# Reformer : The Efficient Transformer

*ICLR 2020, Google Research*

**2020522113 정지훈**

# INDEX

# 1. Introduction

01 **Attention is All you need**

02

03

좋은 성능 → 길고, 깊게 → 데이터, 컴퓨팅 코스트 → Winner : The Rich??

# 1. Introduction

01 **Reformer Contribution**

02 **Transformer의 한계**

03

1. 어텐션 기반 모델이 갖는 메모리 문제

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V \Rightarrow O(N^2)$$

*Input Length가 64K라고 하면, batch_size가 1이라 해도 16GB가 필요*

2. N-stacked Residual Connection에 의한 메모리 문제

FeedForwad 와 BackPorpagation을 통해 weight들이 업데이트가 된다. 입력과 출력을 저장하기 위한 메모리가 많이 사용된다. Transformer는 N개의 레이어를 쌓는데 Attention Layer와 FeedForward가 결합돼 있으므로 중간 결과를 저장하는여 N배에 달하는 메모리가 필요하다.

**Reformer**

1. Locality-Sensitive Hashing(LSH)

유사한 벡터끼리 Hashing하여 어텐션을 계산
O(n^2) -> O(n*log(n))

2. Reversible Layer

출력만을 이용해 입력을 다시 복원할 수 있는 방법을 적용(2017, An Gomez)

**-> 메모리 문제를 해결(단일 GPU로도 준수한 성능)**
**-> 기존 트랜스포머 모델과 동등한 성능을 유지**

# 2. Background Knowledge : Transformers(Attention Is All You Need)

01

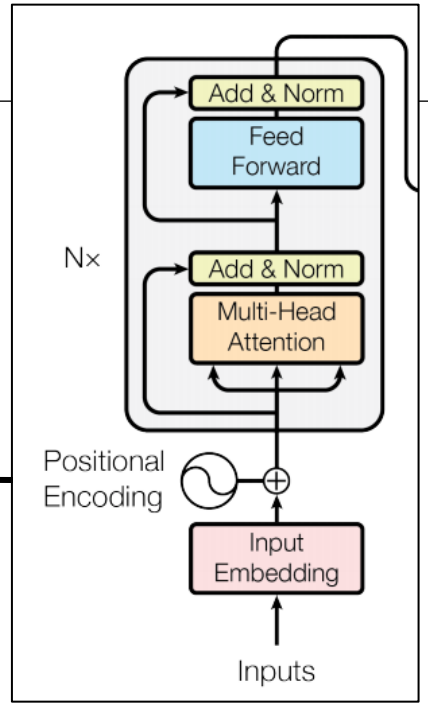
$$Self - Attention(Q,K,V) = softmax\left(\frac{(Q * K^T)}{\sqrt{d_{model}}}\right) * V$$

