2.1. Reformer : The Efficient Transformer

논문 : https://arxiv.org/abs/2001.04451

참고자료

https://speakerdeck.com/scatterlab/reformer-the-efficient-transformer

https://brunch.co.kr/@synabreu/31

https://iclr.cc/virtual_2020/poster_rkgNKkHtvB.html

Transformer의 문제점 - **더 긴 문장에 대해서 처리할 때의 메모리 문제** (2. Transformer 논문 리뷰 바로가기)

Transformer, BERT 기법을 기반으로한 Task는 최대 512개의 토큰을 입력으로 사용

=> 문서 단위의 언어 데이터/문서 데이터가 다뤄지면 모델을 사용하는데 필요한 메모리가 너무 커짐

Attention에서의 메모리 문제

- Query (Q): 영향을 받는 단어 *A*를 나타내는 변수입니다.
- Key (K) : 영향을 주는 단어 *B*를 나타내는 변수입니다.
- Value (V) : Key에 대응되는 영향력 값입니다.

이 경우, 가중치 합은 다음과 같이 나타납니다.

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

그림 : Scaled dot-Product Attention 수식

- Attention에서 사용하는 Query와 Key의 크기는 (배치, 토큰 길이, 모델 히든사이즈)
- QK^T 결과로 (배치, 토큰 길이, 토큰 길이) 형태의 Attention Score 가 나옴
  => 토큰 길이가 커질수록 Attention Score의 크기는 제곱에 비례함
  ex) 토큰 길이가 10배 길어지면 Attention Score의 크기는 100배 큰 구조가 됨
- 문서 단위의 내용을 한꺼번에 넣을려면 굉장히 많은 GPU/TPU가 필요하게 됨

Feed Forward Layer의 메모리 문제

$$\text{FFN}(x) = max(0, x \cdot W_1 + b_1) \cdot W_2 + b_2$$

그림 : Transformer/BERT에서 사용하는 Feed Forward Layer 구조 수식

- Transformer에서는 Feed Forward Layer의 크기를 2048로 지정함 (Transformer의 Encoder/Decoder의 크기는 512)
- Feed Forward Layer의 입력 x 의 형태 : (배치, 토큰 길이, Transformer 모델 크기 512)
  weight 저장 형태 : (Transforemr 모델 크기 512, Feed Forward 모델 크기 2048)
  출력 형태 : (배치, 토큰 길이, Feed Forward 모델 크기 2048)
  => 토큰 길이가 커질수록 Feed Forward Layer 크기가 곱에 비례함

N-stacked Residual Connection의 메모리 문제

- 여러 머신러닝 프레임워크 (Torch, Tensorflow)에서 미분값을 계산하려면? (학습하려면?)
  - 모델 전체에 입/출력을 메모리에 기록 (Forward 과정)
  - 메모리에 기록된 입/출력값을 종점부터 시작점까지 거슬러 올라가며 미분값에 근사한 수치를 계산함 (Backward 과정)
- 학습과정에서 구조가 복잡한 Transformer/BERT의 입력과 출력을 저장하기 위한 메모리가 많이 사용됨
  => GPU/TPU 메모리에 모델도 로드해야되는데 학습할때 필요한 추가 메모리도 많이 필요함
  => Transformer 모델은 동일한 구조의 Encoder/Decoder가 쌓여있는 구조로 N층 쌓여있으면 N배의 메모리가 필요함

이 논문이 하려는 것은?

- 서로 비슷한 쌍만 Attend 하기
    - Attention에서 Key, Query를 Query값으로 통일해버리기
    - LSH(Locality-Sensitive Hashing)을 통해 비슷한 토큰을 빠르게 찾기
- Chunking을 이용한 메모리 줄이기
    - 전체 토큰간의 Attention을 하지 않고 비슷한 토큰끼리 묶어서(정렬해서) Attention을 진행하도록 하기
- Reversible Layer를 통해 N층의 입/출력을 메모리에 저장하지않고 말단 층의 입/출력만 메모리에 저장하기
    - Residual Connection을 변형한 Reversible Residual Layer(Reversible Residual Network 논문 바로가기)을 사용해서 출력값으로 입력을 복원하는 기법 적용

모르는 기법 알고가기

LSH(Locality-Sensitive Hashing)이 뭔데?

