Attention is All you need

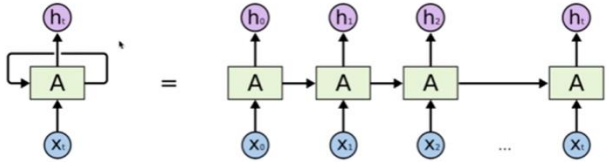
NeurIPS 2017

2025.05.02

Review

▪ 순환형 모델 (Recursion Model)

- 문장과 같은 시계열 데이터 (Sequence Data)를 다루기 위한 모델
- 대표적으로 RNN/LSTM/GRU 등의 모델이 존재



RNN 모델의 입력?

컴퓨터는 자연어를 이해할 수 없으므로 단어를 vector로 변환하는 과정이 필요하다 (= Embedding)
Ex) word2vec, GloVe, BERT 등…

RNN 모델에서는 각 입력을 어떻게 처리할까?

시계열 데이터의 각 데이터 $(X_1, X_2 \ldots)$를 순차적으로 입력, 이때 각 데이터는 Embedding 과정을 거친 Vector의 형태

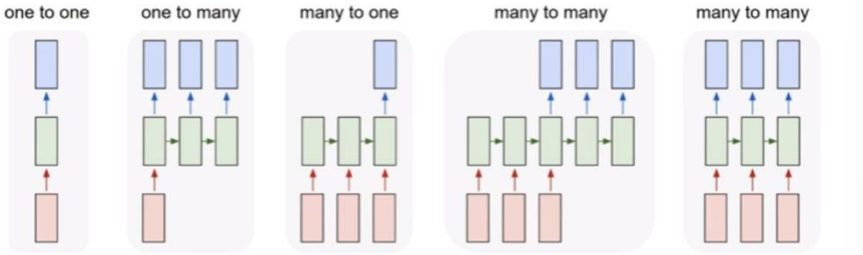
- 매 Time-step에서의 출력 값은 (=hidden state vector) 현재까지 입력의 정보를 모두 반영한다.
- 각 time-step에서의 hidden state vector를 다음 time-step으로 전달한다.
- RNN 계층은 이전 Time-step의 hidden state vector (= 이전 입력의 정보)와 현재의 입력값 (= 새로운 정보)를 모두 반영하여 새로운 출력 값 (=현재 Time-step의 hidden state vector)를 생성한다

RNN 모델에서는 각 출력 값을 어떻게 활용할까?

➢ 시계열 데이터 전체의 맥락만이 필요하다? 마지막 time-step의 hidden state vector만을 활용
➢ 매 입력까지의 맥락을 바탕으로 각각의 출력 값이 필요하다? 매 time-step의 hidden state vector를 활용

Review

- **순환형 모델의 활용**



- One-to-one
- One-to-many (ex: Image Captioning)
- Many-to-one (ex: 스팸 메일 분류)
- Many-to-many (ex: 기계 번역)

Introduction

▪ Transformer

sequence transduction task에서 활용, 높은 성능과 연산 효율성을 보인다

기존의 모델?

RNN 기반 encoder-decoder 구조의 model을 활용
(seq2seq + attention 기반 모델)

Prob1 RNN 모델에 의존

- 순환형 모델의 long-term dependency (장기 의존성 문제) 발생
- 순환형 모델은 각 토큰을 순차적으로 입력하기에, 완전한 병렬화가 불가능하다

Prob2 Attention vector의 연산 과정

- Decoder의 매 타임 스텝에서 이루어지는 Attention vector의 연산 과정이 매우 복잡하다

Sol?

- RNN을 사용하지 않고 텍스트를 처리한다
- 전처리를 거친 임베딩 벡터와 가중치의 곱으로 attention vector를 계산한다

seq2seq model Architecture

- **seq2seq 모델의 Encoder-Decoder 구조**

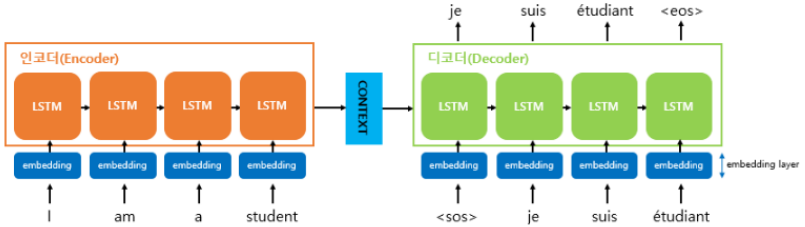
Encoder, Decoder에서 각각 하나의 순환형 모델 활용

1. Encoder에서 번역 대상 문장의 각 단어를 순차적으로 순환형 모델의 입력으로 제공

2. 마지막 Time step에서의 hidden state vector를 Decoder의 첫 Time step에서의 hidden state vector로 활용