02

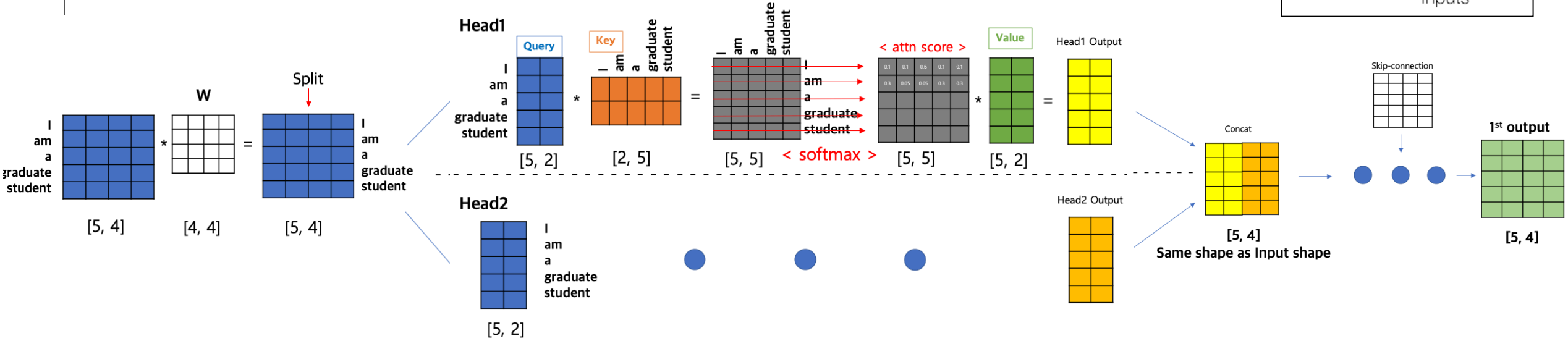
$$Q = K = V \; from \; inputs$$

03

04



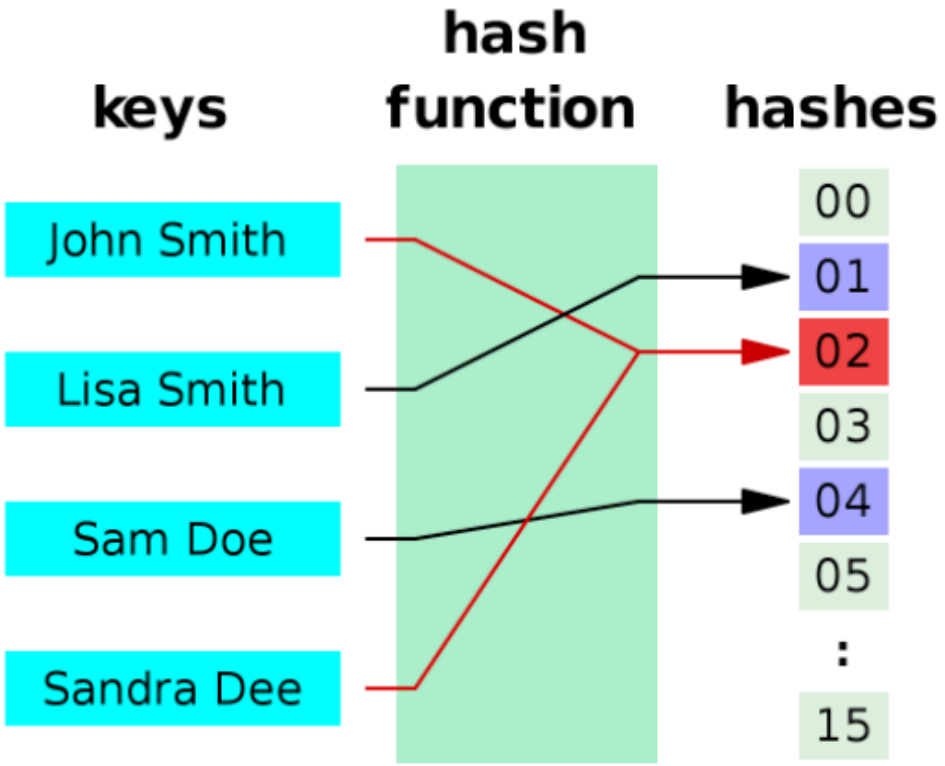

## 1. 어텐션 기반 모델이 갖는 메모리 문제



# 3. Reformer-LSH(Locality Sensitive Hashing)

01
02
03
04

## Hashing




## Hashing

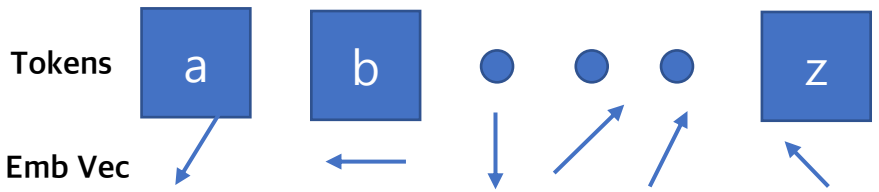
이름을 번호로 맵핑, 이름을 사번으로 맵핑…

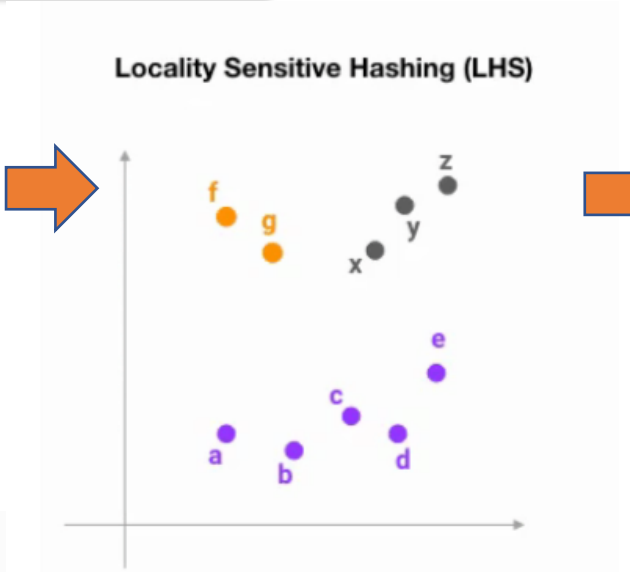
Hash는 연결된 데이터와 관련이 없음.
-> 데이터의 상대적 위치를 확인할 수 있는 방법을 생각
-> 가까운 데이터 끼리는 가까운 Hash값을 갖도록 구성
-> Locality Sensitive Hashing

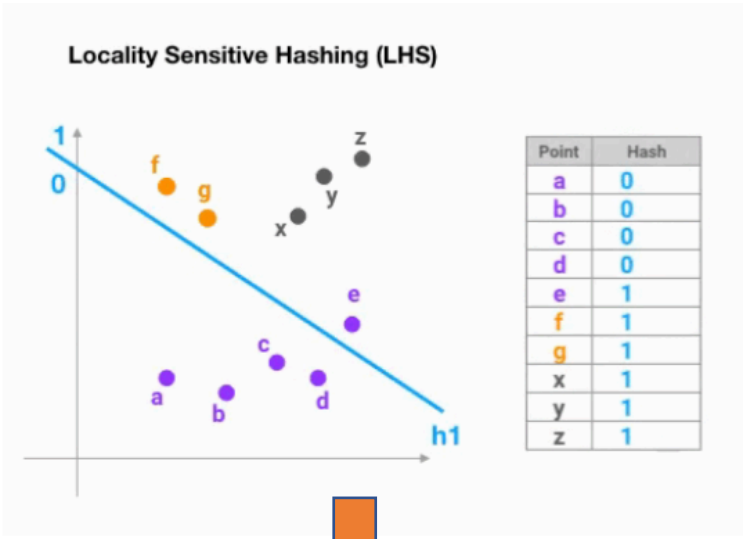
# 3. Reformer-LSH (Locality Sensitive Hashing)