Locality-Sensitive Hashing : 가까운 값들끼리 가까운/비슷한 Hash값을 갖도록 Hashing하는 방법

Reformer에서 사용한 LSH기법 : Angular LSH

Angular LSH : 방향 성분만을 활용하여 Hash 값을 생성하는 방법

- 전체 데이터 포인트들의 벡터를 단위 구면에 사상. (반지름 1인 구면좌표계 형태로 표현, 아래 그림들의 맨 왼쪽)
- 벡터를 필요한 만큼 임의로 회전 (몇번 회전할지, 몇사분면으로 표현할지는 표현력 차이)
- 비슷한 Hash값을 가지면 가까운 백터로 판단

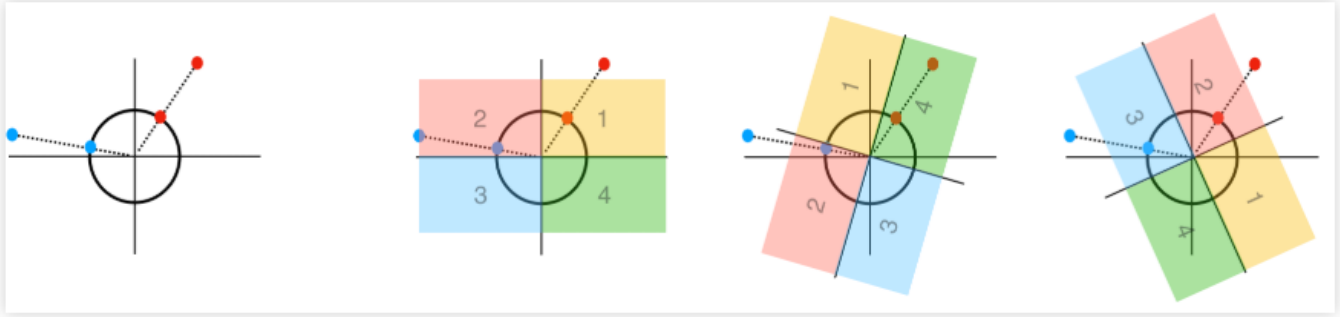

그림 : 벡터에 Angular LSH를 적용한 모습. 빨간색 백터는 (1, 4, 2) 사분면에 위치. 파란색 백터는 (2, 2, 3) 사분면에 위치해서 다른 hash 값을 가지는 것을 보여줌

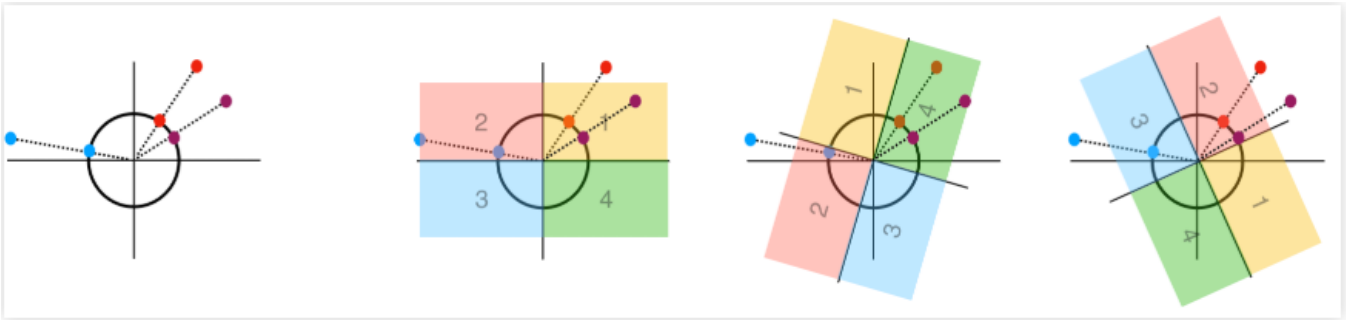

그림 : 벡터에 Angular LSH를 적용한 모습. 빨간색 백터는 (1, 4, 2) 사분면에 위치. 주황색 백터는 (1, 4, 2) 사분면에 위치. 파란색 백터는 (2, 2, 3) 사분면에 위치해서 주황색과 빨간색 백터는 같은 hash 값을 가지는 것을 보여줌

Reversible Residual Network가 뭔데?

