NLP 기초

Part 1. NLP(Natural Language Processing, 자연어처리) 기초 다지기 위해 강의 수강

- https://www.youtube.com/watch?v=dKYFfUtij_U&list=PLVNY1HnUlO26qqZznHVWAqjS1fWw0zqnT
  - Bag of Words
  - n-gram
  - TF-IDF
  - 자연어처리의 유사도 측정 방법(거리측정, 코사인 유사도)
  - TF-IDF 문서 유사도 측정
  - 잠재의미분석(LSA - Latent Semantic Analysis)
  - Word2Vec
  - WMD 문서 유사도 구하기(word mover's distance)
  - RNN 기초(순환신경망 - Vanila RNN)
  - LSTM 쉽게 이해하기
  - Sequence to Sequence + Attention Model
  - Transformer(Attention is all you need)

1. Bag of words

- 문장을 숫자로 표현하는 방법 중 하나.
- Sentence similarity : 문장의 유사도를 구할 수 있음.
- 머신러닝 모델의 입력 값으로 사용할 수 있다.(문자를 숫자형으로 바꿔주어서 입력값으로 사용 가능하다)
- 한계(Limitation)
  - `Sparsity` : If we use all English words for bag of words, the vector will be very long, but very few non zeros.
    - 실제 문장 하나를 표현할 때 0이 무수히 많기 때문에 계산량이 높으며 메모리도 많이 사용한다.
  - Frequent words has more power : 많이 출현한 단어는 힘이 세진다.
  - Ignoring word orders : 단어의 순서를 철처히 무시한다. 단어의 출현 순서를 무시하기 때문에 문맥이 무시된다. 예시로 home run vs run home 같은 것으로 인식함.
  - Out of vocabulary : 보지 못한 단어들은 처리하지 못한다.

2. n-그램(n-gram)

- n-그램은 연속적으로 n개의 토큰으로 구성된 것을 의미함. 토큰은 자연어 처리에서 보통 단어나 캐릭터로 얘기함.

- 1-gram == unigram

  - EX) fine thank you
  - Word level : [fine, thank, you]
  - Character level : [f,i,n,e, ,t,h,a,n,k, ,y,o,u]

- 2-gram == bigram

  - EX) fine thank you

    - Word level : [fine thank, thank you] -> fine you는 붙어있지 않아서 토큰이 안됨.
    - Character level : [fi,in,ne,e , t,th,ha,an,nk,k , y,yo,ou]

- 3-gram == trigram

    - EX) fine thank you
    - Word level : [fine thank you]
    - Character level : [fin,ine,ne, et, th,tha,han,ank,nk, ky, yo,you]

`Why n-gram?`

- bag of words는 단어의 순서가 철저하게 무시되는 단점이 있는데 이를 n-gram으로 극복할 수 있다.

    - bag of words' drawback(단점)
    - EX) machine learning is fun and is not boring
    - `bag of words` : (machine,fun,...boring) -> (1,1,2,1,1,1,1)
    - `n-gram` : (machine learning, learning is, is fun, fun and, and is, is not, not boring) -> (1,1,1,1,1,1,1)
    - not boring as token now! : not boring을 인식 잘하게 될 수 있다.

- Next word prediction : 다음 단어가 무엇이 올지 예측가능하다.

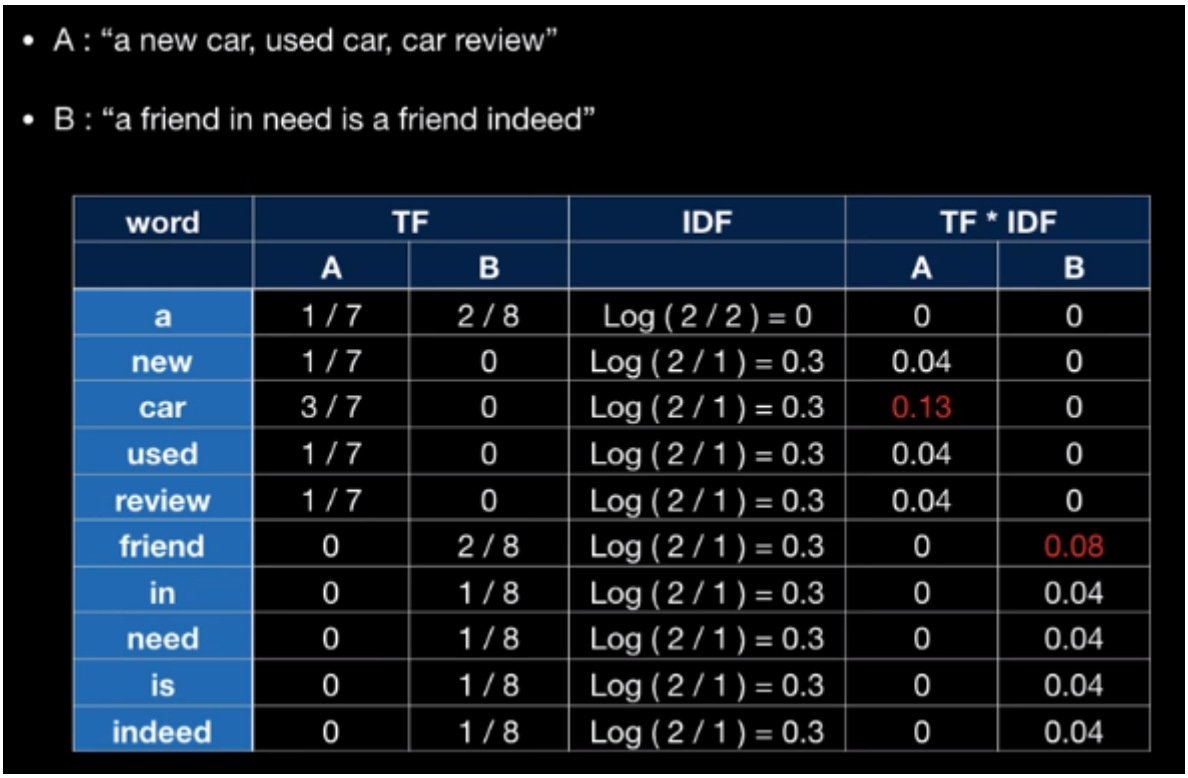
    - Naive Next word prediction
    - EX) how are you doing, how are you, how are they -> Trigram(how are you, are you doing, how are they) = (2,1,1)
    - how are ??? : ???을 예측하려면 어떤 단어를 예측할까? -> `you` [나이브한 방법일 때]

- Find Misspelling

    - Naive한 Spell checker 방법
    - EX) quality, quater, quit -> Bigram(qu,ua,al,li,it,ty,at,te,er,ui) = (3,2,1,1,2,1,1,1,1,1)
    - q`w`al => w가 잘못 되고 qual로 스펠링 체크해줄 수 있음(나이브한 방법)

3. TF-IDF

- 용어
    - TF : Term(단어) Frequncy
    - IDF : Inverse Documnet Frequency
- TF-IDF를 왜 이용할까?
    - To find how `relevant` a term is in a document : 한 문서는 단어로 구성되어 있다. 각 단어별로 문서에 관한 연관성을 알고 싶을 때 TF-IDF를 이용한다.
    - 각 단어별로 문서에 대한 정보를 얼마나 갖고 있는지를 TF-IDF 수치로 나타낸 것이다.
- `Term Frequncy`
    - TF measures how frequently a term occurs in a document : 문서에서 단어가 몇번 출현했는지.
    - TF를 사용 시 가정
        - If a term occurs more times than other terms in a document, the term has more relevance than other terms for the documnet. : 문서가 있을 때 한 단어가 여러번 출현했을 때 그 단어는 문서와 연관성이 높을 것이다.
    - EX1) "a new car, used car, car review"

