reproducibility_challenge

- Error Correction Code Transformer

1. introduction

오류정정부호는 노이즈 있는 채널을 통해 data를 전송할 때 신뢰성을 확보한 것으로, 통신 물리 계층에서 주된 부분 중 하나로, 최근 deep learning이 접목된 연구는 기존의 decoding 기술보다 성능이 좋았지만, training 복잡도와 같은 이슈는 아직 해결되지 않았고, 이를 해결하는 방법을 고안했습니다. 해당 연구는 기존의 방법과 달리 기계번역과 같은 자연어 처리에서 주로 사용되던 transformer architecture 기반의 model free decoder를 임의의 블록 길이를 가진 선형 코드(LDPC, BCH 등)의 soft decoding에 처음으로 제안하였고, adapted masked self-attention module을 통해 algebraic code와 bits 간의 상호 작용을 모델에 적용한 것이 연구의 방향성입니다.

2. background

2.1) transformer

저자가 활용한 transformer는 2017년에 Vaswani 등이 발표한 논문으로 기존의 'seq2seq'의 구조인 encoder - decoder 구조와 Attention을 결합하였고, RNN을 사용하지 않았음에도 우수한 성능을 가졌습니다. 이때, transformer는 기존의 모델처럼 encoder에서 input sequence를 받고, decoder에서 output sequence를 출력하고, 'RNN'은 't'개의 time step을 갖지만, N개의 encoder, decoder로 구성되어 있고, 모든 token을 동시에 입력받아 병렬 연산하여 학습 속도와 성능을 '향상'시켰습니다.

2.2) Attention

Attention에는 'Encoder Self-Attention', 'Masked Decoder Self-Attention', 'Encoder-Decoder Attention'이 있는데, 저자가 활용한 'Self-Attention'은 다음의 과정을 거칩니다.

1) input vector로부터 Query, key, Value 벡터를 만듭니다.

2) 특정 위치의 단어가 다른 단어와 얼마나 연관되어 있는지, Query와 Key 벡터의 내적을 통해 score를 측정합니다.

3) Score를 Key 벡터의 차원 수의 제곱근으로 나눠서, gradient를 조절하고, Softmax를 계산합니다.

scores=torch.matmul(query, key. transpose (-2,-1)) / math.sqrt(d_k)

4) Value 벡터와 앞서 구한 Softmax score를 곱합니다.

return torch.matmul(p_attn, value), p_attn

```
def attention(self, query, key, value, mask=None):
    d_k = query.size(-1)
    scores = torch.matmul(query, key.transpose(-2, -1)) ₩
        / math.sqrt(d_k)
    if mask is not None:
        scores = scores.masked_fill(mask, -1e9)
    p_attn = F.softmax(scores, dim=-1)
    if self.dropout is not None:
        p_attn = self.dropout(p_attn)
    return torch.matmul(p_attn, value), p_attn
```

self-attention code

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 '앞' 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.

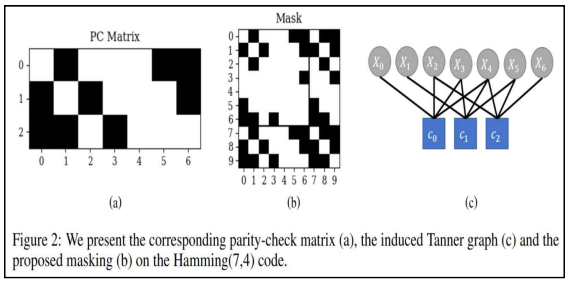
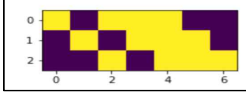


Figure 2: We present the corresponding parity-check matrix (a), the induced Tanner graph (c) and the proposed masking (b) on the Hamming(7,4) code.