- 기존의 Residual Network/Residual Connection은 아래 수식과 같음

$$y = x + F(x)$$

    - ResNet 구조에서 메모리를 효율적으로 사용하기 위해 고안됨.
    - 망이 깊어지면 발생하는 Gradient vanishing/exploding 또는 degradation이 발생하는데 Residual connection/skip connection을 통해 방지함
- Reversible Residual Network : y에서 x를 역으로 계산할 수 있도록 고안한 방법

$$y_1 = x_1 + F(x_2), y_2 = x_2 + G(y_1)$$

    - 기존의 입/출력 x, y를 (x1, x2), (y1, y2)쌍 형태로 분리

- 이 경우 y1, y2가 주어졌을 때 $x_2 = y_2 - G(y_1)$, $x_1 = y_1 - F(x_2)$ 로 역산할 수 있음
  => 임의의 시점의 출력값을 토대로 그 출력에 대한 입력값을 표현할 수 있음!
  => 중간 결과를 저장할 필요없이 Forward 연산을 반복적으로 적용해 수치적 미분값을 얻을 수 있게됨!
- (Reversible Residual Network 논문 바로가기) 에서 실험 결과를 살펴보면 미세한 성능하락이 있지만 사용 메모리는 상당히 감소한 것을 볼 수 있음

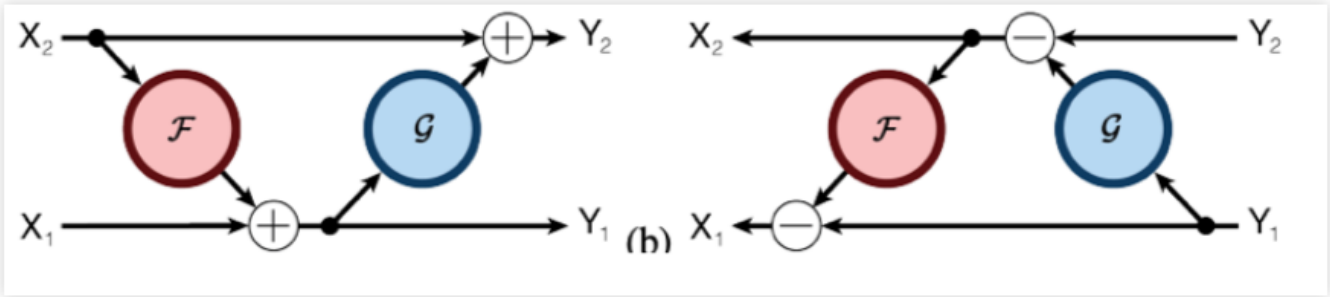

그림 : Reversible Residual network의 구조

Transformer 모델에 적용

논문이 추구하는 바는 모델을 사용할 때 필요한 메모리 감소!