3-1. 예측 과정
Decoder의 첫 입력으로 <SOS> 토큰을 제공하고, 이후 각 Time step에서의 출력 값을 다음 Time step의 입력으로 제공한다. (Auto-regressive)

3-2. 학습 과정
Decoder의 입력으로 이전 Time step의 출력 대신, 정답 레이블을 제공한다. (Teacher-Forcing)



Problem

- Long-term dependency problem (장기 의존성 문제)
- 전체 문장의 맥락과 현재까지의 입력을 한 context vector만으로 반영하여야 한다

-> Decoder의 토큰과 Encoder의 모든 context vector의 유사도를 활용하여 중요도를 계산하자 (= Attention)

seq2seq model + attention Architecture

- **seq2seq + attention**

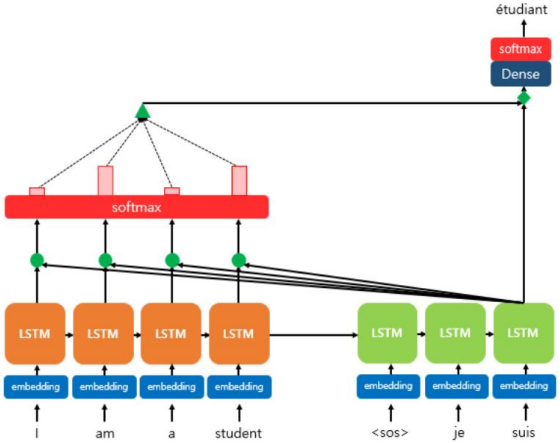
Decoder의 각 Time step에서 attention 활용

1. Decoder의 각 hidden state vector와 Encoder의 모든 hidden state vector의 유사도 계산 (= attention score)

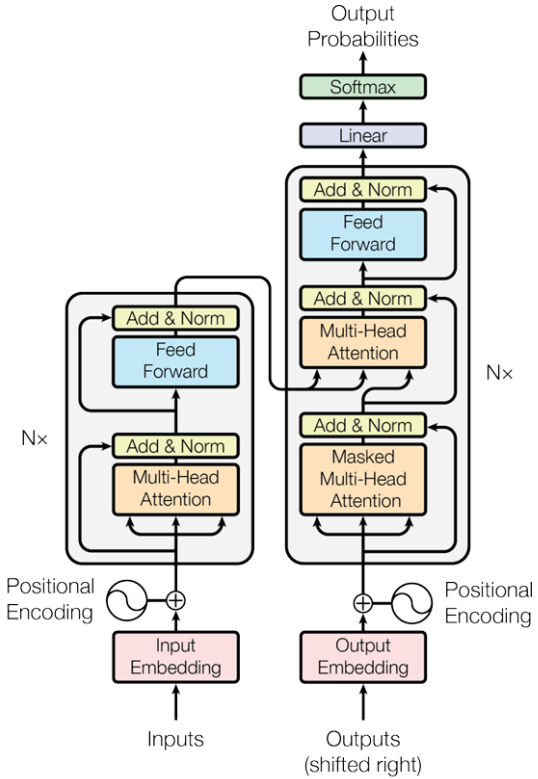
2. Encoder의 매 Time step에서의 유사도를 softmax 연산 (= attention weight)

3. Attention weigh와 Encoder의 각 Time step에서의 hidden state vector의 가중합 계산 (= context vector)

4. context vector를 Decoder의 hidden state vector와 concat 후 linear transform



Model Architecture



- **Overall**

1 Encoder-Decoder 구조로 구성

2 RNN 모델을 포함하지 않는다

3 Encoder와 Decoder 모두 Multi-Head Attention 블록을 포함

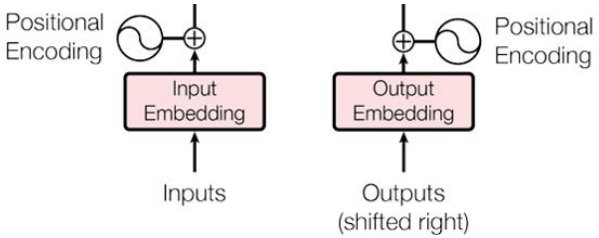
Encoder와 Decoder에서 각각 수행하는 역할은?

RNN 모델의 활용 없이 어떻게 시계열 데이터를 처리할까?

Multi-Head Attention과 기존의 Attention vector 계산 과정의 차이는?