위 논문의 그림에서는 '흰'색 부분이 parity check matrix의 '1'의 위치에 해당하는 부분을 의미합니다. 예를 들어, (c)의 X_0와 c_0가 연결된 것은 parity check matrix(PCM)의 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 \n',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()
```

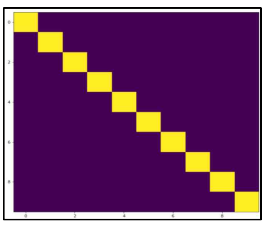


다음은 해당 논문에서 설명한 Algorithm 1로, 사용한 'Mask construction Pseudo Code'입니다.

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)



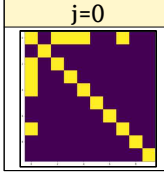
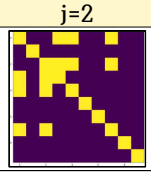
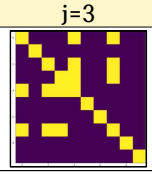
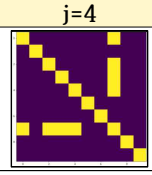
identity matrix로 '초기화'합니다. 따라서, 그림으로 표현했을 때, 1의 위치에 노란색으로 표현된 것을 볼 수 있습니다.

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

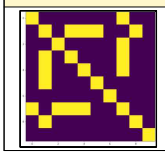
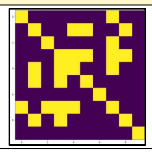
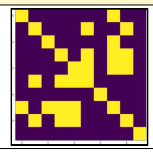
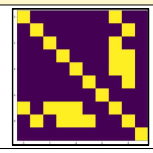
```
for i in range(0,3):#range(0,n-k):
  idx=np.where(H[i]==1)
  print('idx',idx)

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]))
```

PCM의 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.

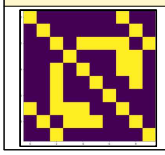
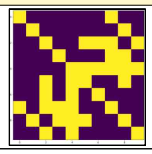
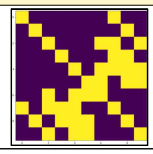
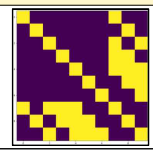
전체 과정을 반복하면 1) $i=0, idx=0,2,3,4$

j=0	j=2	j=3	j=4
			

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

k=1	k=3	k=4	k=5
			

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

k=2	k=4	k=5	k=6
			

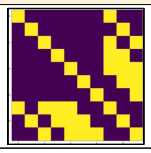
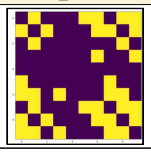
해당 순서로 'Algorithm 1 : Mask construction Pseudo Code'를 통해, mask를 다음처럼 구했습니다.

```
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'**로 표현되는데, 코드는 **'mask[code.n+ii,jj]+=1'**로 표현되고, binary 값이기에, '1+1=0'으로 표현됩니다.

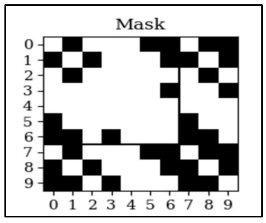
```
mask[j,k]+=1
mask[k,j]+=1

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, 논문의 mask가 맞게 표현한 것을 확인하였습니다.

mask	src_mask
	

다만 'Pseudo Code'에서 'mask[n+i,j]=mask[j,n+i]=1' 표현을 'mask[n+i,j]=mask[j,n+i]+=1'로 표현하거나, binary 계산이니까 switch처럼 on, off로 표현하고, 'mask'를 살펴본 이유는 'unsqueeze' 전 내용까지만 고려하여 논문의 내용과 mask 모양이 달라 오류라고 생각했기 때문에, 이점을 고려하여 'Pseudo Code'를 구성하였으면 좋지 않았을까? 하고 생각했습니다.

```
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, 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

3.2.1) Sum-Product Algorithm (SPA)

'LDPC'의 decoding algorithm인 Sum Product Algorithm (SPA)와 Min-Sum Algorithm (MSA)와 제안된 알고리즘의 성능을 비교했고, 논문은 'LDPC'의 성능에 대한 부분을 언급하지 않아서 적용했습니다. SPA는 Neural Network model로 표현할 때,

odd ith layer : Variable node update :

$$\mu_{v,c}^{t} = l_v + \sum_{c' \in N(v) \setminus c} \mu_{c',v}^{t-1} \quad (1)$$

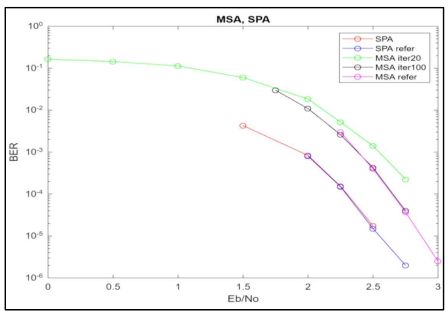
even ith layer : Check node update :

$$\mu_{c,v}^{t} = 2\tanh^{-1}\left(\prod_{v' \in M(c) \setminus v} \tanh\left(\frac{\mu_{v',c}^{t}}{2}\right)\right) \quad (2)$$