    - word(a,new,car,used,review) -> TF(1/7,1/7,3/7,1/7,1/7) : car가 이 문장과 가장 연관성이 높은 것으로 보임.
    - TF measure의 치명적인 단점(drawback)
        - EX2) "a friend in need is a friend indeed"
        - word(a,friend,in,need,is,indeed) -> TF(2/8,2/8,1/8,1/8,1/8,1/8)
        - EX1과 동일하기 TF Score를 구했는데 중요하지 않은 a가 friend와 같이 가장 높았음. -> TF의 가정이 틀림. 따라서 IDF의 개념이 나오게 된다.
- `IDF(Inverse Documnet Frequency)`
    - `LOG(Total # of Docs / # of Docs with the term in it)` : 총 문장의 개수를 단어가 출현한 문장의 개수로 나누어 준것에 LOG를 취한것.
    - LOG(Total # of Docs / # of Docs with the term in it `+1`) : 때로는 0으로 나누는 것을 방지하기 위해 +1을 더해주기도 함(smoothing)
    - A : "a new car, used car, car review"
      B : "a friend in need is a friend indeed"
    - 

word	TF		IDF	TF * IDF	
	A	B		A	B
a	1 / 7	2 / 8	Log (2 / 2) = 0	0	0
new	1 / 7	0	Log (2 / 1) = 0.3	0.04	0
car	3 / 7	0	Log (2 / 1) = 0.3	0.13	0
used	1 / 7	0	Log (2 / 1) = 0.3	0.04	0
review	1 / 7	0	Log (2 / 1) = 0.3	0.04	0
friend	0	2 / 8	Log (2 / 1) = 0.3	0	0.08
in	0	1 / 8	Log (2 / 1) = 0.3	0	0.04
need	0	1 / 8	Log (2 / 1) = 0.3	0	0.04
is	0	1 / 8	Log (2 / 1) = 0.3	0	0.04
indeed	0	1 / 8	Log (2 / 1) = 0.3	0	0.04

4. 자연어 유사도 측정(Euclidean Distance cosine similarity)

- 자연어 처리에서 유사도 측정하는 방법 2가지
    - 1.Euclidean Distance
    - 2.cosine similarity
    - 영상 참고

5. TF-IDF 문서 유사도 측정

- How to get document similarity?
    - Cosine Similarity on Bag of Words
    - Cosine Similarity on TF-IDF with Bag of Words -> 이걸 쓰는게 좋음.
    - 영상 참고
- IF-IDF with Bag of words의 장점

    - Easy to get document similarity : 문서의 유사도를 구하기 쉽고 구현하기 쉽다.
    - Keep relevant words socre : 중요한 단어의 점수를 유지한다.
    - lower just frequent words score : 자주 출현하지만 여러 문서에 등장한다면 점수를 줄여준다.
- IF-IDF with Bag of words의 단점
    - Only based on Terms(words) : 단어만 본다. 단어의 유사성 같은건 안봄
    - Weak on capturing document topic : 단어는 알되 그 topic을 아는 것은 한계가 있음.
    - Weak handling synonym(different words but same meaning) : 이음동이의를 처리하기 힘듬.
- IF-IDF with Bag of words의 단점을 보안할 수 있는 방안
    - LSA(Latent Semantic Analysis)
    - Word Embeddings(Word2Vec, Glove)
    - ConceptNet - knowledge 그래프 사용

6. 잠재 의미 분석(LSA)

- LSA similarity is based on `topic` : IF-IDF with Bag of words는 word(단어) 기반이지만 LSA는 topic 기반이다.
- 영상 참고

7. WMD(Word mover's distance) 문서 유사도 구하기

- WMD는 Word2Vec의 유클리디안 거리를 사용한다.
- What is Word2Vec?
    - Word Embedding
    - Word similarity comes from the word's neighbor words : 한 문장에서 그 단어의 이웃들로 계산이 된다.
    - You also can easily download pre-trained word2vec
    - GoogleNews-vectors-negative300.bin.gz
- Word2Vec example
    - 영상 참고
- WMD 핵심 아이디어
    - Normalized Bag of Wors after stop words removal
- WMD 단점
    - 계산 속도가 굉장히 느리다. -> RWMD(Relaxed WMD)에서는 조금 개선되서 $p^3logp;에서;p^2;만큼;빨라짐$; p는 문장 내 unique 단어 개수.

8. Tensor란?

- tensor example in `NLP`
    - ex) (3,2,4)[3D tensor] = `(sample dimension,max length of sentence, word vector dimension)` : 3개의 샘플(문장)을 갖고 있는데, 문장 내 단어 개수는 2개이며, word들은 총 4개로 이루어져 있다.
- tensor example in `grayscale image`

    - (3,5,5)[3D tensor] = `(you have 3 images, 5 rows, 5 columns)` : 3개의 이미지를 갖고 있는데 이미지는 5x5 행열로 이루어져 있다
- tensor example in `rgb color image`
    - (3,5,5,3)[4D tensor] = `(you have 3 images, 5 rows, 5 columns, red-green-blue)` : 3개의 이미지를 갖고 있고 이미지는 5x5 행열로 이루어져 있는데 3개의 색으로 구성되어 있다.
- tensor example in `rgb color video`
    - (3,5,5,5,3)[5D tensor] = `(you have 3 frames, 5 images, 5 rows, 5 columns, red-green-blue)` : 총 3개의 비디오가 있고 비디오 안의 frame은 5개의 이미지로 구성되어 있다. 그리고 그 이미지는 5x5 행열로 이루어져있고 3개의 rgb 색이 있다.

9. RNN(순환 신경망)

- 영상 참고

10. LSTM

- 영상 참고

11. Sequence to Sequence with Attention Model

- 영상 참고

12.Transformer(Attention is all you need)

- 영상 참고

1. Transformer는 기존 encoder decoder architecture를 발전시킨 모델이다.
2. RNN을 사용하지 않는다.
3. RNN 기반 모델보다 학습이 빠르고 성능이 좋다.(Faster, Better!)

- How faster?
    - Reduced sequence computation
    - Parallelization(병렬)

※ `Encoder`

기존 인코더, 디코더의 주요 컨셉을 간직하되 RNN을 없애서 학습 시간을 단축시켰으며 성능도 올렸다.

- Positional encoding
    - 단어의 위치와 순서 정보를 활용하기 때문에 rnn을 사용했는데 rnn을 제거 했기 때문에 Positional encoding을 사용한다.
    - Positional encoding이란 인코더 및 디코더 입력값마다 상대적인 위치정보를 더해주는 기술이다.
    - 장점 1 : 항상 Positional encoding의 값은 -1~1 사이의 값이다.
    - 장점 2 : 학습 데이터 중 가장 긴 문장보다도 긴 문장이 운영 중 들어왔을 때 에러없이 상대적인 인코딩 값을 줄 수 있다.

- Self Attention
    - Quary, key, value => vector의 형태

    - Quary * key = Attention Score
        - Score가 높을수록 단어의 연관성이 높고 Score가 낮을수록 연관성이 낮다.

- Residual Connection followed by layer normalization
- Encoder Layer에 입력 vector와 출력 vector의 차원의 크기는 동일하다. -> 이는 즉, Encoder Layer를 여러 개 붙혀서 사용할 수 있다는 뜻이다. Transformer는 Encoder Layer를 6개 붙혀서 만든 구조다.
- Transformer Encoder의 최종 출력 값은 6번째 인코더 레이어의 출력값이다.

※ `Decoder` Decoder는 Encoder와 동일하게 6개의 Layer로 구성되어있다.

- Label Smoothing -one hot encoding이 아님.

Question

- word embedding이랑 character embedding을 같이 사용하는 이유 -> word embedding만 사용하면 새로운 단어가 들어왔을 때 얼타는 경우가 있는데, character emmbedding을 사용하면 모든 문자와 특수문자 모두를 사용하기 때문에 조금 더 정확도가 올라간다.
- uni-LSTIM(한방향)이 아니라 Bi-LSTM(양방향)으로 학습하는 이유? -> 앞으로도 읽어보고 뒤로도 읽어보고 다양한 방법으로 context를 이해하기 위함.

세미나 발표 준비

발표 논문 : XLNet: Generalized Autoregressive Pretraining for Language Understanding

- paper url : https://arxiv.org/pdf/1906.08237.pdf
- 참고 사이트
    - https://www.youtube.com/watch?v=koj9BKiu1rU&t=1327s
    - https://www.slideshare.net/SungnamPark2/pr175-xlnet-generalized-autoregressive-pretraining-for-language-understanding-152887456
    - https://ai-information.blogspot.com/2019/07/nl-041-xlnet-generalized-autoregressive.html
    - https://blog.pingpong.us/xlnet-review/
    - https://ratsgo.github.io/natural%20language%20processing/2019/09/11/xlnet/
- 목표 : 전부 다 읽고 XLNet 이해하기

※ 읽기 전 알아두면 좋을 용어

- 1.state-of-the-art: https://www.stateoftheart.ai/ 에서 어떠한 과제에서 가장 우수한 모델을 제출했을 때, state-of-the-art를 기록했다고 하는 듯 하다. 단어 자체도 '최첨단의 기술' 이라는 의미와 일맥상통 한다.
- 2.`Transfer Learning`: 기존의 만들어진 모델을 사용하여 새로운 모델을 만들시 학습을 빠르게 하며, 예측을 더 높이는 방법이다. 이미 잘 훈련된 모델이 있고, 특히 해당 모델과 유사한 문제를 해결시 transfer learining을 사용한다.
- 3.`autoregressive(AR) language modelings`: 자기회귀 언어 모델링이라고도 하며, Sequence가 주어졌을 때 문장에게 점수를 부여하는 방법이며, 이전 단어가 주어졌을 때, 다음 단어가 나올 확률들을 전부 곱한 것이다. 이렇게 정의하면서 비지도 학습 문제를 지도학습으로 지도학습으로 바꿀 수 있다.