Query, Key 동일하게 사용하기

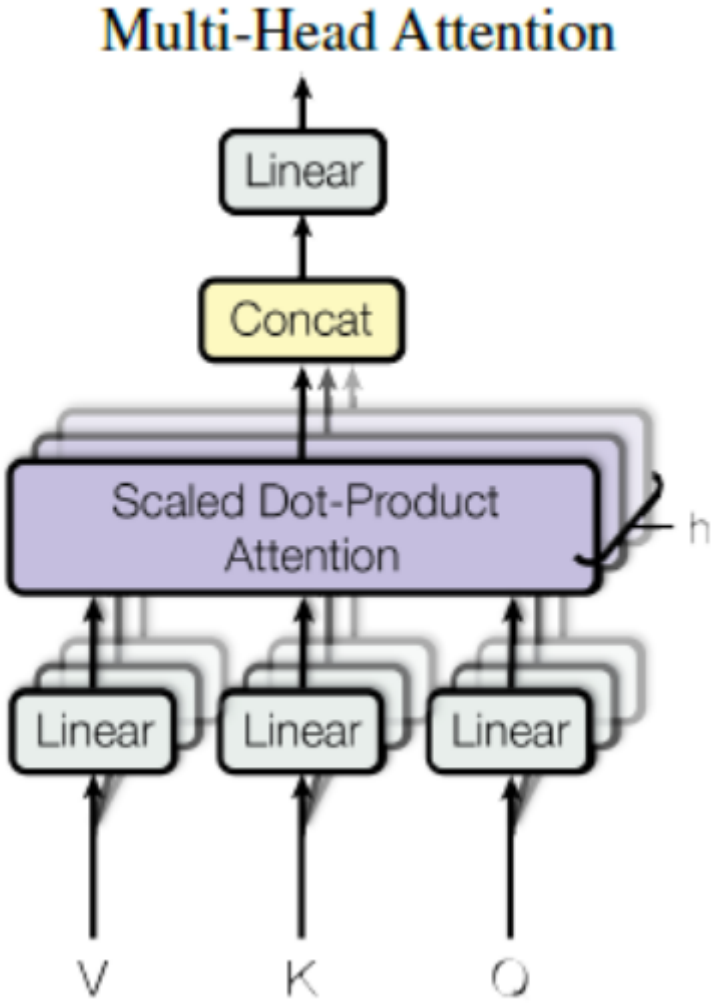

그림 : Multi-Head Attention 개략도

- 기존의 Attention은 토큰별 백터를 Query, Key로 만들기 위해 별도의 Linear Layer를 통과시킴
- 본 논문에서는 Query와 Key를 동일하게 사용해도 된다고 가정함
  따라서 Value를 계산하기 위한 Linear Layer를 버리고 Query값을 Key 값으로 사용함

$$Query(Q) = Key(K)$$

- 가설을 검증하기 위해 enwik8, imagenet64 데이터셋으로 실험했을 때 성능이 비슷하거나 더 빨리 수렴하는 것을 확인
  => 데이터가 풍부하다면 괜찮다~

  - imagenet64 : 이미지넷 데이터셋은 이미지에 등장한 객체의 종류를 분류하는 작업과 관련있으며 20,000개 이상의 종류 아래 14,000,000 개 이상의 이미지로 구성.
  - enwiki8-64K : enwiki8은 전체 영어 위키피디아 데이터를 압축하는 작업으로 본 논문에서 사용하는 enwiki8-64K는 각 부분이 64K 토큰으로 구성된 데이터.

