<https://baekyeongmin.github.io/paper-review/transformer-xl-review/>

<http://mlgalaxy.blogspot.com/2019/07/transformer-xl.html>

**Transformer-XL: Attentive Language Models Beyond a Fixed-Length Context3(ACL, 2019)**

기존의 Fixed Length LM의 한계점 지적 + 더 긴 의존성 이용이 핵심

다양한 NLU task에서 SOTA + XLNet과 같은 논문에서 활용(사실 같은 저자)

사실 기존 Transformer에서 진작에 의문이 들었던 점이다.  
바로, 디코더의 입,출력에서도 seq\_len가 fixed 된다는 점이다.(물론 그 값이 크긴 하지만)

**Main Idea**

기존 트랜스포머는 코퍼스를 여러 개의 세그먼트로 나눈다. (어떤 기준으로 나누는지 모르겠음, 한 세그먼트 내에 관련 없는 문장끼리 들어 있을 수도 있나? 아니면 한 문장이 매우 길다면 중간에 끊길수도 있나?)

여튼 seq\_len이 정해지기 때문에 그 내에서만 학습이 가능하다. => 매우 긴 corpus는 같이 학습이 안돼서 long-term dependency를 학습 못할수도 있다.

또한 context fragmentation: 단순하게 연속적인 symbol들로 구성 => 해당 세그먼트의 처음 몇개의 symbol들을 예측하기에는 정보 양 부족 => 중간에 짤렸을 때 그 다음 세그먼트의 앞 몇 단어는 갑자기 나타난 것이라 무슨 의미인지 알 수가 없다.

=> 긴 문장은 Transformer-XL(extra long) 필요

**기존 Transformer**

2017, Attention is all you need에서 소개(RNN 기반의 long-term dependency & GVP 해결)

또한, character-level language modeling with deeper self-attention(2018)에서는 char LM을 transformer로 해결했는데 RNN 기반보다 좋은 성능이지만 문제가 있다.

1. 고정된 길이의 세그먼트 => 해당 길이 이상의 의존성을 학습하기 힘듬.
2. 각 세그먼트의 의미적 경계( ex) 긴 문장이 한 세그먼트다, 이 문장까지가 일관성 있는 한 문단이다.) 등을 고려하지 않고 / corpus에서 연속적인 token을 잘라서 사용 => context fragmentation 문제

**Transformer XL**

두 문제를 해결하기 위해 Transformer의 Recurrence 형태를 제시한다.

연속된 세그먼트들을 모델링 할 때, 세그먼트를 독립적으로 모델링(기존 트랜스포머)하는 것이 아니라 특정 세그먼트의 모델링에 이전 세그먼트의 정보를 이용한다.

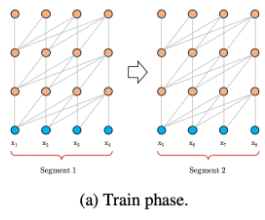
이에 따라 세그먼트들의 의존성을 파악할 수 있고 자연스럽게 고정된 길이의 의존성 문제 & context fragment 문제를 동시에 해결하게 된다. (기존 트랜스포머가 LSTM의 Long dependency & GVP를 해결했지만 XL은 한번 더 그 부분을 증가시킴)

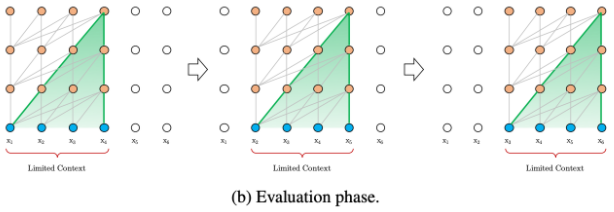
역시 기존 트랜스포머처럼 의 auto-regressive 방식으로, P(x)의 분포를 추정하는 방식이다.

기존 트랜스포머로 매우 긴 컨텍스트 모델링 하려면?

그냥 그 컨텍스트를 한번에 때려 넣으면 된다. => But 메모리/계산상의 한계로 불가능

따라서 여러 개의 세그먼트로 나눈다. (limited length 보다 짧게 해서)



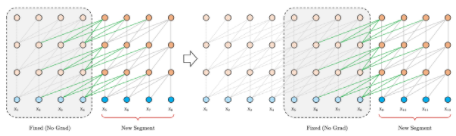


첫번째처럼 Train & Evaluation 이 이루어지는데 여기서 문제가 되는 것이 계속 말했던 segment들 간의 dependency를 배울 수 없다는 것이다. (=long-term dependency를 완전히 학습 불가)

또한 평가를 할 때는 두번째처럼 1칸씩 옮기는데(이거는 처음 앎), 이 방법은 아마 limited length 때문에 세그먼트간의 관련성을 sliding window로 거쳐서 평가하기 위함인 것 같다. 하지만 그만큼 계산량을 요구하고 느리다.

**Segment-Level Recurrence with State Reuse**

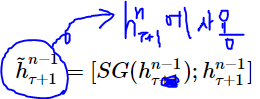
기존 문제를 해결하기 위해 recurrence 방법 사용



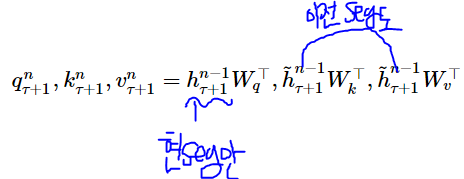
위 그림처럼 세그먼트 결과를 다음 세그먼트가 이용할 수 있도록 iteration 마다 caching 한다.