- 4.Transformer-XL: https://medium.com/dair-ai/a-light-introduction-to-transformer-xl-be5737feb13 을 참조.

참고 : https://catsirup.github.io/ai/2019/07/24/XLNet.html

※ XLNet 정리

참고 사이트 : https://ratsgo.github.io/natural%20language%20processing/2019/09/11/xlnet/

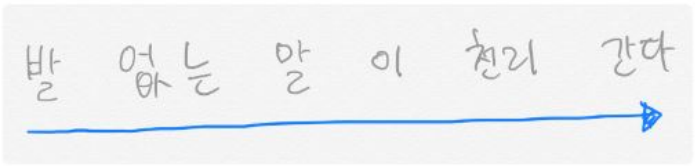
기본 내용

- XLNet은 트랜스포머 네트워크(Vaswani et al., 2017)를 개선한 '트랜스포머-XL(Dai et al., 2019)'의 확장판 성격의 모델입니다.
- 여기에서 XL이란 'eXtra-Long'으로, 기존 트랜스포머보다 좀 더 넓은 범위의 문맥(context)를 볼 수 있다는 점을 강조하는 의미로 쓰였습니다.

퍼뮤테이션 언어모델 (Permutaion Language Model)

- Yang et al.(2019) 최근 임베딩 모델의 두 가지 흐름 : 오토리그레시브(AutoRegressive, AR) 모델, 오토인코딩(AutoEncoding, AE) 모델
- 오토리그레시브(AutoRegressive, AR) 모델
  - AR 모델은 데이터를 순차적으로 처리하는 기법의 총칭을 뜻합니다.
  - 이 관점에서 보면 ELMo나 GPT를 AR 범주로 분류할 수 있습니다. 두 모델 모두 이전 문맥을 바탕으로 다음 단어를 예측하는 과정에서 학습하기 때문입니다.
  - 아래 예제의 문장을 학습하는 경우 AE 모델은 단어를 순차적으로 읽어 나갑니다.
  - Ex)

그림 1. 오토리그레시브(AR) 모델



- 오토인코딩(AutoEncoding, AE) 모델
  - AE 모델은 입력값을 복원하는 기법들을 뜻합니다. -> y=f(x)=x를 지향합니다.
  - 대표적인 AE 모델로는 BEAR가 있는데, BERT는 문장 일부에 노이즈(마스킹)을 주어서 문장을 원래대로 복원하는 과정에서 학습하는 모델입니다.
  - 즉, 마스킹 처리가 된 단어가 실제로 어떤 단어일지 맞추는데 주안점을 두는 것입니다.
  - 이런 맥락에서 BERT를 디노이징 오토인코더(Denoising Autoencoder)라고 표현하기도 합니다.
  - 디노이징 오토인코더란 노이즈가 포함된 입력을 받아 해당 노이즈를 제거한 원본 입력을 출력하는 모델입니다.

그림 2. 오토인코딩(AE) 모델

    - Ex)

- ※ Yang et al.(2019)의 AE,AR 모델의 **문제점 제안**
    - AR 모델의 문제점
        - 문맥을 양방향(bidirectional)으로 볼 수 없는 태생적인 한계가 있다.
        - 이전 단어를 입력(Input)으로 하고 다음 단어를 출력으로 하는 언어모델을 학습할 때, 맞춰야 할 단어나 그 단어 이후의 문맥 정보를 미리 알려줄 수는 없기 때문이다.
        - 물론 ELMo의 경우는 모델의 최종 출력값을 만들 때 마지막 레이어에서 순방향(forward), 역방향(backward) LSTM 레이어의 계산 결과를 모두 사용하기는 합니다.
        - 그러나 pre-train을 할 때 순방향, 역방향 레이어를 각각 독립적으로 학습하기 때문에 ELMo는 진정한 의미의 양방향(bidirectional) 모델이라고 말하긴 어렵습니다.
    - AE 모델의 문제점
        - BERT는 AE의 대표 양방향 모델입니다. 이는 마스크 단어를 예측할 때 앞뒤 문맥을 모두 살피며 성능 또한 좋았습니다. 하지만, AE 모델 역시 단점이 있습니다.
        - 가장 큰 문제는 마스킹 처리한 토큰들을 서로 독립(independent)이라고 가정한다는 점입니다. 이 경우 마스킹 토큰들 사이에 의존 관계(dependency)를 따질 수 없게 됩니다.

그림 3. BERT 모델의 학습

        - 위 사진은 Yang et al.(2019)이 논문에서 사용한 예시 문장('New York is a city')을 가지고 BERT의 학습 과정을 시각화한 것입니다. 영어 말뭉치에서 "New가 나온 다음에 York라는 단어가 나올 확률"과 "New가 나오지 않았을 경우에 York가 등장할 확률"은 분명히 다를 것입니다. 하지만 BERT 모델은 두 단어의 선후 관계나 등장 여부 등 정보를 전혀 따지지 않습니다. 그저 is a city라는 문맥만 보고 New, York 각각의 마스크 토큰을 독립적으로 예측합니다.
        - 또한, BERT는 pre-train할 때 사용하는 마스크 토큰(mask token)은 파인 튜닝(fine-tuning) 과정에서는 사용하지 않습니다. fine-tuning과 다른 pre-train 환경을 구성하면 모델의 일반화(generalization) 성능이 떨어질 수 있다는 단점이 있습니다. 마지막으로 BERT는 긴 문맥을 학습하기 어렵다는 단점도 있습니다.
- ※ AR 모델 문제점 해결 : 퍼뮤테이션 언어모델 - 양방향(bidirectional) 문맥 고려 가능

    - 토큰을 랜덤으로 셔플한 뒤 그 뒤바뀐 순서가 마치 원래 그랬던 것인 양 언어모델을 학습하는 기법이다.

    - 아래 그림은 발 없는 말이 천리 간다를 퍼뮤테이션 언어모델로 학습하는 예시입니다. 모델은 '없는, 이, 말, 발, 간다'를 입력받아 시퀀스의 마지막 단어인 '천리'를 예측합니다.

    그림 4. 퍼뮤테이션 언어모델 (1)

    - Ex) 

    - 이렇게 퍼뮤테이션을 수행하면 특정 토큰을 예측할 때 문장 전체 문맥을 살필 수 있게 됩니다. 즉, 해당 토큰을 제외한 문장의 부분집합 전부를 학습할 수 있다는 뜻입니다.

    - Ex) 발 없는 말이 천리 간다는 문장을 한번 더 퍼뮤테이션해서 이번엔 발, 없는, 천리, 이, 말, 간다가 나왔다고 가정하면, '천리'라는 단어를 예측할 때의 입력 시퀀스는 발, 없는이 됩니다.

    - 위 예제처럼 "천리"라는 단어를 맞추기 위해 모든 입력 시퀀스들을 조합할 수 있고, 거기엔 순방향 언어모델과 역방향 언어모델 모두 퍼뮤테이션 언어모델의 부분집합으로 들어가게 됩니다.

    - 다시 말해 퍼뮤테이션 언어모델은 시퀀스를 순차적으로 학습하는 AR 모델이지만 퍼뮤테이션을 수행한 토큰 시퀀스 집합을 학습하는 과정에서 문장의 양방향 문맥을 모두 고려할 수 있게 된다는 이야기입니다.

- ※ AE 모델 문제점 해결 : 퍼뮤테이션 언어모델 - 단어 간 의존관계를 포착

    - 퍼뮤테이션 언어모델은 AR이기 때문에 BERT 같은 AE 모델의 단점 또한 극복할 수 있다고 설명합니다.

    - 퍼뮤테이션 언어모델은 셔플된 토큰 시퀀스를 순차적으로 읽어가면서 다음 단어를 예측하며, 이전 문맥(New)을 이번 예측(York)에 활용합니다.

    - 따라서 퍼뮤테이션 언어모델은 예측 단어(Masking token) 사이에 독립을 가정하는 BERT와는 달리 단어 간 의존관계를 포착하기에 유리합니다.