가설 검증 1: Q=K

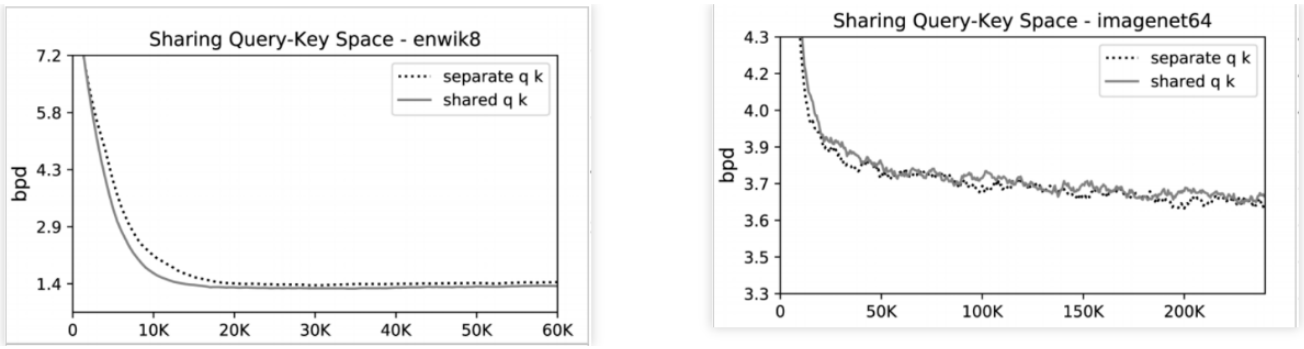

그림 : Query와 Key를 분리해서 계산할때와 Query를 Key로도 사용했을 때의 실험 결과

LSH Attention 사용하기

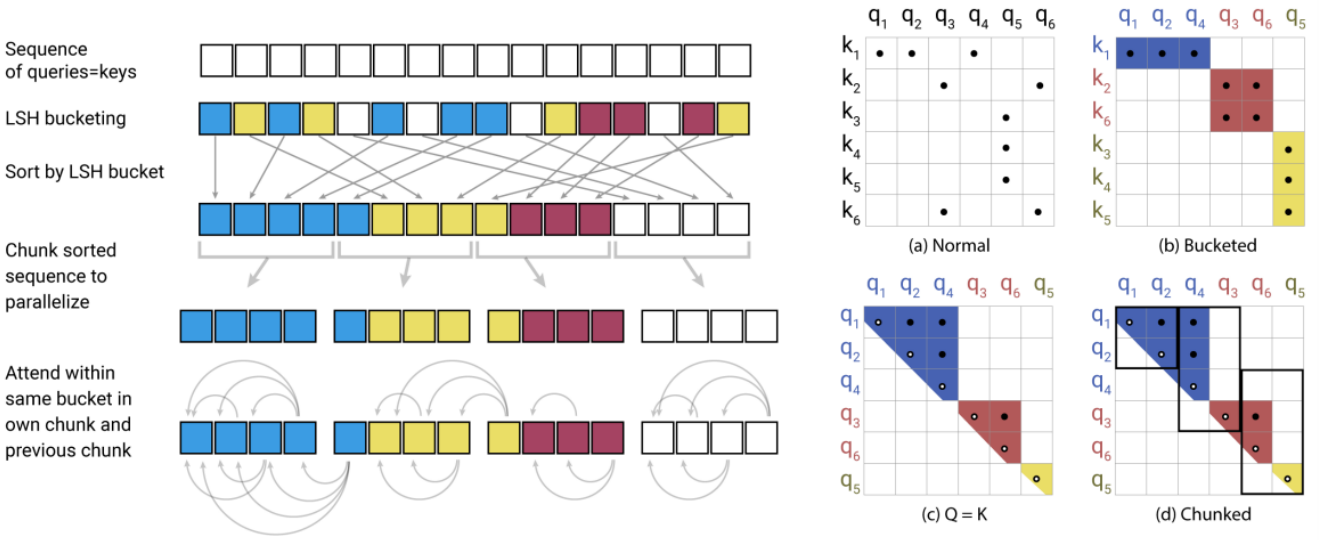

그림 : 왼쪽) LSH Attention에서 사용하는 hash-bucketing, sorting, chunking에 대한 개략도. 오른쪽 a-d) 그 과정에서 Attention Score의 변화

- 그림 설명
  1. Query와 Key는 동일하므로 백터 형태로 표현
  2. 각 토큰별 백터에 LSH를 적용. 같은 Hash를 가진 데이터 끼리 묶기
     (LSH bucketing)
  3. 각 Hash 버킷에 임의로 순서를 매겨서 정렬하기
     (Sort by LSH bucket)

4. 전체 토큰을 고정된 배치 사이즈로 분리하기
   (Chunk sorted sequence to parallelize)
5. Attention Weight를 계산할 때 같은 버킷(같은 색)에 있고, 앞의 토큰만 Attend할 수 있도록 처리
   이때, Query와 Key가 같아서 스스로 Attention을 하면 다른 토큰과의 Attention보다 무조건 클것이기 때문에 토큰 자신과의 Attention은 제외됨(그림 c의 흰 원)
   (Attend within same bucket in own chunk and previous chunk)