그렇게 되면 현재 세그먼트는 직전 세그먼트의 정보를 이용할 수 있게 된다.

즉 수식으로 나타내면

과 같다. SG는 Stop Gradient로 직전 세그먼트의 파라미터 학습에는 영향을 안끼친다.(Gradient 전파 없다.)

현 레이어에서 사용될 QKV를 방금 만든 hidden state로 만들어야 한다.



Query는 현 Segment에 대한 정보만 담고 있으면 된다. 마지막으로 디코더를 거쳐 다음 layer에 쓰일 입력이자 현 레이어의 출력을 만들어 낸다.



평가를 할 때도 이런식으로 하면 이전 세그먼트 정보 자동으로 얻기 때문에 sliding window 사용 안해도 된다. => 1800배 빠르다.

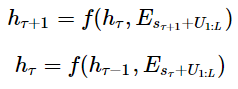
또한 리소스가 좋다면(메모리 등) 이전 세그먼트 길이를 더 길게 가져가 더 긴 의존성 만들 수 있다. (=memory augmented neural net의 메모리와 비슷한 역할)

또한 모델 구조 설정에 따라 segments들의 의존성 길이를 구할 수 있다.(segment length \* layer) (위 같은 경우 4\*3이다.) 길수록 이전 segment의 길이 단위가 길어 더 많은 내용을 참고

**Relative Positional Encodings**

기존 트랜스포머는 토큰 위치의 Positional Encoding 값을 더한다.

이것을 recurrence에 맞춰서 적용할려면 어떻게 해야 하나?

처럼 된다. E는 그냥 word embedding이다.

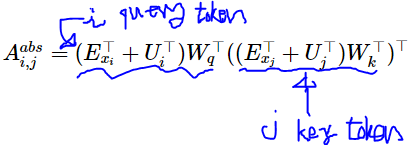
세그먼트마다 U가 동일하여 모델이  는 구분할 수 없다. (이 토큰이 어떤 세그먼트에서 왔는지 판단 못하나?)

Positional encoding은 모델에게 토큰의 위치에 대한 단서/bias를 제공!

각 layer의 어텐션 스코어에 위치 정보를 포함시킨다. (기존에 초기 임베딩에 포함시키는 것 대신에)

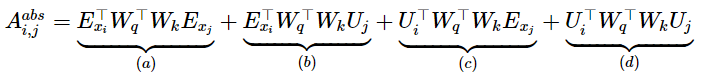
또한 어텐션에서는 쿼리, 키 벡터 사이의 유사도 계산 시(아까 위에서 한것처럼) 절대적 위치 보다는 상대적인 거리를 아는 것이 중요하다.

따라서 Recurrence한 상황에서는 절대적 방식이 아닌 상대적 방식이 직관적이고 일반화 가능이다.

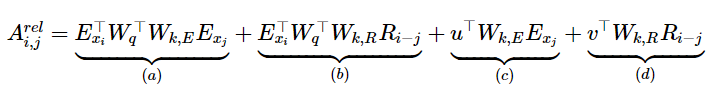
 이 식은 기존의 어텐션 계산이다.

쿼리의 i 번째 토큰과 키의 j 번째 토큰의 어텐션을 구하는 식이다.

풀어서 쓰면



인데 4가지 term이 있고 이것을 relative하게 바꾸면



이 된다.

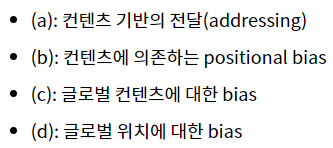
이때 두 토큰 사이의 거리는 최소 0 ~ 최대 Lmax 이다.(아마 seq\_len 인듯)

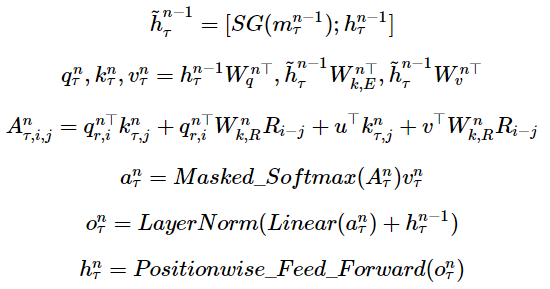
따라서 절대적인 positional encoding 인 U들을 상대적인 거리 encoding인 로 바꿨다.

또한 c,d terms에서 u, v를 볼 수 있는데 U가 없어져서 이 값을 계산할 수 없다. 그리고 query 벡터의 단어의 위치와 상관없이 같은 값을 갖기 때문에 이렇게 대신 가능... 무슨말인지 잘 몰르겠다. 그냥 현 segment가 기준이니 이것은 고정해도 된다는 뜻인듯?

또한, W-k가 W-k,E & W-k,R 로 바뀌었는데 이는 임베딩 어텐션 Query, Key를 구하기 위해 각 word 임베딩과 상대 위치 임베딩을 따로 하겠다는 소리인듯

이 부분 넘나 어렵

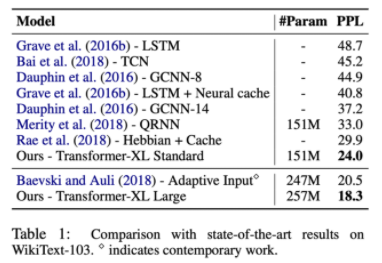




= 전체 flow

가장 초기 입력은 단어 임베딩만 => 

**Experiments**



일단 LM으로 Test => 바로 SOTA

**Ablation Study**

Recurrence & relative positional encoding 각각 빼면서 실험

실험부분이랑 relative positional encoding은 다시보기