    - 뿐만 아니라, pre-train 때 마스크하지 않기 때문에 pre-train & pine-tuning 간 불일치 문제도 해결할 수 있습니다.

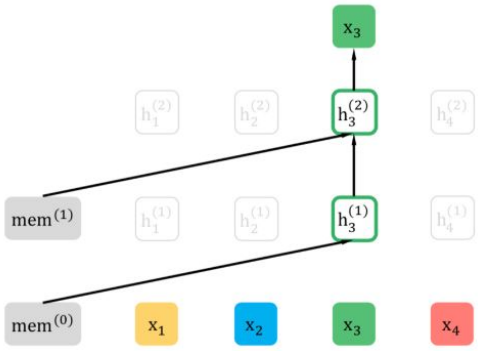
    그림 6. 퍼뮤테이션 언어모델 (3)

    - Ex) 

- ※ 퍼뮤테이션 언어모델(permutaion language model) 학습 예시

그림 7. 퍼뮤테이션 언어모델 학습 (Yang et al., 2019)

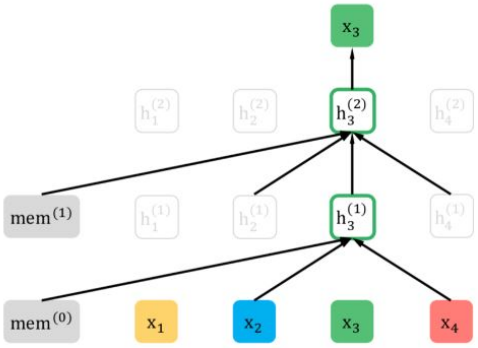
- [3, 2, 4, 1]



- ◎ Example 1 : [3,2,4,1]
- 토큰 네 개짜리 문장을 랜덤으로 뒤섞은 결과가 그림 7처럼 [3,2,4,1]이고 셔플된 시퀀스의 첫번째 단어(3번 토큰)를 맞춰야 하는 상황이라고 가정해 봅시다.
- 그러면 이 때 3번 토큰 정보를 넣어서는 안 됩니다. 3번 토큰을 맞춰야 하는데 모델에 3번 토큰 정보를 주면 문제가 너무 쉬워지기 때문입니다.
- 2번, 4번, 1번 토큰은 맞출 토큰(3번) 이후에 등장한 단어들이므로 이들 또한 입력에서 제외합니다. 결과적으로 이 상황에서 입력값은 이전 세그먼트(segment)의 메모리(memory) 정보뿐입니다. 메모리와 관련해서는 트랜스포머-XL(transformer-XL)에서 설명합니다.
- ◎ Example 2 : [2,4,3,1]

그림 8. 퍼뮤테이션 언어모델 학습 (Yang et al., 2019)

- [2, 4, 3, 1]



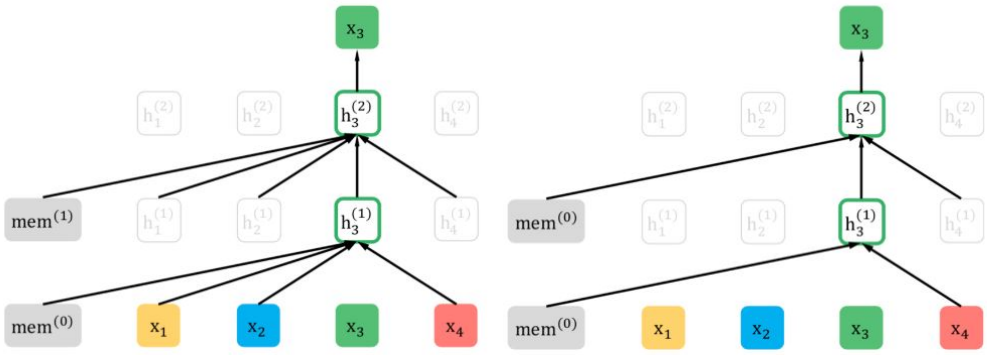
- 같은 문장을 또한번 셔플했더니 [2,4,3,1]이고 이번 스텝 역시 3번 토큰을 예측해야 한다고 가정합시다. 그러면 3번 토큰 이전 문맥(메모리, 2번, 4번 단어)이 입력됩니다. 3번 토큰은 정답이므로 입력에서 제외합니다. 그림 8과 같습니다.
- ◎ Example 3 & 4 : [1,4,2,3], [4,3,1,2]

그림 9. 퍼뮤테이션 언어모델 학습 (Yang et al., 2019)

- [1, 4, 2, 3]

그림 10. 퍼뮤테이션 언어모델 학습 (Yang et al., 2019)

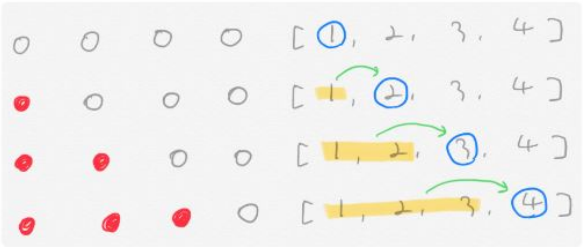
- [4, 3, 1, 2]



- 셔플 시퀀스가 [1,4,2,3]이고 3번 토큰을 맞춰야 한다면 입력벡터는 과거 문맥(메모리, 1번, 4번, 2번 단어)이 됩니다. 마찬가지로 [4,3,1,2]이고 3번을 예측한다면 입력값은 메모리, 4번 단어가 됩니다. 각각 그림 9, 그림 10과 같습니다.

- ※ 퍼뮤테이션 언어모델 학습 과정 1 : 원래 시퀀스의 어텐션 마스크
  - 퍼뮤테이션 언어모델의 실제 구현은 토큰을 뒤섞는 게 아니라 어텐션 마스크(attention mask)로 실현됩니다.
  - XLNet의 근간은 기존 트랜스포머 네트워크(Vaswani et al.,2017)이고, 그 핵심은 쿼리(query), 키(key) 벡터 간 셀프 어텐션(self-attention) 기법이기 때문입니다.
  - 예컨대 토큰 네 개짜리 문장을 단어 등장 순서대로 예측해야 하는 상황을 가정해 봅시다. 이 경우 어텐션 마스크는 그림 11처럼 구축하면 됩니다.

그림 11. 원래 시퀀스의 어텐션 마스크



  - 위 그림에서 좌측 행렬은 셀프 어텐션을 수행할 때 소프트맥스 확률값에 적용하는 마스크 행렬입니다. 여기서 마스크(Mask)란 소프트맥스 확률값을 0으로 무시하게끔 하는 역할을 한다는 뜻입니다. 소프트맥스 확률값이 0이 되면 해당 단어의 정보는 셀프 어텐션에 포함되지 않습니다. 회색 동그라미는 확률값을 0으로 만드는 마스크라는 뜻이며, 붉은색 동그라미는 확률값을 살리는 의미를 지닙니다.
  - 그림을 읽는 방법 : 마스크 행렬의 행은 쿼리 단어, 열은 키 단어에 각각 대응합니다. 그림 11처럼 토큰 순서대로 예측해야 하는 경우 1번 단어를 예측할 때는 자기 자신(1번 단어)을 포함해 어떤 정보도 사용할 수 없습니다. 2번 단어를 맞춰야할 때는 이전 문맥인 1번 단어 정보를 활용합니다. 마찬가지로 3번 단어는 1, 2번 단어, 4번 단어는 1, 2, 3번 단어 정보를 쓰게끔 만듭니다. GPT가 그림 11과 같은 방식으로 학습합니다.

- ※ 퍼뮤테이션 언어모델 학습 과정 2 : 셔플된 시퀀스의 어텐션 마스크
  - 아래 그림 12는 퍼뮤테이션 언어모델이 사용하는 어텐션 마스크의 예시입니다.

그림 12. 셔플된 시퀀스의 어텐션 마스크

    - 셔플된 토큰 시퀀스가 [3,2,4,1]이라고 가정해 봅시다. 그러면 3번 단어를 맞춰야할 때는 어떤 정보도 사용할 수 없습니다. 2번 단어를 예측할 때는 이전 문맥인 3번 단어 정보를 씁니다. 마찬가지로 4번 단어를 맞출 때는 3번, 2번 단어를, 1번 단어를 예측할 때는 3번, 2번, 4번 단어 정보를 입력합니다.