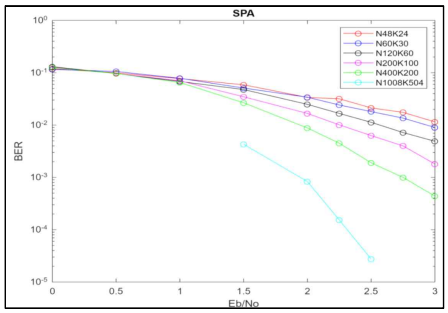
final network의 vth soft output :

$$s_v^t = l_v + \sum_{c' \in N(v)} \mu_{c',v}^{t} \quad (3)$$

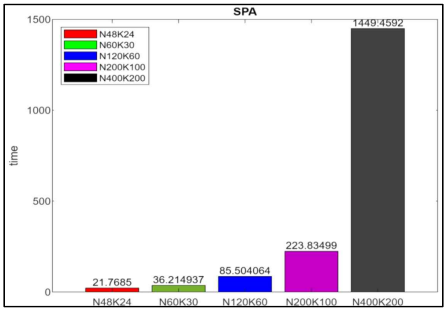
로 표현됩니다.



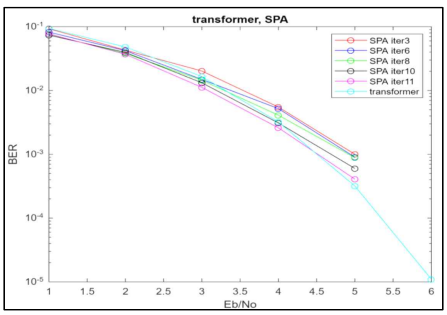
우선, 비교를 위해 사용한 SPA, MSA의 성능의 평가를 위해 [2]의 논문과 성능을 비교하였습니다. 이를 통해, 기존의 decoding 방법과의 제안한 방법의 성능 비교와 code word에 따라 성능과 소요 시간(연산시간, 복잡도)을 비교하겠습니다.



1번째 그래프는 'SPA'의 performance를 보여주고, 2번째는 '매트랩'의 'tic', 'toc'을 통해 작동 시간으로 복잡도를 확인하였습니다.



따라서, codeword **'길이'** 값인 N의 값이 증가함에 따라 성능이 개선되고 그만큼 복잡도도 증가하는 것을 확인했습니다.

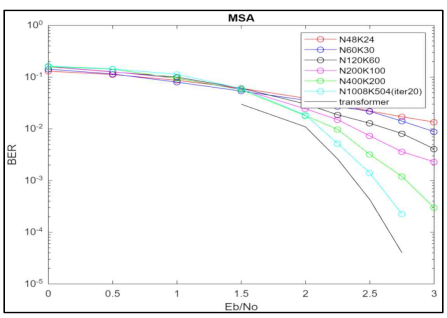


해당 그래프는 parity check matrix에서 iteration을 다르게 설정하였고, iteration이 증가함에 따라 계산량이 많아져서, complexity는 늘어나지만, 그에 따라 성능이 개선되는 trade off가 존재하고, 새로운 decoding 연구는 complexity를 줄이면서, 기존의 연구와 성능을 비슷하게 가져가거나, complexity를 일부 늘리더라도, 성능 개선이 많이 되는 것에 집중합니다. SPA와 비교했을 때 10회 이상의 iteration과 유사한 성능을 가졌다는 것을 확인했습니다. 'code'에선 "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 한 번씩 decoding이 진행되기에, @번의 iteration은 2@개의 layer가 존재합니다. 따라서, **'N=6'의 default 값은 iteration 3** 임을 의미한다는 것을 알 수 있고, 제안한 것은 SPA에 비해 더 적은 iteration을 사용하였음에도 비슷한 성능을 가진다는 것을 확인할 수 있습니다.

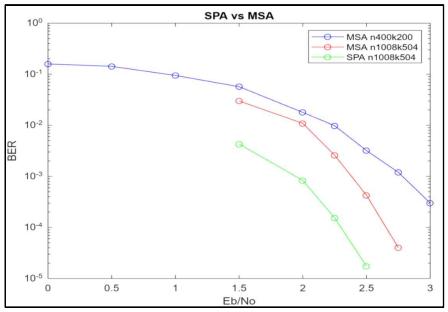
3.2.2) Min-Sum Algorithm(MSA)