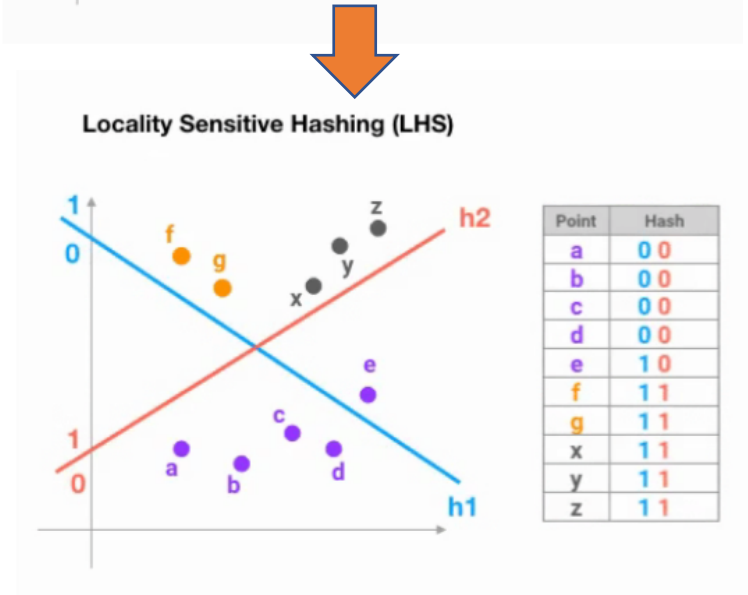
$$Self - Attention(Q,K,V) = softmax\left(\frac{(Q * K^T)}{\sqrt{d_{model}}}\right) * V$$

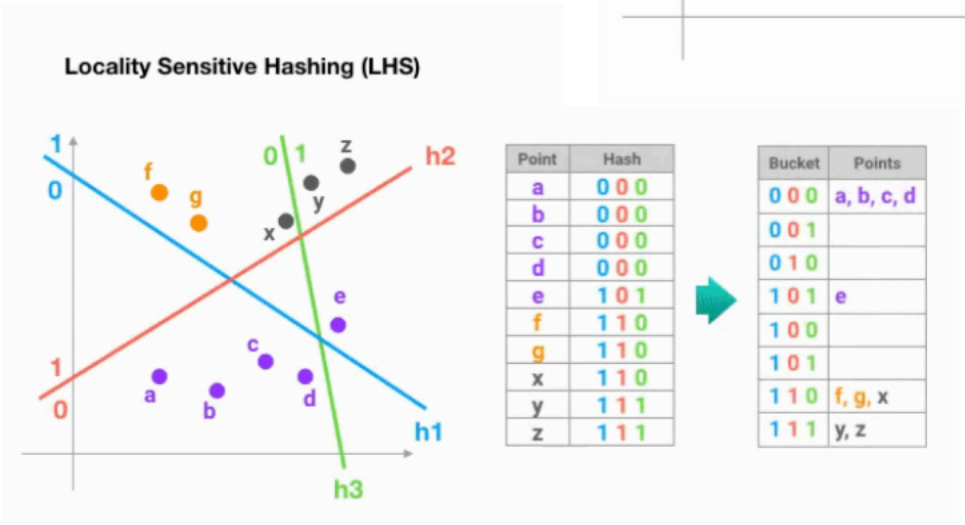
Softmax로 인해 Attention은 결국 QK의 큰 값에 의해 결정
-> 따라서 가까운 값을 갖는 Vector끼리 연산값만 중요
-> 64K input이 있더라도 Query와 유사한 토큰만 유의미한 Attention Score를 갖음