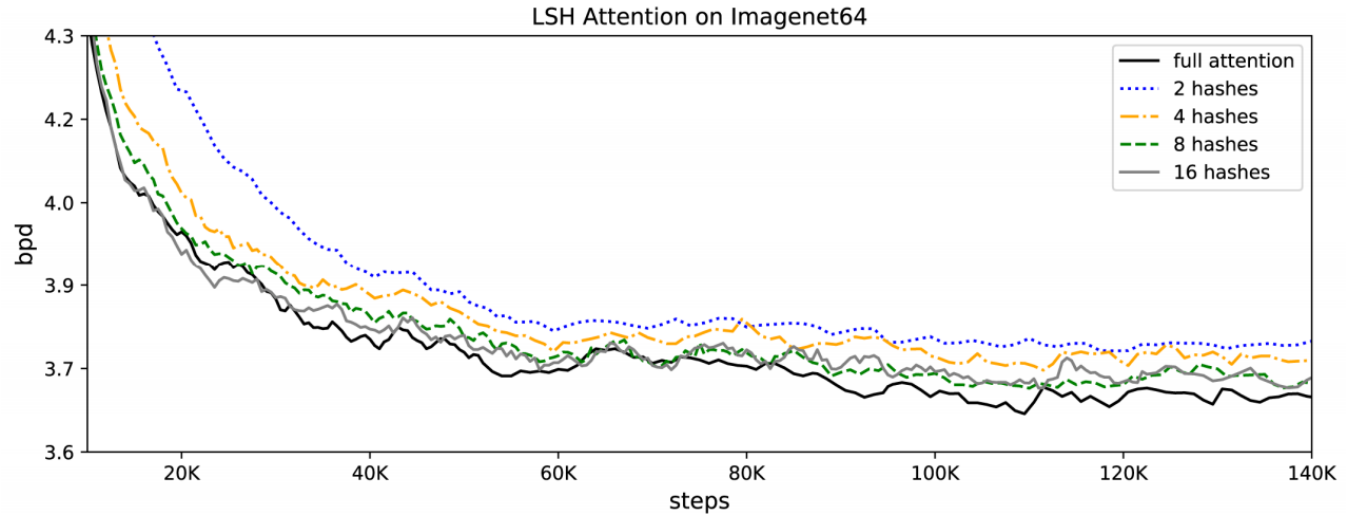

그림 : Imagenet64 데이터셋에 Hash 크기를 조절하면서 LSH Attention을 적용했을 때의 성능

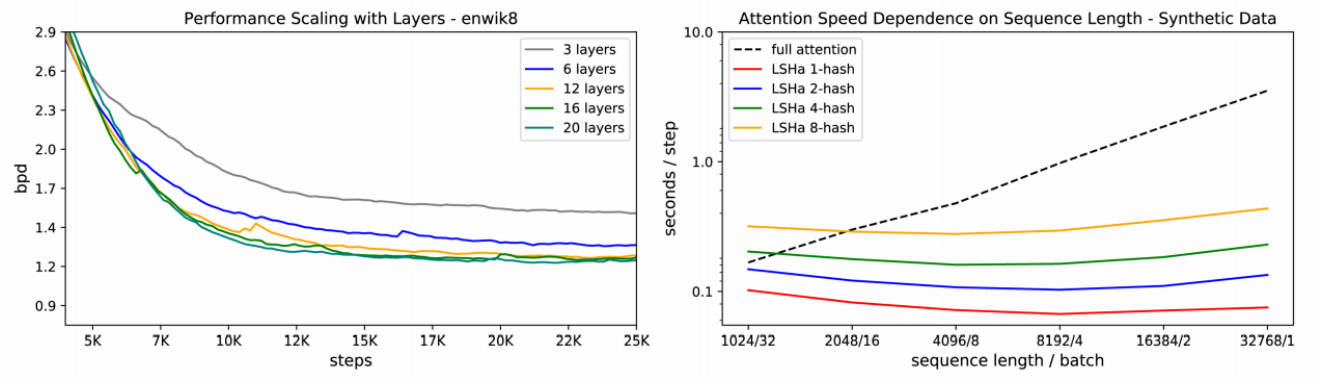

그림 : 왼)enwiki8 데이터셋에 LSH Attention을 적용한 레이어 수 차이에 따른 성능. 오) 입력 토큰 개수/배치사이즈 별 Attention Speed 차이