- ※ 퍼뮤테이션 언어모델 단점 : 투-스트림 셀프 어텐션으로 해결
    - 예컨대 단어가 네 개인 문장을 랜덤 셔플한 결과가 다음와 같고 이번 스텝에서 셔플 시퀀스의 세번째를 예측해야 한다고 해봅시다. ==> `[3, 2, "4", 1] , [3, 2, "1", 4]`
    - 이 경우 모델은 동일한 입력(3번, 2번 단어)을 받아 다른 출력을 내야 하는 모순에 직면합니다. Yang et al.(2019)는 이같은 문제를 해결하기 위해 `투-스트림 어텐션(two-stream self attention) 기법`을 제안했습니다.

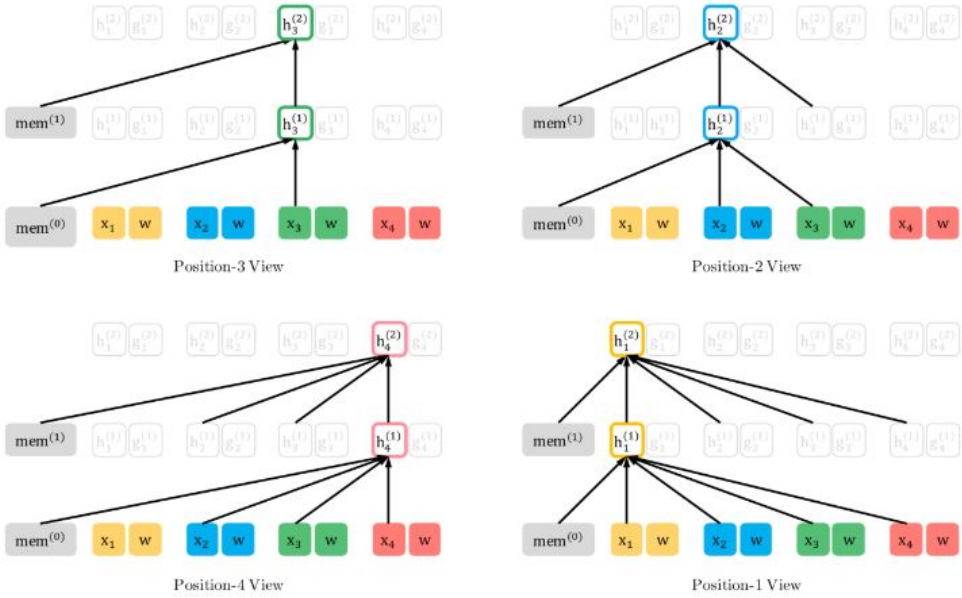
투-스트림 셀프어텐션(Two-Stream Self Attention)

- ※ 정의
    - 투-스트림 셀프어텐션(Two-Stream Self Attention)은 `쿼리 스트림(query stream)`과 `컨텐트 스트림(content stream)` 두 가지를 혼합한 `셀프 어텐션 기법`입니다.
- ※ 컨텐트 스트림(content stream)
    - 컨텐트 스트림은 기존 트랜스포머 네트워크와 거의 유사하다.
    - **수식 1. 컨텐트 스트림 h (1)**

수식 1. 컨텐트 스트림 (1)

$$\mathbf{h}_{z_t}^{(m)} \leftarrow \text{Attention}\left(\mathbf{Q}=\mathbf{h}_{z_t}^{(m-1)}, \mathbf{KV}=\mathbf{h}_{z_{\le t}}^{(m-1)}; \theta\right)$$

그림 13. 컨텐트 스트림 (Yang et al., 2019)



- 수식 1 세부 내용
  - h : 컨텐트 스크림 벡터
  - z : 원래 문장 순서를 랜덤 셔플한 인덱스 리스트
    - Ex) 단어가 네 개인 문장이라면 이를 랜덤 셔플한 샘플 하나(ex. [3,2,4,1])가 바로 z입니다.
  - z_t : z의 t번째 요소
  - 수식 1 계산 결과 : m번째 트랜스포머 블록의 z_t에 해당하는 컨텐트 스트림 벡터입니다. 즉, m번째 블록의 컨텐트 스트림 벡터는 $m-1$번째 컨텐트 스트림 벡터에 기존 트랜스포머 블록과 동일한 계산을 수행한 결과입니다.
- 그림 13 세부 내용
  - x : 토큰 임베딩 => 단어가 네 개인 문장 "나 어제 학교 갔어"가 있다면 x_1은 나, x_2는 어제, x_3은 학교, x_4는 갔어에 해당하는 토큰 임베딩입니다.
  - 가정 : 위 문장을 셔플한 인덱스 리스트 z가 [3,2,4,1]이고 이번에 예측할 단어가 z의 첫번째($z_1=3$)라고 가정해 봅시다.
  - z의 첫번째 컨텐트 스트림($g_{z_1}=g_3$)을 만들 때 자기 자신의 토큰 임베딩 정보($x_{z_1}= x3 = 학교$) 역시 셀프 어텐션 계산에 포함되는 걸 확인할 수 있습니다.
  - 예측할 단어가 z의 첫번째($z_1=3$)인 상황이라면 이전 문맥(메모리)과 자기 자신의 토큰 임베딩 정보($X_3=학교$)가 게산에 포함됩니다.(그림 13 상단 좌측). z의 두번째($z_2=2$)인 상황이라면 이전 문맥(메모리, 학교)과 자기 자신($X_2=어제$)을 넣습니다.(그림 13 상단 우측). 이후 마찬가지입니다.
  - 참고 : 쿼리 스트림을 계산할 때는 x_3을 빼고 계산합니다. 자기 자신의 토큰 정보를 빼는 방식이 컨텐트 스트림에도 동일하게 적용되고 이 방식이 트랜스포머 블록별로 누적되면

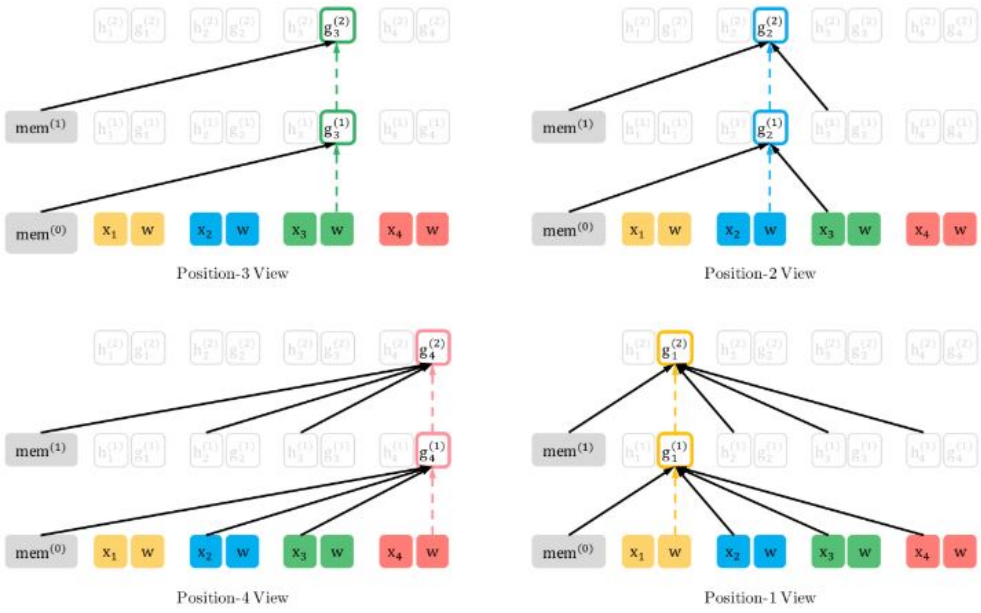
x_3 이후 등장하는 토큰들을 예측할 때 x_3 정보를 참고하기 어려워져 불리합니다. 컨텐트 스트림을 만들 때는 x_3을 포함하도록 설계했습니다.