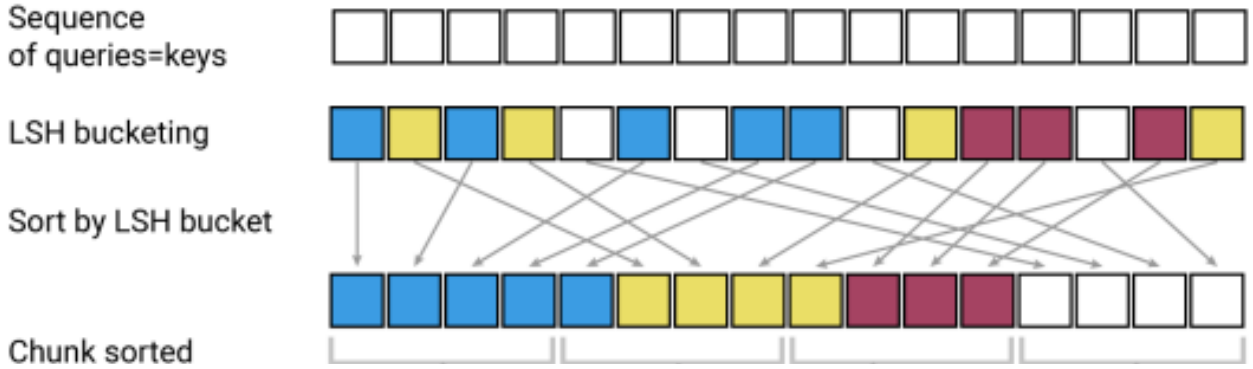



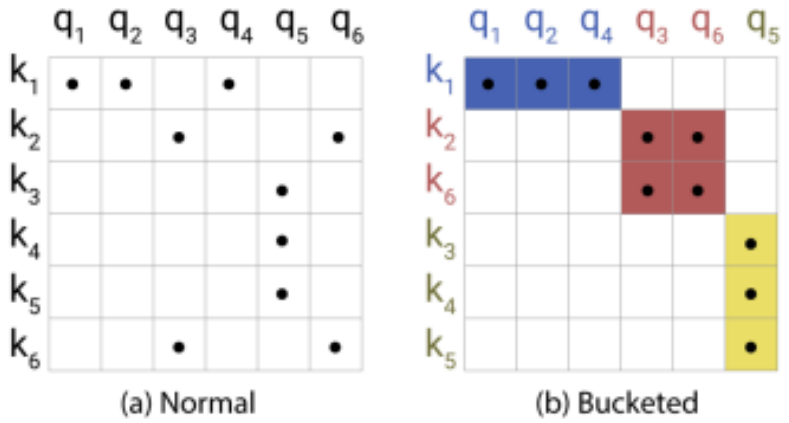



# 3. Reformer-LSH(Locality Sensitive Hashing)

01
02
03
04




1. 초평면을 통해서 몇개의 버킷으로 나눔

2. 같은 버킷끼리 그룹화(Sorted By LSH bucket)

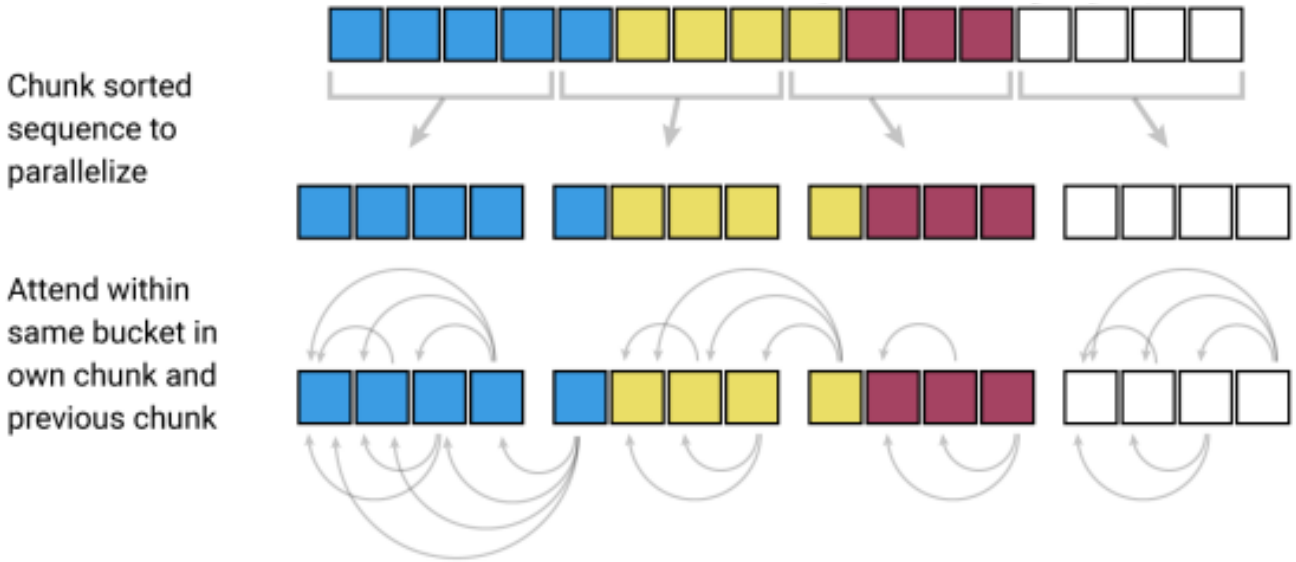
# 3. Reformer-LSH(Locality Sensitive Hashing)

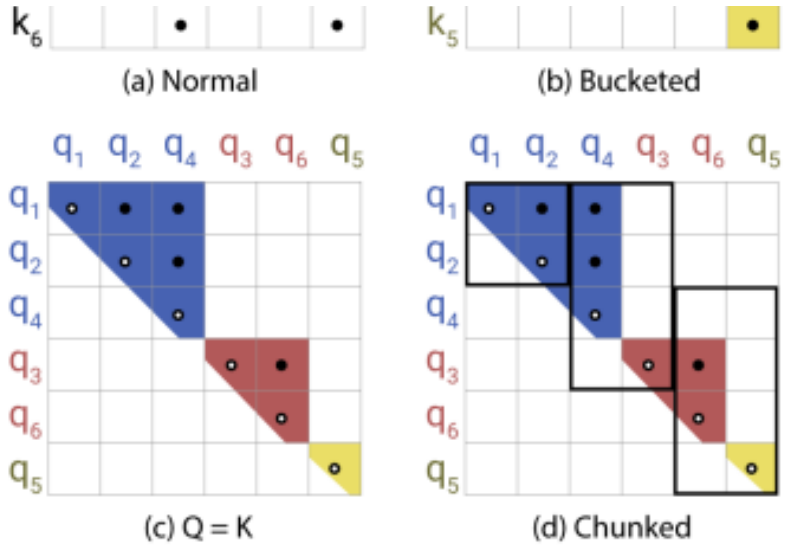
01

02

03

04




**3. Chunk단위로 나눔**

**4. 같은 Chunk단위끼리 어텐션을 계산하고, 뒤에 있는 Chunk와 계산을 함.**
**Chunk내에서 Full Attention을 계산하지 않고, i번째보다 뒤에있는 tensor와 계산을 한다.**
*(이유, Full Attention을 하게되면 스스로를 참조하게 되고 가장 높은 스코어값이 나오기 때문에 본인을 참조하지 않게 계산)*
**단, Chunk내에 하나의 Hash만 있을 경우 self-Attention을 한다.**