Model Architecture

- **Preprocessing**



1 Word Embedding (word > vector)

텍스트 데이터를 벡터로 변환하는 과정
기존의 word embedding (ex: word2vec, BERT 등) 활용

2 Positional Encoding

- Transformer는 RNN 구조를 활용하지 않는다
- 문장은 각 토큰의 위치 정보에 따라 의미가 달라질 수 있는 시계열 데이터이기에, 각 토큰의 위치 정보를 반영하여야 한다
- 각 토큰의 위치 정보를 바탕으로 embedding vector와 동일한 차원의 position vector를 만든 후 word embedding vector와 더하여 각 토큰에 위치 정보를 추가한다.

- position encoding을 거친 embedding vector 자체에서 위치 정보를 반영하고 있기에 순차적인 입력 없이 각 토큰을 한 행렬로 처리할 수 있다, 즉 병렬화가 가능해진다

$$PE_{(pos,2i)} = sin(pos/10000^{2i/d_{\text{model}}})$$
$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d_{\text{model}}})$$

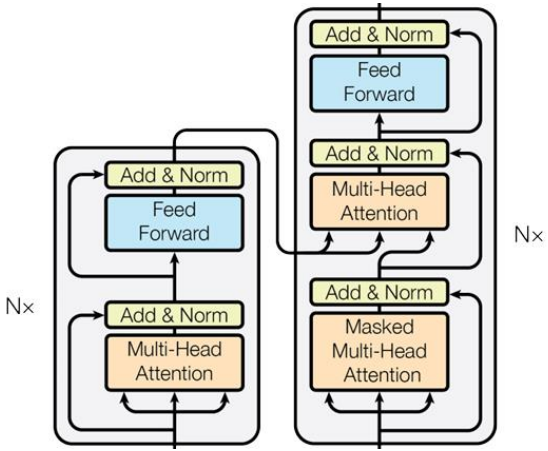
d_model: embedding vector의 차원

Pos: 문장 내에서 토큰의 위치

i: position vector의 d_model 차원 중 몇번째 요소인지

Model Architecture

- Multi-Head Attention



- **Attention (in seq2seq)**

Query: decoder의 hidden state vector

Key, Value: encoder의 모든 타임 스텝에서의 hidden state vector

- **Multi-Head Attention**

1) Self Attention

1-1 Query/Key/Value 모두를 embedding vector와 가중치 (W_Q, W_K, Q_V)의 곱으로 구한다

1-2 Query와 Key의 내적을 통해 Attention score를 계산한다

1-3 Attention Score를 d_k로 정규화한 값을 softmax 연산하여 Attention weight를 계산한다

1-4 Attention weight와 Value의 weighted sum으로 Attention Value를 구한다

2) Multi-Head Attention

Self-Attention을 여러 번 수행한 결과를 concat한 후 linear transform을 통해 Multi-Head Attention value를 구한다

Model Architecture

- Attention vs Self-Attention

Attention vector를 계산하는 과정은 동일

1 Query와 모든 토큰의 key의 유사도 계산

2 각 key별로 유사도를 softmax하여 Attention weight 계산

3 Attention weigh를 value와 weighted sum 연산하여 Attention vector 계산

Attention

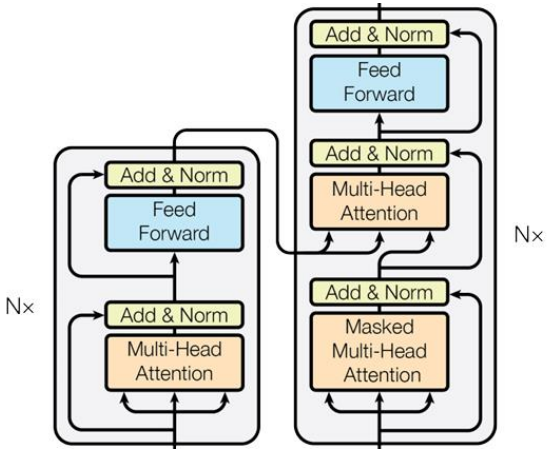
- Decoder의 모든 토큰에 사용되는 key, value는 Encoder의 hidden state vector로 동일
- Query는 Decoder의 각 타임 스텝에서의 hidden state vector이기에 순차적으로 각 토큰의 attention value를 구해야 한다

Self-Attention

- 토큰과 각 가중치의 곱으로 query, key, value를 모두 구할 수 있다
- 한 시퀀스 데이터를 행렬로 처리하여 한번에 한 시퀀스 데이터의 query, key, value를 구한 후 attention value 연산이 가능하다