- **수식 2. 컨텐트 스트림 h (2)**
- $$h_\theta\left(\mathbf{x}_{z_{<t}}\right)$$
- 수식 2 세부 내용
  - 컨텐트 스트림 h를 수식 1과 그림 13을 간소화해 표기하면 수식 2와 같습니다. z의 t번째 요소에 해당하는 컨텐트 스트림을 만들 때는 이전 문맥과 자기 자신에 대응하는 토큰 정보($x4$)를 활용한다는 의미입니다. 첫 번째 트랜스포머 블록에 입력되는 컨텐트 스트림 벡터의 초기값은 해당 단어에 해당하는 임베딩 벡터(x)입니다.
- **수식 3. 쿼리 스트림 g (1)**

수식 3. 쿼리 스트림 (1)

$$\mathbf{g}_{z_t}^{(m)} \leftarrow \text{Attention}\left(\mathbf{Q}=\mathbf{g}_{z_t}^{(m-1)}, \mathbf{KV}=\mathbf{h}_{z_{<t}}^{(m-1)}; \theta\right)$$

그림 14. 쿼리 스트림 (Yang et al., 2019)



- 수식 3 세부 내용
  - 쿼리 스트림은 토큰(token)과 위치(position) 정보를 활용한 셀프어텐션 기법입니다. 쿼리 스트림을 만들 때는 이전 토큰 정보뿐만 아니라 이번에 맞춰야할 타겟 단어의 위치 정보를 활용합니다.
  - ★ 이번 레이어(m)에서 특정 시점(z_t)에 해당하는 단어의 쿼리 스트림 벡터($g_{z_t}(m)$)를 계산할 때는 이전 레이어($m-1$)의 t번째 "미만"의 컨텐트 스트림($h_{z<t}(m-1$))을 키(K)와 값(V)으로 씁니다. 결과적으로 $g_{z_t}^{(m)}$에는 지금 맞춰야할 단어의 임베딩 정보(x_{z_t})가 들어가지 않습니다.
  - 위 예시 활용 : 원래 문장의 첫번째 단어의 임베딩인 x_1은 "나", 두번째 단어 임베딩(x_2)은 "어제", 세번째 단어 임베딩(x_3)은 "학교", 네번째 단어 임베딩(x_4)은 "갔

어"에 대응하는 벡터입니다. 이 문장을 셔플한 인덱스 리스트 z가 [3,2,4,1]이고 이번 예측할 단어가 z의 세번째($z_3 = 4$)라고 가정해봅시다. -> 그림 14 좌측 하단

- 그림 14 좌측 하단 세부 설명
  - m번째 레이어의 z_t에 해당하는 단어의 쿼리 스트림을 구하려면 쿼리(Q), 키(K), 값(V)이 필요합니다. 1번 째 레이어(그림 14의 하단 좌측 두번째 층)의 z의 세번째($z_3=4$)에 해당하는 단어의 쿼리 스트림($g_{z_3^{(1)}} = g_{z_4^{(1)}}$)을 계산할 때 키($K$)와 값($V$)은 $t(=3)$번째 미만의 컨텐트 스트림이 됩니다. 키(K), 값(V)에는 자기 자신의 임베딩($x_{z_3} = x_4 = 갔어$)을 제외한 이전 문맥(메모리, $x_3 = 학교, x_2 = 어제$)과 관련한 컨텐트 스트림만 반영됩니다.(검은 실선으로 표시된 것)
  - 쿼리(Q)를 만들 땐 z의 세 번째에 해당하는 단어 임베딩 정보($x_{z_3} = x_4 = 학교$)는 빼고 위치 정보($z_3=4$)만 포함됩니다(붉은색 점선으로 표시).
  - 결과적으로 이번 레이어 쿼리 스트림 $g_{z_3}^{(1)} = g_4^{(1)}$을 게산할 때 쿼리($Q$), 키($K$), 값($V$) 모두, 지금 맞춰야 할 단어 임베딩 정보($x_{z_3}=x_4=갔어$)가 빠지게 됩니다.
  - 다음 레이어의 쿼리 스트림 $g_{z_3}^{(2)} = g_4^{(2)}$ 역시, 쿼리(Q)는 이전 레이어의 쿼리 스트림($g_{z_3}^{(1)} = g_4^{(1)}$)이기 때문에, 레이어가 계속 거듭되더라도 지금 맞춰야 할 단어의 임베딩 정보($x_{z_3}=x_4=갔어$)는 모델이 볼 수 없습니다
- 그림 14 우측 하단 세부 설명
  - 이번엔 그림 14의 우측 하단을 봅시다. 이번에 계산할 단어가 z의 네번째($z_4=1$)라면 이전 문맥(메모리, $x3=학교, x2=어제, x4=갔어$)만 입력하고, 자기 자신의 토큰 정보($x_1=나$)는 쿼리 스트림에 넣지 않습니다. 하지만 $z_4=1$이라는 위치 정보는 쿼리 스트림 계산에 포함됩니다. 이는 모델에 다음 질문을 던지는 것과 같다고 생각합니다.
    - 여태까지 학교, 어제, 갔어라는 단어를 봤는데 말이야. 이번에 맞춰야 할 단어는 원래 문장에서 첫번째에 있었어. 이 단어는 뭘까?
- Yang et al.(2019)이 쿼리 스트림에 자기 자신의 토큰 정보를 빼고, 이전 문맥 단어 정보와 자신의 위치 정보를 넣은 이유는 앞서 언급한 퍼뮤테이션 언어모델의 한계 때문입니다. 퍼뮤테이션 탓에 모델은 동일한 입력을 받아 다른 출력을 내야 하는 모순에 직면한다는 점을 이미 언급한 바 있습니다. 쿼리 스트림을 위와 같이 설계하게 되면 아래 두 퍼뮤테이션 시퀀스는 다음과 같은 의미를 지니게 돼 모순을 회피할 수 있습니다.
  - `[3, 2, 4, 1]` : 여태까지 학교, 어제라는 단어를 봤는데 말이야. 이번에 맞춰야 할 단어는 원래 문장에서 네번째에 있었어. 이 단어는 뭘까?
  - `[3, 2, 1, 4]` : 여태까지 학교, 어제라는 단어를 봤는데 말이야. 이번에 맞춰야 할 단어는 원래 문장에서 첫번째에 있었어. 이 단어는 뭘까?
- **수식 4. 쿼리 스트림 g (2)**

$$g_\theta\left(\mathbf{x}_{z<t}, z_t\right)$$

  - 쿼리 스트림 g를 수식 3과 그림 14을 간소화해 표기하면 수식 4와 같습니다. z의 t번째 요소에 해당하는 쿼리 스트림을 만들 때는 현 시점 "미만"의 이전 문맥에 대응하는 단어 정보($x_{<t}$)와 자기 자신의 위치 정보(z_t)를 활용한다는 의미입니다. 첫번째 트랜

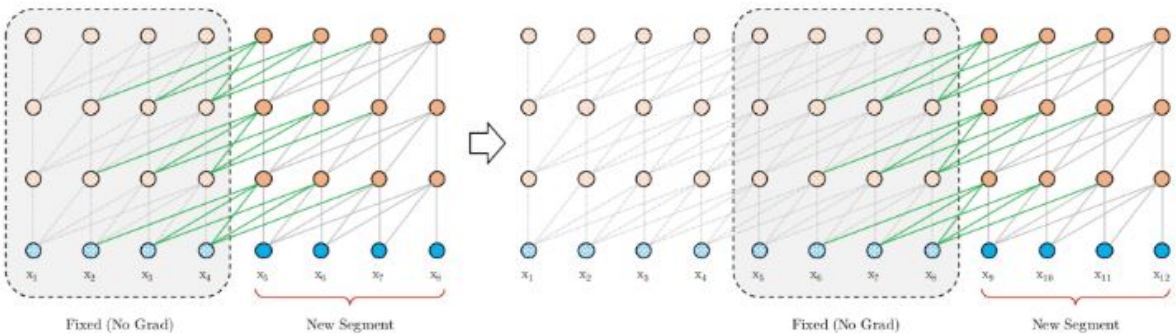
스포머 블록의 쿼리 스트림을 계산할 때 사용되는 쿼리(Q)는 랜덤 초기화한 벡터이며 다른 모델 파라메터와 같이 학습합니다.

- 프리트레인(pre-train) 과정에서 트랜스포머-XL 레이어를 m개 사용했을 경우 z_t번째 단어에 대응하는 XLNet 임베딩의 최종 출력 벡터(output)는 마지막 m번째 트랜스포머-XL 레이어의 쿼리 스트림 $g_{z_t}(m)$입니다. XLNet 모델은 이를 활용해 다음 단어 예측을 수행합니다.