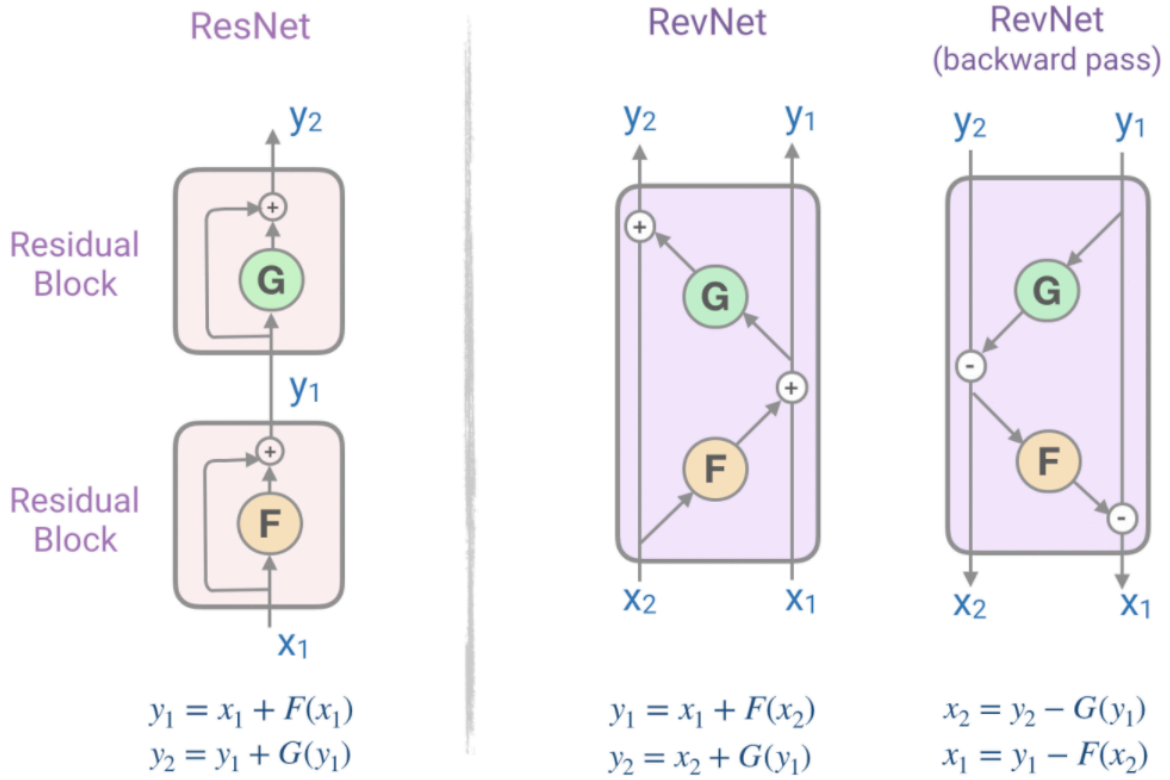
# 3. Reformer-Reversible Residual Layer

01
02
03
04

## 2. N-stacked Residual Connection에 의한 메모리 문제

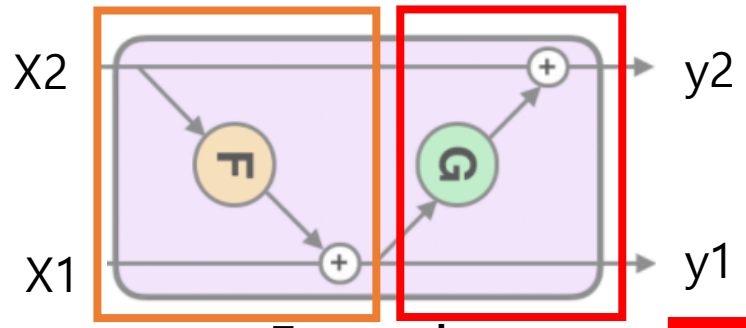
### Reversible Residual Layer(2017, Aidan N)

- Residual Connection의 효율화 버전
- 시간이 오래 걸려도 메모리 사용량을 줄임
- 마지막 엑티베이션 값만 있다면 아웃풋에서 인풋까지 복원 가능
- 계산량이 33% 증가