Model Architecture

- Encoder 구조



1) Embedding vector를 대상으로 Multi-Head Attention

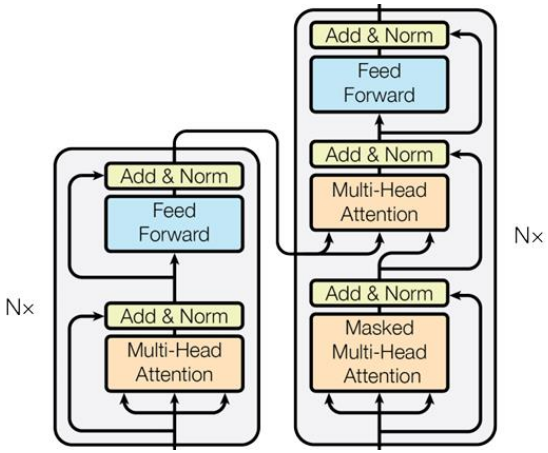
2) Multi-Head Attention의 출력 (= Attention vector)를 기존의 Embedding vector와 Add & norm

3) Feed Forward 계층을 통과하여 비선형성을 추가한 후 기존의 입력값과 Add & norm

4) 1~3의 Encoder 블록을 N번 반복한 후 Key와 Value를 Decoder의 Multi-Head Attention 블록으로 전달한다 (N=6)

Model Architecture

- Masked Multi-Head Attention



- **Transformer의 입력 형태**

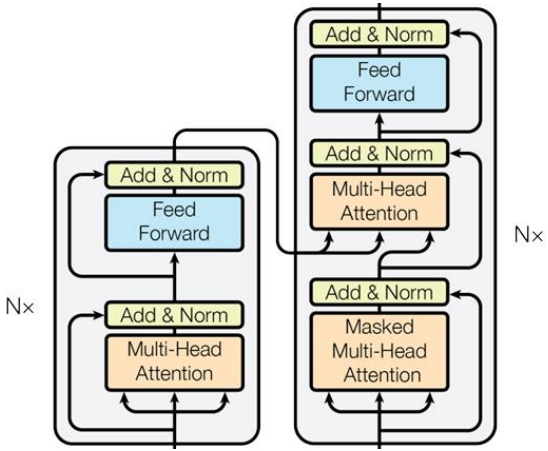
한 시퀀스 데이터의 모든 토큰을 행렬 형태로 입력하여 병렬 연산

Encoder: Key와 Value는 문장 전체 맥락을 반영하여야 한다

Decoder: 예측 시점의 Query는 이후 시점의 단어를 반영하면 안된다, 이후 시점의 단어를 mask하여 Multi-Head Attention

Model Architecture

- Decoder 구조



1) Embedding Vector를 대상으로 Masked Multi-Head Attention

2) Masked Multi-Head Attention의 출력과 Embedding vector를 Add & norm

3) 1~2의 출력값이 Decoder의 Multi-Head Attention 블록의 W_Q와 곱해져 Query 생성

   Encoder의 출력값이 Decoder의 Multi-Head Attention 블록의 W_K, W_V와 곱해져 key와 value 생성

4) 3에서 만든 Query, Key, Value를 대상으로 Multi-Head Attention 이후 Masked Multi-Head Attention 블록의 출력값과 Add & norm

5) FFN 계층 통과 후 입력값과 Add & norm

6) 1~5의 과정을 N번 반복

Discussion

- **Multi-Head Attention**

Self-Attention을 여러 번 수행 > 각각의 attention vector concat > linear transform

1 self-attention을 여러 번 수행하는 이유?

각 단어가 여러 맥락을 가질 수 있다
각각의 head에 서로 다른 가중치를 초기화한 후 self-attention을 수행하여 여러 맥락을 고려할 수 있도록 한다

2 add & norm 대신 concat 후 linear trasnform하는 이유?

각 head의 attention vector를 concat하여 고차원 벡터로 유지하면, 다양한 정보를 독립적으로 유지할 수 있다
이후 선형 변환을 통해 더욱 단어의 복잡한 관계를 학습할 수 있다

Discussion

- **Positional Encoding**

1) position encoding의 방식

- Fixed 방식: 토큰의 임베딩 차원, 위치 정보 등을 바탕으로 sin/cos 함수를 통해 고정적인 위치 정보를 학습
- Leanable 방식: 토큰의 position vector를 학습 가능한 파라미터로 설정 후 학습 과정에서 위치 정보를 학습할 수 있도록 한다

2) 위치 정보를 임베딩 벡터에 반영하는 방식

Position vector를 embedding vector와 같은 차원으로 만들고, 이를 기존의 embedding vector에 더한다

Q1) Fixed 방식에서 sin/cos 함수를 통해 만든 position vector가 어떻게 위치 정보를 반영할까?
Q2) 위치 정보를 반영하는 position vector와 단어의 의미를 반영하는 context vector를 단순히 더했을 때 벡터의 의미가 훼손되지 않을까?