트랜스포머-XL (Transformer-XL)

- ※ 정의 및 간단 설명
  - 트랜스포머-XL(Dai et al., 2019)는 XLNet 이전에 발표된 모델입니다. Yang et al. (2019)는 트랜스포머-XL의 `세그먼트 리커런스(segment recurrence)`와 `상대 위치 임베딩(relative position embedding)`기법을 그대로 차용했습니다.
- `세그먼트 리커런스(segment recurrence)`
  - Dai et al. (2019)는 기존 트랜스포머 네트워크(Vaswani et al., 2017)의 단점으로 고정된 길이의 문맥 정보만 활용할 수 있다는 점을 꼽았습니다. 이에 좀 더 긴 컨텍스트(context)를 보기 위해 세그먼트 리커런스라는 기법을 제안했습니다.
  - 기존 트랜스포머 네트워크는 문서가 `연구자가 정한 최대 시퀀스 길이`를 넘을 경우 그 길이 이후에 나타난 토큰들은 학습에서 제외합니다. 하지만 Dai et al. (2019)의 방식은 다릅니다.
  - `우선 문서를 작은 세그먼트 단위로 자른다.` 첫번째 세그먼트를 기존 트랜스포머 네트워크처럼 학습합니다. 첫번째 세그먼트를 충분히 학습했다면 이를 저장(cache)해 두고, 두번째 세그먼트를 학습합니다. 두번째 세그먼트를 계산할 때는 첫번째 세그먼트 정보를 활용합니다.
  - Dai et al. (2019)는 현재 세그먼트를 학습할 때 `고려 대상에 포함하는 직전 세그먼트 계산 결과`를 `메모리(memory)`라고 이름 붙였습니다. 투-스트림 셀프 어텐션의 예시 그림에서 입력 토큰이 현재 세그먼트, memory라고 표현된 부분이 직전 세그먼트에 해당합니다. 단 현재 세그먼트를 계산할 때는 메모리를 학습하지 않습니다. `다시 말해 학습 손실(train loss)을 줄이기 위한 그래디언트를 메모리 쪽에 반영하지 않는다는 이야기입니다.` 그림 15에서 확인할 수 있는 것처럼 이같은 세그먼트의 학습은 반복적(recurrence)으로 수행합니다.

그림 15. 세그먼트 리커런스의 학습 (Dai et al., 2019)



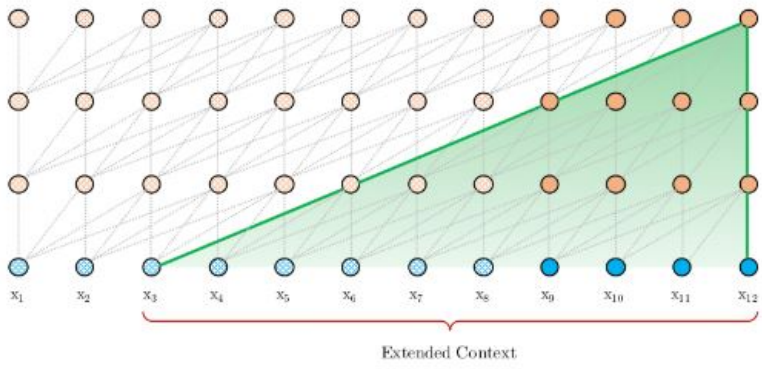
  - 코드 1은 현재 세그먼트 계산 결과를 메모리에 저장하는 함수를 텐서플로로 구현한 것입니다. `이전 메모리 정보(prev_mem)와 현재 계산 결과(curr_out)를 합친(concat) 후 이들에 대해서는 그래디언트 업데이트를 하지 않도록(tf.stop_gradient) 처리합니다.` XLNet의

modeling.py에 정의되어 있습니다. 세그먼트 리커런스 학습이 완료되면 모델이 고려할 수 있는 문맥 범위가 넓어집니다. 그림 16은 이를 시각화한 것입니다.

- 코드 1. 현재 세그먼트를 메모리에 저장

```
def _cache_mem(curr_out, prev_mem, mem_len, reuse_len=None):
  """cache hidden states into memory."""
  ...
  new_mem = tf.concat([prev_mem, curr_out], 0)[-mem_len:]
  return tf.stop_gradient(new_mem)
```

그림 16. 세그먼트 리커런스의 예측 (Dai et al., 2019)



- Dai et al. (2019)는 단어 쌍 사이의 거리 정보인 상대 위치(relative position)를 활용했습니다. 원래 문장을 작은 세그먼트들로 쪼개고, 세그먼트별로 트랜스포머 네트워크를 학습하면 각 단어가 문장 내에서 차지하는 절대 위치(absolute position) 정보가 무의미해지기 때문입니다.

- ※ "상대 위치"와 "절대 위치" 개념 설명

  - <발, 없는, 말, 이, 천리, 간다> 문장이라면 천리의 절대 위치는 5입니다. 문장에서 다섯 번째로 등장한 단어라는 의미입니다. 천리를 기준으로 한 발의 상대 위치는 4입니다. 천리에서 발까지 이동하려면 왼쪽으로 네 칸을 이동해야 한다는 뜻입니다.

  - Dai et al. (2019)가 세그먼트에 상대 위치를 적용한 걸 도식화하면 그림 17과 같습니다.

  - 예컨대 토큰 수 기준 세그먼트 길이가 3이고 학습 대상 문장이 <발, 없는, 말, 이, 천리, 간다>이며 이번이 두번째 세그먼트를 학습할 차례라고 가정해 봅시다. 그러면 모델은 현재 세그먼트 학습의 첫 단계에서 <발, 없는, 말>이라는 메모리 정보를 입력받아 이를 예측해야 합니다. 이는 쿼리(query)가 되고, 발, 없는, 말은 각각 키(key)가 됩니다.

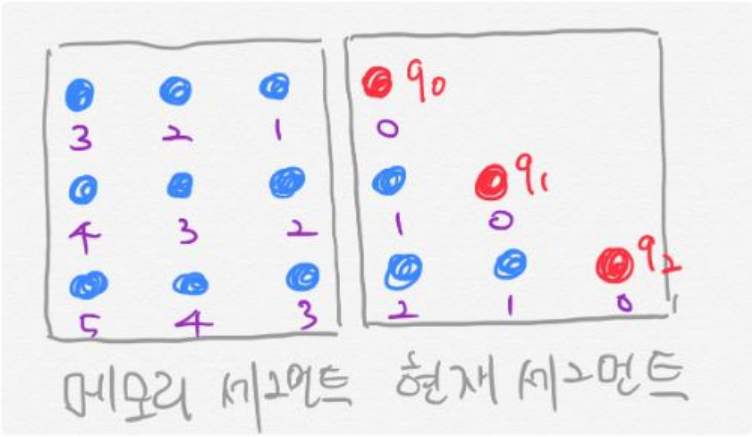
  - 이때 "이"를 기준으로 한 "발"의 상대 위치는 3입니다. "이"에서 "발"까지 이동하려면 왼쪽으로 세 칸을 이동해야 한다는 의미입니다. 같은 방식으로 계산하면 "이-없는", "이-말"은 상대 위치가 각각 2, 1이 된다. 자기 자신의 상대 위치(이-이)는 0입니다.

  - 현재 세그먼트 학습의 두번째 스텝에서는 메모리 정보(발, 없는, 말)와 직전 단어("이")를 바탕으로 천리를 예측해야 합니다. 천리가 새로운 쿼리가 되었기 때문에 이전 문맥 단어들과의 상대 거리를 다시 계산해야 합니다. 천리-발, 천리-없는, 천리-말, 천리-이, 천

리-천리의 상대 위치는 각각 4, 3, 2, 1, 0이 됩니다. 세그먼트 학습의 세번째 단계(쿼리가 간다인 상황)도 같은 방식으로 수행하면 됩니다.

- Dai et al. (2019)가 설계한 상대 위치는 음수값이 존재하지 않습니다. 트랜스포머-XL은 단어를 순차적으로 학습하는 AR 모델이기 때문입니다.
- ★ 다시 말해 현재 쿼리 단어를 예측해야 하는 상황이라면, 쿼리 단어 이후의 단어에 어텐션이 걸리지 않도록 어텐션 마스크를 만듭니다. 예컨대 "천리"를 기준으로 한 "간다"의 상대 거리는 -1이겠지만, 간다는 천리 이후에 등장하는 단어이기 때문에 어텐션 계산에서 제외하게 됩니다. 따라서 천리-간다의 상대 위치 역시 구할 필요가 없습니다.

그림 17. 상대 위치 시각화



- **수식 5. Scaled Dot-Product Attention $Attention(Q,K,V)$**

수식 5. Scaled Dot-Product Attention (1)

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

- 수식 5는 기존 트랜스포머 네트워크의 셀프 어텐션(Vaswani et al., 2017)을 나타낸 것입니다.