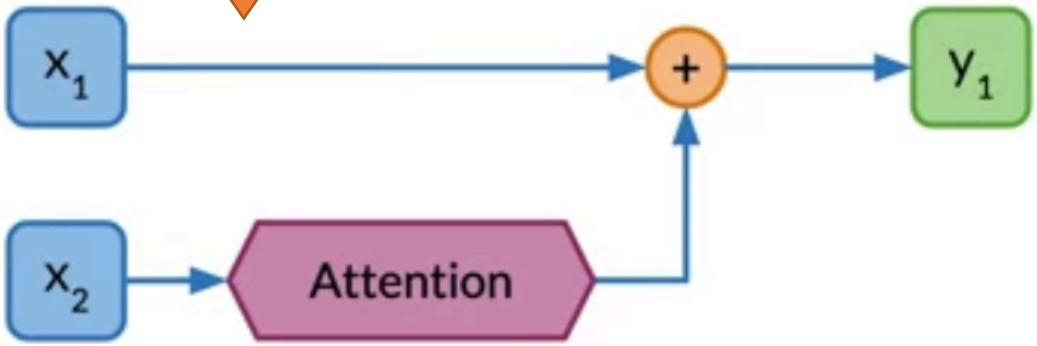
# 3. Reformer-Reversible Residual Layer

01
02
03
04

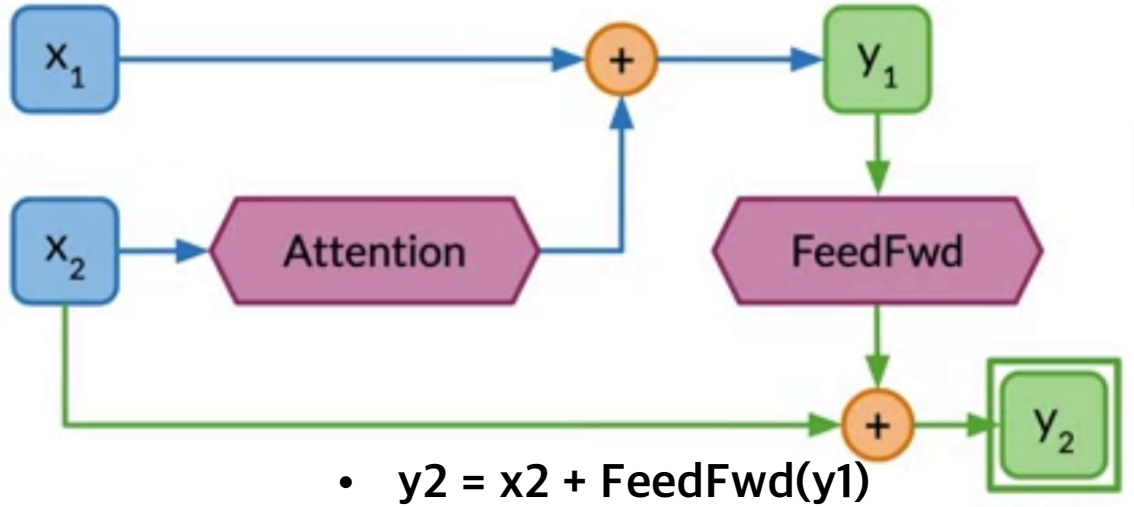



<Forward>

1) 첫번째

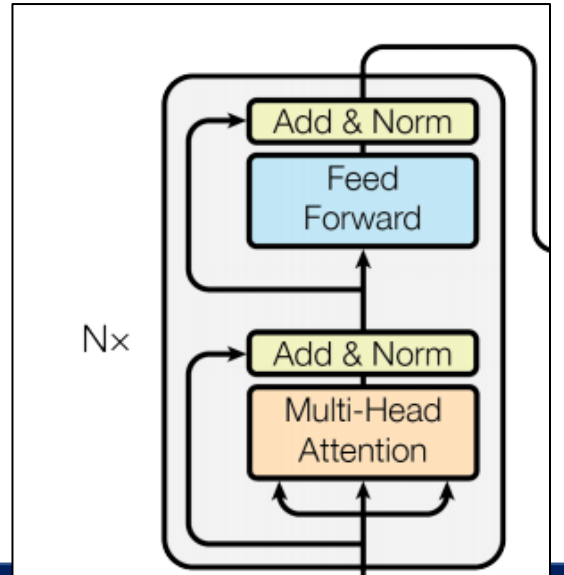



2) 두번째



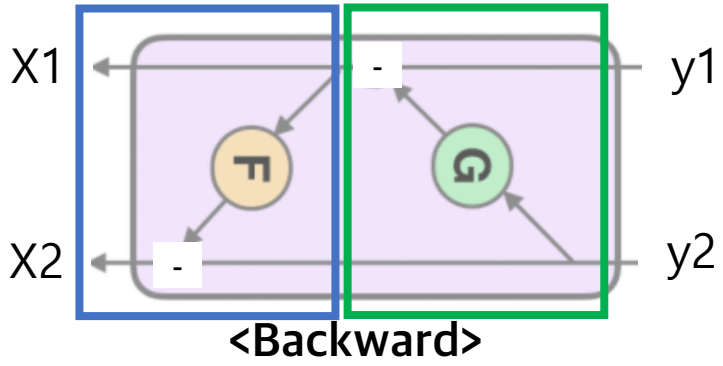

- y1 = x1 + Attention(x2)
- Input x로 부터 x1, x2가 복제됨
- y2 = x2 + FeedFwd(y1)

