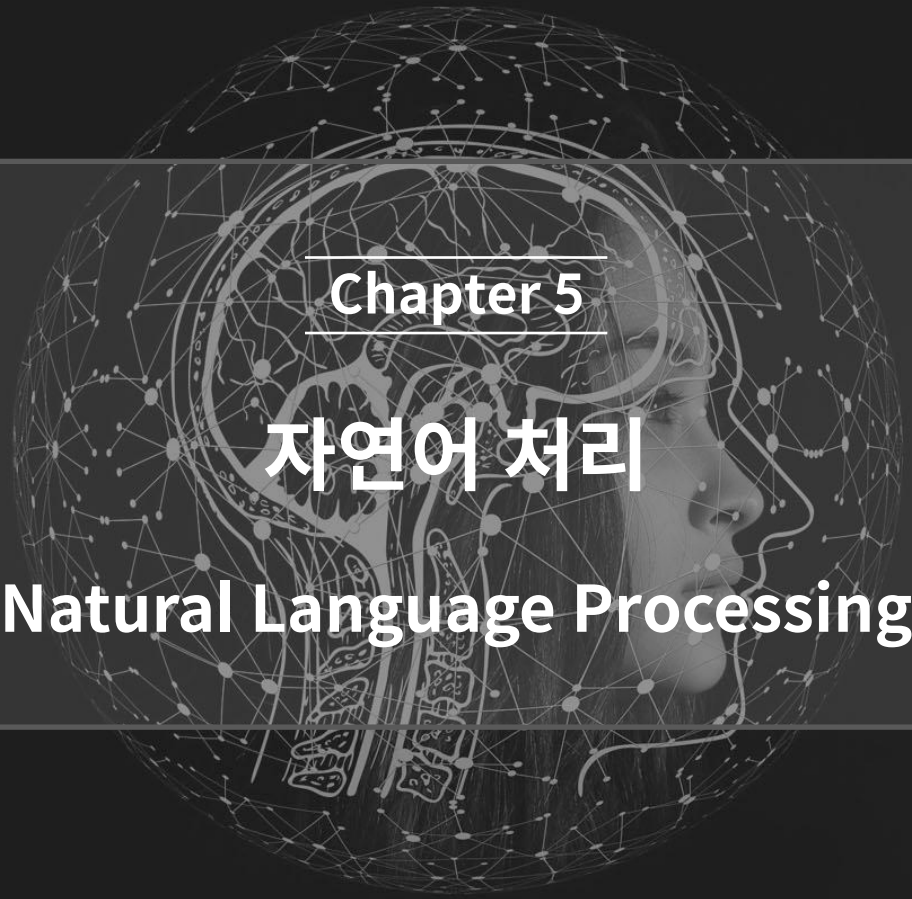


Chapter 5

자연어 처리

Natural Language Processing

자연어처리란?

자연어 처리 : Natural Language Processing (NLP)

자연어를 이해하는 영역인 자연어 이해 : Natural Language Understanding (NLU)

모델이 자연어를 생성하는 영역인 자연어 생성 : Natural Language Generation (NLG)

- NLU : Text ⇒ Meaning
- NLG : Meaning ⇒ Text
- NLP = NLU + NLG

Data & Task

Sentiment Analysis (감정 분석)

- 문장에 대한 정보를 통해 분류하는 문제 중 하나로 그 중 문장에 대한 특정 감정을 분류해내는 문제로 NLP의 대표적인 Task라고 할 수 있음.
- ' 이 집은 맛집이야 ', ' 이 영화는 재미 없어 ' 이러한 문장 속 의미가 긍정인지 부정인지를 판단하는 것이라고 볼 수 있음
- Dataset : The Stanford Sentiment Treebank (SST-2)

Summarization (요약)

- 요약 분야는 주어진 Text에서 중요한 부분을 찾아내는 **Extractive Summarization**과 모델이 주어진 Text의 의미를 완전히 이해하여 이를 요약하는 새로운 문장을 만들어내는 **Abstractive Summarization**으로 나뉘어 짐
- 요약은 모델링의 어려움 뿐 아니라, 정답 요약문을 만들어야 하는 데이터 수급 문제, 정답 평가의 어려움도 있어 NLP에서 난이도가 높은 Task임
- Dataset : CNN/DailyMail

Data & Task

Machine Translation (기계 번역)

- 구글 번역기, 파파고와 같은 언어 번역 분야
- Dataset : The Workshop On Machine Translation (WMT) 2014 English-German dataset (WMT English to German)

Question Answering (질문 응답)

- 주어진 문서를 이해하고, 문서 속 정보에 대한 질문을 했을 때 답을 이끌어내는 Task
- Dataset : Stanford Question Answering Dataset (SQuAD)와 Conversational Question Answering systems (CoQA) 등이 있으며, 한국어 version의 korQuAD 도 존재

Data & Task

etc

- 각 단어의 품사를 예측하는 **Part-Of-Speech Tagging (POS Tagging) 분야**, 많은 회사들이 연구하고 있는 **챗봇 연구**, 문장 간의 논리적인 관계에 대한 **분류 모델**, 각 단어의 중의적 표현을 구분해내는 **Word Sense Disambiguation (WSD)**, 주어진 이미지 속 상황을 설명하는 글을 만들어내는 **Image Captioning** 등등 다양한 문제가 존재하고, 이에 대한 데이터와 모델에 대한 연구들이 활발히 진행 중임.
- 더 많은 Task들이 대해서 궁금하다면, 다음 사이트들을 참고
  - PaperWithCodes : https://paperswithcode.com/area/natural-language-processing
  - NLPPogress : http://nlpprogress.com/