- **수식 6. 절대 위치에 기반한 셀프 어텐션 $A_{i,j}^{abs}$**

수식 6. 절대 위치에 기반한 셀프 어텐션

$$\mathbf{A}_{i,j}^{\text{abs}} = \underbrace{\mathbf{E}_{x_i}^\top \mathbf{W}_q^\top \mathbf{W}_k \mathbf{E}_{x_j}}_{(a)} + \underbrace{\mathbf{E}_{x_i}^\top \mathbf{W}_q^\top \mathbf{W}_k \mathbf{U}_j}_{(b)} + \underbrace{\mathbf{U}_i^\top \mathbf{W}_q^\top \mathbf{W}_k \mathbf{E}_{x_j}}_{(c)} + \underbrace{\mathbf{U}_i^\top \mathbf{W}_q^\top \mathbf{W}_k \mathbf{U}_j}_{(d)}.$$

- 수식 6은 Dai et al. (2019)가 수식 5 가운데 QK^T만 떼어내어 다시 표현한 것입니다.

- 수식 6 세부 설명
  - A : 수식 5의 Q와 K를 내적한 결과인 정방행렬(Square matrix)입니다.

- i 와 j : 각각 쿼리와 키의 인덱스(index)를 뜻합니다.
- A_{ij} 값 : i번째 쿼리 단어와 j번째 키 단어가 태스크 수행에 얼마나 관련을 맺고 있는지를 나타내는 점수(score)가 됩니다.
- W_qE_{xi} : 수식 5의 행렬 Q의 i번째 행, 즉 i번째 쿼리 벡터를 가리킵니다.
- W_kE_{xj} : 수식 5의 행렬 K의 j번째 행, 즉 j번째 키 벡터를 의미합니다.
- 기존 트랜스포머 네트워크는 단어의 절대 위치(absolute position) 정보를 사용합니다.
- U_i와 U_j : 각각 문장에서 i번째로 등장한 단어와 j번째 단어의 절대 위치 정보가 담긴 임베딩을 나타냅니다.
- 따라서 W_qU_i는 쿼리 단어의 위치 정보, W_kU_j엔 키 단어의 위치 정보가 녹아 있습니다.

- **수식 7. 상대 위치에 기반한 셀프 어텐션 $A_{i,j}^{rel}$**

수식 7. 상대 위치에 기반한 셀프 어텐션

$$\mathbf{A}_{i,j}^{\text{rel}} = \underbrace{\mathbf{E}_{x_i}^\top \mathbf{W}_q^\top \mathbf{W}_{k,E} \mathbf{E}_{x_j}}_{(a)} + \underbrace{\mathbf{E}_{x_i}^\top \mathbf{W}_q^\top \mathbf{W}_{k,R} \mathbf{R}_{i-j}}_{(b)} + \underbrace{u^\top \mathbf{W}_{k,E} \mathbf{E}_{x_j}}_{(c)} + \underbrace{v^\top \mathbf{W}_{k,R} \mathbf{R}_{i-j}}_{(d)}.$$

- Dai et al. (2019)는 상대 위치에 기반한 셀프 어텐션 기법을 제안했으며 수식 7로 표현 했습니다. 수식 6에서 바뀐 부분은 칼라 표시가 되어 있습니다.

- 수식 7 세부 설명

- 우선 절대 위치 임베딩 행렬 U를 상대 위치 임베딩 행렬 R로 대체한 점이 눈에 띕니다.
- R은 최대 시퀀스 길이 $×d$차원 크기를 갖는 행렬입니다.
- 쿼리 단어를 기준으로 한 키 단어의 상대 위치가 k일 때 R의 k번째 행 벡터를 참조(lookup)해서 씁니다.
- Dai et al. (2019)는 행렬 R을 사인(sin), 코사인(cosine) 함수 등을 활용해, 학습하지 않는 파라메터(non-trainable parameter)로 두었습니다.
- 그림 17에서 확인했던 것처럼 쿼리 단어를 기준으로 한 쿼리 단어의 상대 위치는 늘 0이기 때문에 쿼리 단어의 위치 정보를 사용하는 게 큰 의미가 없습니다. 이에 Dai et al. (2019)는 W_qU_i를 벡터(u,v) 형태로 단순화했습니다.
- 수식 7의 (a)와 (c)는 기존 트랜스포머 네트워크 계산방식과 크게 달리진 것이 없습니다. 문제는 R이 포함된 (b)와 (d)입니다.
- Dai et al. (2019)는 상대 위치가 포함된 셀프 어텐션 행렬 계산을 효율적으로 하는 테크닉을 제시했습니다. 우선 (b)를 봅시다. 그림 17처럼 최대 시퀀스 길이가 6이라 가정하고, (b)에서 $W_{k,r}R_{i-j}$를 떼어 다시 표현하면 수식 8과 같습니다.

- **수식 8. 상대 위치 셀프 어텐션 계산 예시 (1)**

수식 8. 상대 위치 셀프 어텐션 계산 예시 (1)

$$\mathbf{Q} = \begin{bmatrix} [\mathbf{W}_{k,R}\mathbf{R}_5]^T \\ [\mathbf{W}_{k,R}\mathbf{R}_4]^T \\ [\mathbf{W}_{k,R}\mathbf{R}_3]^T \\ [\mathbf{W}_{k,R}\mathbf{R}_2]^T \\ [\mathbf{W}_{k,R}\mathbf{R}_1]^T \\ [\mathbf{W}_{k,R}\mathbf{R}_0]^T \end{bmatrix} = \begin{bmatrix} \mathbf{Q}_0{}^T \\ \mathbf{Q}_1{}^T \\ \mathbf{Q}_2{}^T \\ \mathbf{Q}_3{}^T \\ \mathbf{Q}_4{}^T \\ \mathbf{Q}_5{}^T \end{bmatrix}$$

그림 18. 상대 위치 셀프 어텐션 계산 예시 (2)

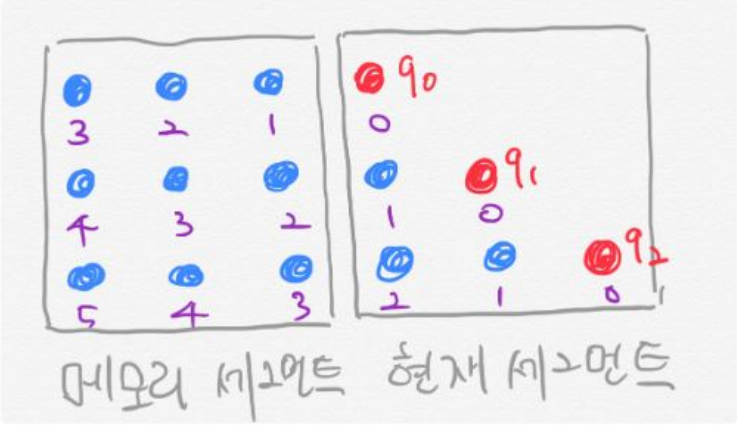
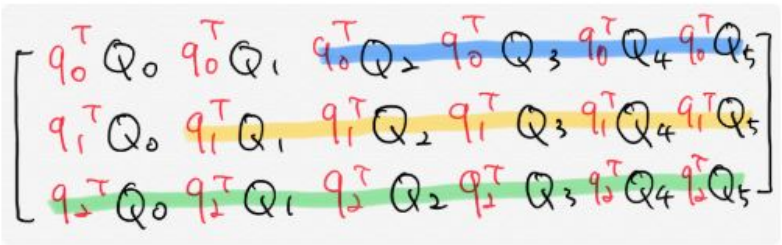


그림 19. 상대 위치 셀프 어텐션 계산 예시 (3)



- $W_qE_{x_i}$를 q_i라 둡시다. 그림 18의 위쪽 그림과 같은 상황에서 수식 8을 참고해 (b)를 계산한 결과는 그림 18의 아래쪽 그림과 같습니다.
- 이와 별개로 각 쿼리 벡터들을 모아 만든 행렬을 q라고 둡시다. 그러면 q와 Q를 내적한 qQ를 시각화한 것은 그림 19와 같습니다. 그림 18의 아래쪽 그림과 비교해서 봅시다. qQ(그림 19)의 각 행별로 한 칸씩 왼쪽으로 옮기면 우리가 원하는 결과(그림 18 아래쪽)를 얻을 수 있습니다.
- 요컨대 쿼리 벡터를 모아 만든 행렬 q와 상대 위치 정보가 담긴 Q를 내적하고 살짝 후처리만 해주는 것으로도 상대 위치 셀프 어텐션을 계산할 수 있습니다. 행렬 내적 연산은 텐서플로 등 다양한 라이브러리에서 최적화되어 있기 때문에 상대 위치 계산에 큰 비용이 들지 않습니다. 수식 7의 (d)도 이와 유사한 방식으로 효율적으로 계산할 수 있습니다.

모델 구현