<Transformer Encoder>

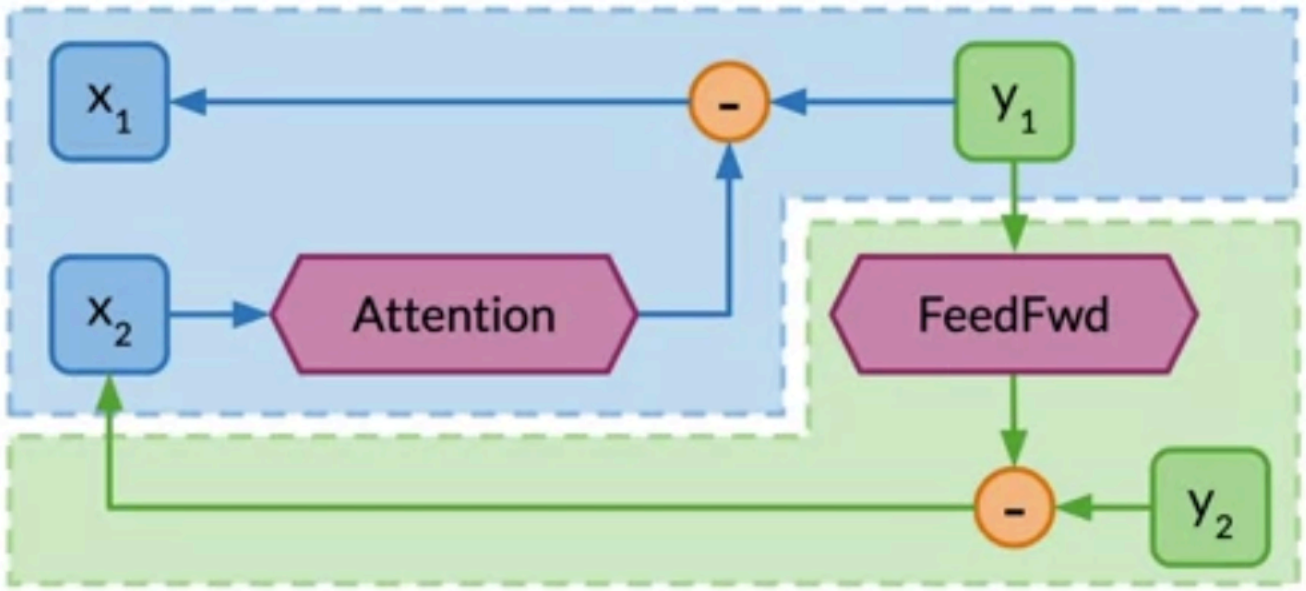


# 3. Reformer-Reversible Residual Layer

01
02
03
04



<Backward>



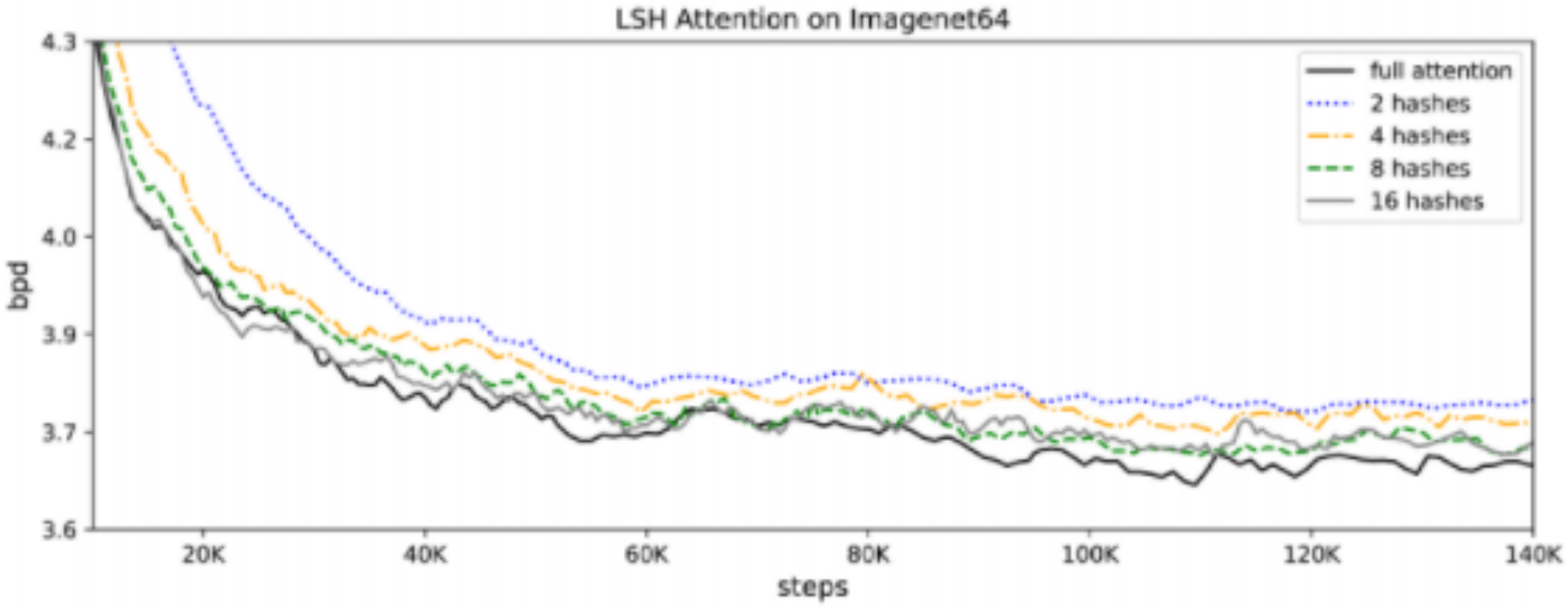

- x2 = y2 - FeedFwd(y1)
- x1 = y1 - Attention(x2)

