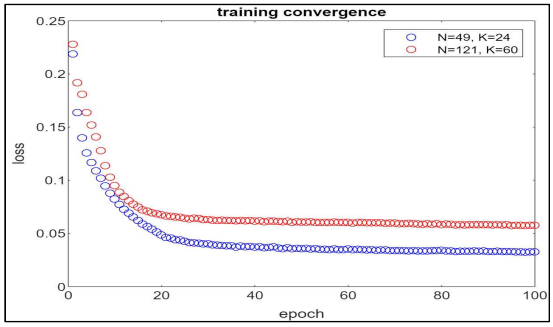


decoding algorithm의 경우, 성능과 복잡도의 tradeoff 관계가 있는데, 이를 개선하는 것이 decoding의 연구의 '목표'입니다. 이때, 'MSA'의 경우, SPA보다 복잡도는 개선되었지만, 성능에서는 약간의 loss가 발생합니다.



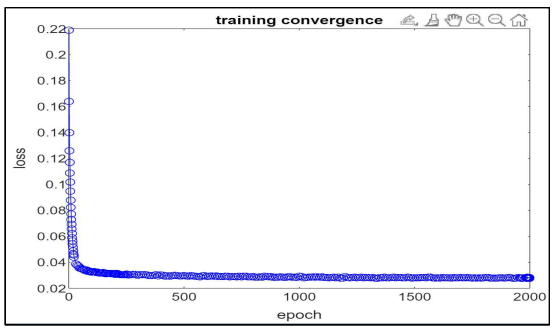
해당 결과는 저자가 제안한 방법과 기존의 MSA의 성능 비교를 한 것으로, SPA보다 더 많은 iteration의 MSA 성능이 비슷한 것을 확인할 수 있었습니다. 즉, 성능보단 복잡도 개선에 집중했던 'MSA'의 iteration 20 이상의 성능과 유사한 성능을 적은 iteration으로 출력한 것을 확인했습니다.

3.3) training convergence

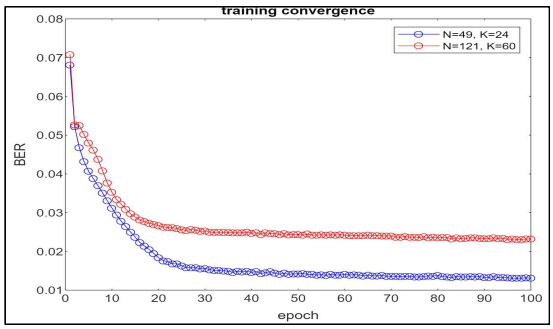


blue : self-attention sparsity ratio=72.26%, self-attention complexity ratio=13.87%, N=49, K=24

red : self-attention sparsity ratio = 74.55%, self-attention complexity ratio =12.27%, N=121, K=60



blue : self-attention sparsity ratio=72.26%, self-attention complexity ratio=13.87%, N=49, K=24



4. 결론

5. 한계점

해당 프로젝트를 진행하면서 아쉬웠던 점이 있습니다. 그 부분에 대해서 언급하겠습니다.

```
if __name__ == '__main__':
    parser = argparse.ArgumentParser(description='PyTorch ECCT')
    parser.add_argument('--epochs', type=int, default=1000)
    parser.add_argument('--workers', type=int, default=4)
    parser.add_argument('--lr', type=float, default=1e-4)
    parser.add_argument('--gpus', type=str, default='-1', help='gpus ids')
    parser.add_argument('--batch_size', type=int, default=128)
    parser.add_argument('--test_batch_size', type=int, default=2048)
    parser.add_argument('--seed', type=int, default=42)
```

해당 부분은 'Main.py'의 코드 중 일부로, argument parser가 있습니다. 즉, 초기 파라미터 입력받는 부분이 있는데, 표시된 'worker'라는 부분이 있습니다. 그리고, 데이터 로더의 프로세서를 몇 개로 할지를 결정하는 부분으로 만일 CPU가 멀티 코어가 안 되면 설정하면 안 되는 옵션입니다. github 코드는 default 값을 4로 설정하였지만, 프로젝트를 진행하기 위해 사용한 컴퓨터에서는 안 되었습니다. 즉, 해당 부분을 통해 데이터 로딩을 멀티프로세스로 하느냐, 싱글 프로세스로 하느냐를 결정하는 것으로 '0'으로 하면 시간이 오래 걸리고, 컴퓨터 세팅의 사양이 좋으면 @값으로 설정하면 되는데, 만일 '8'정도로 설정하면 약 4배 정도로 빠르게 돌아가게 됩니다. 그 이유는 '1' 당 프로세서 한 개를 추가하게 되어, 대략 0.7배 정도 빨라지기 때문입니다. 특히, 선행 연구하는 구글은 알파고 실험할 때 130 이상의 값으로 설정하여 연구를 진행하였고, 해당 사항은 하드웨어에 무리가 됩니다. 그리고 '0'과 '1'의 차이는 '소프트웨어 아키텍처'에서의 차이가 나는데, 실제 성능에서는 큰 차이는 없지만 '1'을 올리면 독립적인 싱글 프로세서가 한 개 생기는데, 일정 컴퓨터 성능 사양 아래에서는 이런 작업이 수행 불가능하기에 default 값을 '0'으로 설정하고 진행하였고, 해당 값은 '0' 이상의 '정수'값만 가능합니다.

Reference

[1] Jinghu Chen, A. Dholakia, E. Eleftheriou, M. P. C. Fossorier and Xiao-Yu Hu, "Reduced-complexity decoding of LDPC codes,"*IEEE Transactions on Communications*, vol. 53, no. 8, pp. 1288-1299, Aug. 2005.

사용 코드

1. paper code : https://github.com/yoniLc/ECCT
2. Creating a Parity Check Matrix, alist file : https://radfordneal.github.io/LDPC-codes/pchk.html