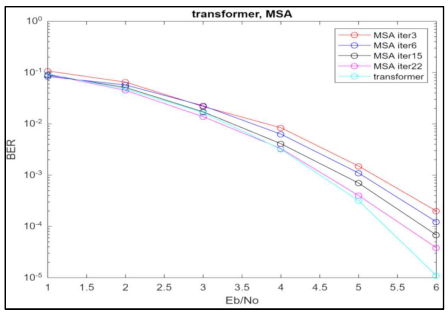
MSA의 경우, SPA의 (2) CN update 식에 존재하는 tanh와 같은 식에 의해 계산의 복잡도가 존재하기에, 이를 개선하기 위해서 고안된 방법으로, 다음처럼 표현됩니다.

$$\mu_{c,v}^{t} = \min_{v' \in M(c) \setminus v}(|\mu_{v',c}^{t}|) \prod_{v' \in M(c) \setminus v} sign(\mu_{v',c}^{t}) \quad (4)$$



decoding algorithm의 경우, 언급한 tradeoff 관계를 고려해야 하며, 'MSA'의 경우, SPA보다 복잡도는 개선되었지만, 성능에서는 loss가 발생합니다.



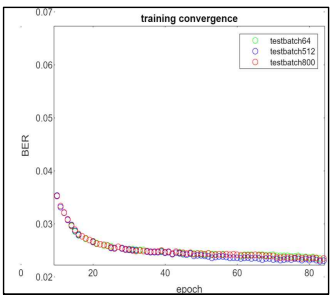


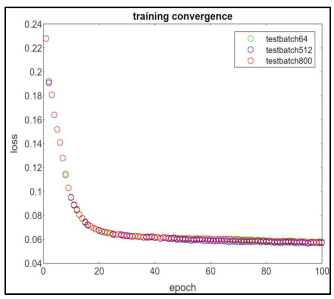
해당 결과는 저자가 제안한 방법과 기존의 'MSA'의 성능 비교를 한 것으로, SPA보다 더 많은 iteration의 'MSA' 성능이 비슷한 것을 확인할 수 있었습니다. 즉, 성능보단 복잡도 개선에 집중했던 'MSA'의 iteration 20 이상의 성능과 유사한 성능을 적은 iteration으로 출력한 것을 확인했습니다.

3.3) training convergence

3.3.1) iteration, test_batch_size

해당 결과는 iteration에 따라 다른 결과와 iteration 3일 때는 batch_size에 따라 달라지는 결과를 봤습니다. batch_size는 '--test_batch_size = 800'와 '--test_batch_size = 512', '--test_batch_size = 64'로 설정하여 출력했습니다.

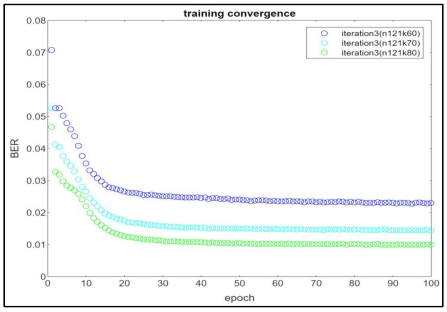




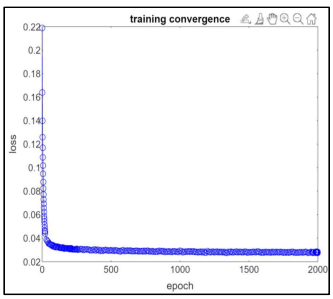
해당 결과를 통해, 'test_batch_size'에 따른 결과(loss, BER), epoch 당 소요 시간이 거의 차이 없다는 것을 확인했습니다.

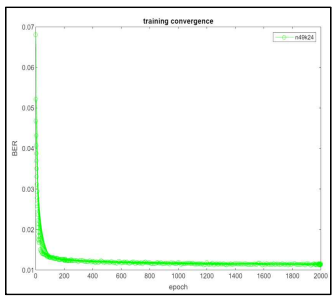
3.3.2) Self-Attention Sparsity/Complexity ratio

	Self-Attention Sparsity Ratio	Self-Attention Complexity Ratio	epoch 당 소요 시간
n121k60	74.55%	12.72%	103.98s
n121k70	75.99%	12%	89.77s
n121k80	78.06%	10.97%	86.83s



해당 결과는 같은 iteration(3)일 때 k 값을 바꾼 것으로, 그 값에 따라 'Self-Attention Sparsity Ratio = 100 * torch . sum ((src_mask).int()) / a:0.2f', 'Self-Attention Complexity Ratio = 100 * torch.sum((~src_mask).int())//2 / a:0.2f', 'epoch 당 소요 시간'을 파라미터로 설정하였습니다. 결과를 통해, k 값이 증가함에 따라 PCM의 '1'의 개수가 증가하기에 'Sparsity Ratio'가 증가하고, 'Complexity ratio'가 감소하기에 epoch 당 소요 시간이 감소하는 것을 확인했습니다.