=> 마지막 엑티베이션 값만 있다면 아웃풋에서 인풋까지 복원 가능

# 4. Experiments

01
02
03
04

## LSH Attention의 활용

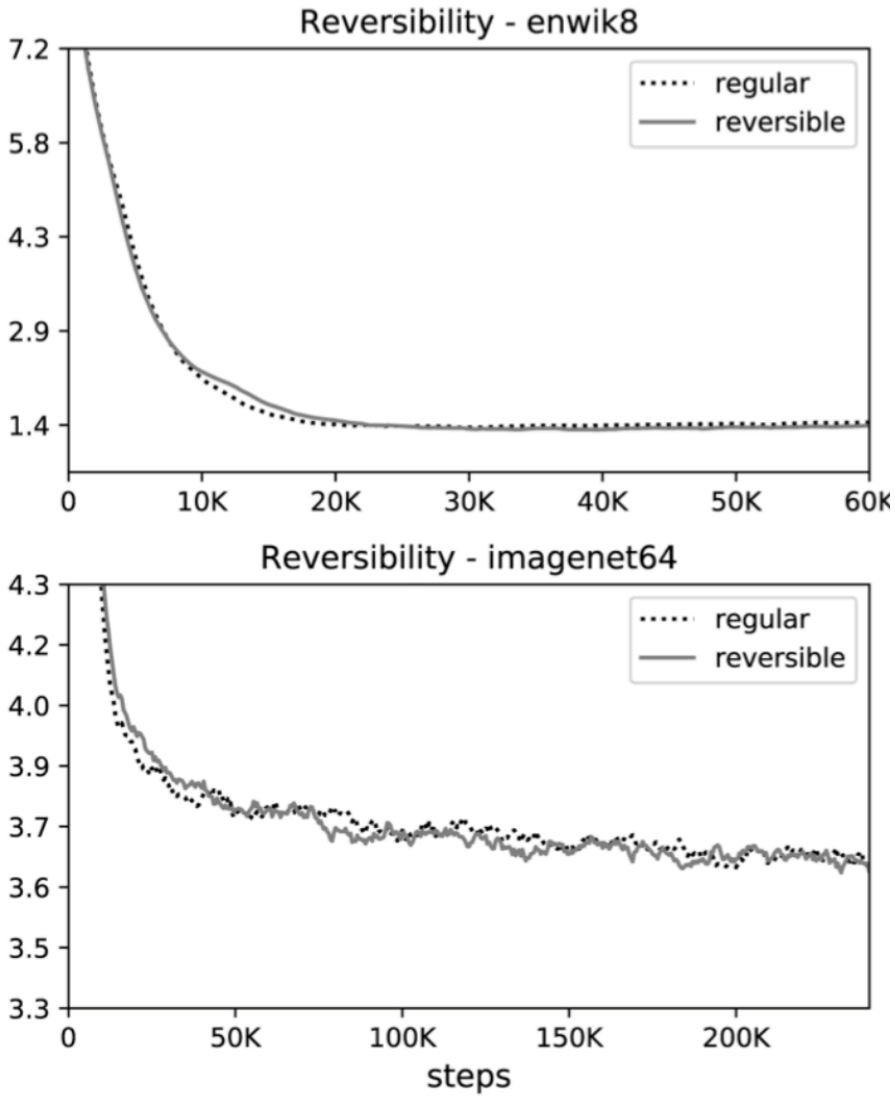



Hash를 많이 둘수록 Full Attention을 한 경우와 성능 차이가 줄어듦.
Hash를 8개 이상 두면 Full Attention할 때와 거의 비등한 절대 성능을 보임.

# 4. Experiments

01
02
03
04

## Reversible Layer




실험 결과, Reversible 형태로 구성한 구조와 그렇지 않은 구조 사이의 성능 차이는 미미함. 즉, Reversible 구조의 활용은 성능에 영향을 주지 않는다.

중간 과정을 기억하지 않아도 성능에 큰 변화가 없음

# 4. Experiments

01

## Memory/Computation Cost

02

Table 3: Memory and time complexity of Transformer variants. We write $d_{model}$ and $d_{ff}$ for model depth and assume $d_{ff} \geq d_{model}$; $b$ stands for batch size, $l$ for length, $n_l$ for the number of layers. We assume $n_c = l/32$ so $4l/n_c = 128$ and we write $c = 128^2$.

03

| 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 | $\max(bld_{ff}, bn_h l n_r c)n_l$ | $(bld_{ff} + bn_h n_r lc)n_l$ |
| Reformer | $\max(bld_{model}, bn_h l n_r c)$ | $(bld_{ff} + bn_h n_r lc)n_l$ |

04

**Execution of Reformer requires much lower memory consumption and achieves impressive performance even when running on only a single GPU**

# Thank you