Reversible Residual Network 적용하기

$$y_1 = x_1 + F(x_2), y_2 = x_2 + G(y_1)$$

을 Transformer에 적용하기 위해서 아래와 같이 Attention과 FeedForward Layer에 적용

$$Y_1 = X_1 + \text{Attention}(X_2) \qquad Y_2 = X_2 + \text{FeedForward}(Y_1)$$

- 구현 소스를 보니 X1과 X2는 같은 값 (배치, 토큰 길이, 모델 히든사이즈)
  Forward 과정에서 Y1과 Y2를 차원별 평균값을 다음 레이어에 전달
- FeedForward Layer는 토큰의 위치에 상관없이 연산하기 때문에 메모리 사용량을 줄이기 위해서 전체 토큰을 임의의 사이즈로 잘라서(chunk)순차적으로 연산
- 계산량은 많아지나 메모리 사용량이 chunk 사이즈에 따라 줄어듦

Model Type	Memory Complexity	Time Complexity
Transformer	$\max(bld_{ff}, bn_h l^2)n_l$	$(bld_{ff} + bn_h l^2)n_l$
Reversible Transformer	$\max(bld_{ff}, bn_h l^2)$	$(bn_h l d_{ff} + bn_h l^2)n_l$
Chunked Reversible Transformer	$\max(bld_{model}, bn_h l^2)$	$(bn_h l d_{ff} + bn_h l^2)n_l$
LSH Transformer	$\max(bld_{ff}, bn_h l n_r c)n_l$	$(bld_{ff} + bn_h n_r l c)n_l$
Reformer	$\max(bld_{model}, bn_h l n_r c)$	$(bld_{ff} + bn_h n_r l c)n_l$

그림 : 각 기법별 Memory Complexity, Time Complexity

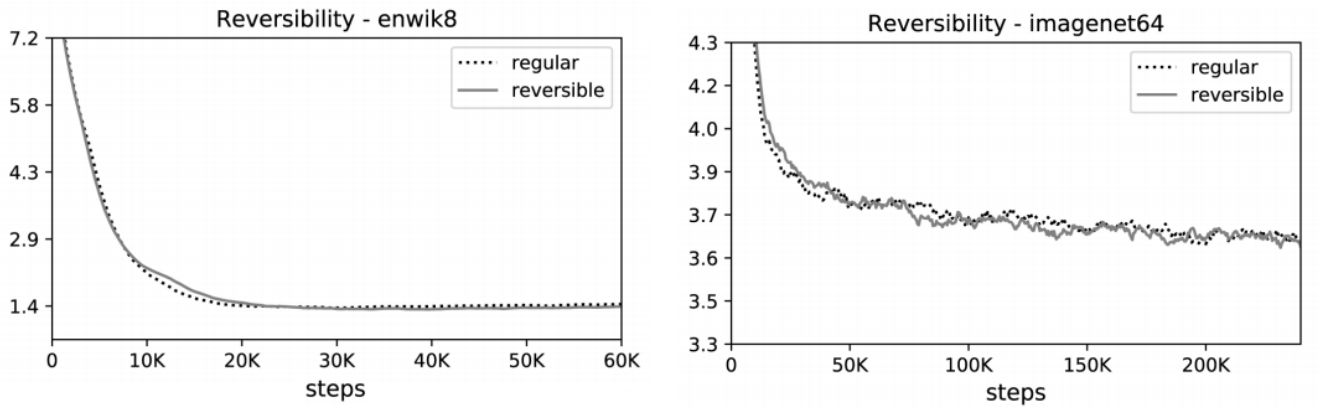

그림 : Residual Network를 사용했을 때와 Reversible Residual Network을 사용했을 때의 정확도 실험 결과