해당 결과는 'self-attention sparsity ratio = 72.26%, self-attention complexity ratio=13.87%, N=49, K=24'에 대한 'loss'와 'BER' 값을 출력한 것입니다.

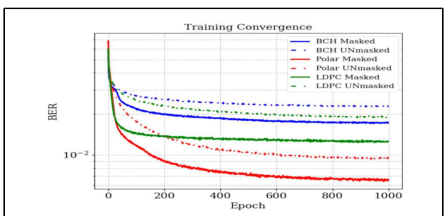


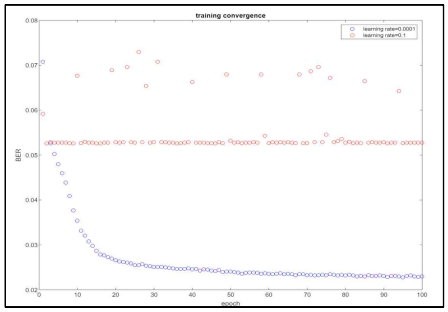
Figure 6: Ablation study on the proposed masking procedure on the training convergence for the BCH(63, 36), POLAR(64, 32) and LDPC(49, 24) using $(N, d) = (6, 64)$ architectures.

논문의 '6.3 Complexity Analysis'의 Figure 6의 그래프를 통해 결과를 보여주고 있는데, 'LDPC(49,24) using (N,d) = (6,64)'로 진행하였고, 해당 조건은 'default' 값이고, 그래프를 처음 봤을 때 헷갈릴 수 있지만, 10^{-2}의 위는 '0.02'이기에 'LDPC Masked'의 결과와 비슷했습니다.

3.3.2) learning rate

learning rate(LR)는 학습률, Gradient descent는 기울기의 크기가 줄어드는 쪽으로 비용 함숫값을 최소가 되는 점을 찾아가는 알고리즘이고, cost의 미분한 값에 learning rate를 곱하여, 어느 정도의 크기로 기울기가 줄어드는 지점으로 이동할지를 나타내는 지표입니다. 이때 LR값이 크면 over shooting이 발생하여, 최솟값에 도달하기 전에 그래프를 벗어날 수 있고,

반대로 작은 LR에 의해, train 속도, 즉 최솟값에 도달할 때까지 걸리는 시간이 오랜 시간이 필요하기에, 'learning rate' 값을 적절하게 설정해야 합니다.



learning rate : 0.0001 vs 0.1

결과를 보면, learning rate(0.1)가 매우 작을 때 성능이 매우 좋지 않은 것을 확인할 수 있습니다.

4. 결론

해당 연구는 기존 연구인 weight를 서로 다른 edge에 할당하거나, correction factor 값을 training을 통해 최적화를 하는 것과 달리 'transformer'라는 architecture를 decoding에 처음으로 접목한 것으로, 다양한 코드에 적용하여 성능의 우수성을 보여주었고, 시뮬레이션 결과를 통해 기존의 decoding 방법인 'SPA', 'MSA'와 비교했을 때, 더 적은 iteration으로 비슷한 성능을 보여주었기에, 획기적인 방법이라는 것을 보여줬습니다. 그리고 다양한 파리미터의 값을 바꿔가면서 결과에 어떤 영향을 주는지 확인했습니다.

5. 한계점

5.1) Simulation

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)
```

Main.py

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 멀티코어가 안 되면 설정하면 안 되는 옵션입니다. github 코드는 default 값을 4로 설정하였지만, 프로젝트 진행을 위해 사용한 컴퓨터에선 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.

5.2) batch_size

연구는 batch_size에 대한 언급이 없었습니다. 시뮬레이션 중 'test_batch_size'와 'batch_size'의 값에 따라 **'RuntimeError: CUDA out of memory. Tried to allocate 968.00 MiB'**와 같은 error가 발생하였고, 'batch_size'에 따라 epoch 1번당 소요시간과 성능에 대해서는 언급하지 않았고, default 값으로 시뮬레이션할 때 일부 코드에서는 앞서 언급한 'Runtime Error'가 발생하였고, 시뮬레이션 결과를 언급할 때 batch_size를 다르게 할 때는 별도의 표시를 하였습니다.

Reference

[1] Vaswani, Ashish, et al. "Attention is all you need." Advances in neural information processing systems 30 (2017).

[2] 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