Vectorization : 문자를 숫자로 표현하는 방법

인간이 문자를 인식하는 방법

- 사람들은 문장을 의미를 가지고 있는 부분들로 쪼개고, 그 부분의 의미들을 조합해서 문장의 의미를 만드는 방식을 사용(영어 문장을 주어, 동사, 목적어, 수식어 등으로 나누고, 각각의 의미를 조합해서 문장의 의미를 만들어가는 방법 과 같이)
- 이 방식과 비슷하게 아래와 같은 순서를 생각해볼 수 있음

1. 문장을 의미 있는 부분 (단위)로 나눈다. (Text Segmentation)
2. 나눠진 의미 있는 부분들을 숫자로 바꿔서 문장을 숫자로 표현한다. (Representation)

Vectorization : 문자를 숫자로 표현하는 방법

컴퓨터가 문자를 인식하는 방법

1. 문장을 의미 있는 부분 (단위)로 나눈다. (Text Segmentation)

- 가장 간단한 방법은 띄어 쓰기를 이용하는 것

```
S1 = '나는 책상 위에 사과를 먹었다'
S2 = '알고 보니 그 사과는 Jason 것이었다'
S3 = '그래서 Jason 에게 사과를 했다'
```

- Python의 Split함수를 이용해 쉽게 나눌 수 있음

```
print(S1.split())
# ['나는', '책상', '위에', '사과를', '먹었다']

print(S2.split())
# ['알고', '보니', '그', '사과는', 'Jason', '것이었다']

print(S3.split())

# ['그래서', 'Jason 에게', '사과를', '했다']
```

- 위와 같이 문장을 의미 있는 부분들로 나누는 과정을 Tokenization 라고 하며, 쪼개진 부분들을 Token 이라고 부름

Vectorization : 문자를 숫자로 표현하는 방법

컴퓨터가 문자를 인식하는 방법

```
token2idx = {}
index = 0

for sentence in [S1, S2, S3]:
    tokens = sentence.split()
    for token in tokens:
        if token2idx.get(token) == None:
            token2idx[token] = index
            index += 1

print(token2idx)

# {'나는': 0, '책상': 1, '위에': 2, '사과를': 3, '먹었다': 4,
#  '알고': 5, '보니': 6, '그': 7, '사과는': 8, 'Jason': 9,
#  '것이었다': 10, '그래서': 11, 'Jason 에게': 12, '했다': 13}
```

- 위와 같이 Token을 저장해 놓은 사전인 token2idx을 Vocabulary 라고 하며, Token의 저장과 관리 뿐 아니라 저장할 때 같이 저장한 Index를 이용해 문자를 숫자로 바꾸는데 참조하는 데 사용

Vectorization : 문자를 숫자로 표현하는 방법

컴퓨터가 문자를 인식하는 방법

사전을 Token에 해당하는 Index를 정해두었기 때문에 Token의 해당 숫자로 각 문장을 바꿔보면 다음과 같이 나타낼 수 있음

(오래전 부터 컴퓨터에게 텍스트를 인식 시킨 방법)

```
def indexed_sentence(sentence):
    return [token2idx[token] for token in sentence]

S1_i = indexed_sentence(S1.split())
print(S1_i)
# [0, 1, 2, 3, 4]

S2_i = indexed_sentence(S2.split())
print(S2_i)
# [5, 6, 7, 8, 9, 10]

S3_i = indexed_sentence(S3.split())
print(S3_i)
# [11, 12, 3, 13]
---- 결과 ----

S1 = '나는 책상 위에 사과를 먹었다'        => [0, 1, 2, 3, 4]
S2 = '알고 보니 그 사과는 Jason 것이었다'  => [5, 6, 7, 8, 9, 10]
S3 = '그래서 Jason 에게 사과를 했다'       => [11, 12, 3, 13]
```

Vectorization : 문자를 숫자로 표현하는 방법

컴퓨터가 문자를 인식하는 방법

```
S1 = '나는 책상 위에 사과를 먹었다'          => [0, 1, 2, 3, 4]
S2 = '알고 보니 그 사과는 Jason 것이었다'   => [5, 6, 7, 8, 9, 10]
S3 = '그래서 Jason 에게 사과를 했다'         => [11, 12, 3, 13]
```

위처럼 표현된 숫자 나열을 딥러닝 모델에 넣고 학습을 돌린다면, 문제가 생길 것. 1+1=2 이지만, "책상" 2개를 더 한다고 "위에" 뜻 하지 않게 되고, "위에"와 "사과를" 곱한다고 "보니" (2*3 = 6 이지만)이 되지는 않기 때문. 딥러닝 모델 학습 과정에서 일어나는 모든 연산은 숫자를 기초로 하는 사칙 연산을 기본으로 두겠지만, 숫자가 표현하고 있는 본연의 의미 변화는 전혀 다르기 때문

```
1+1 =2 → "책상" + "책상" ≠ "위에"

2*3 =6 → "위에" * "사과를" ≠ "보니"
```

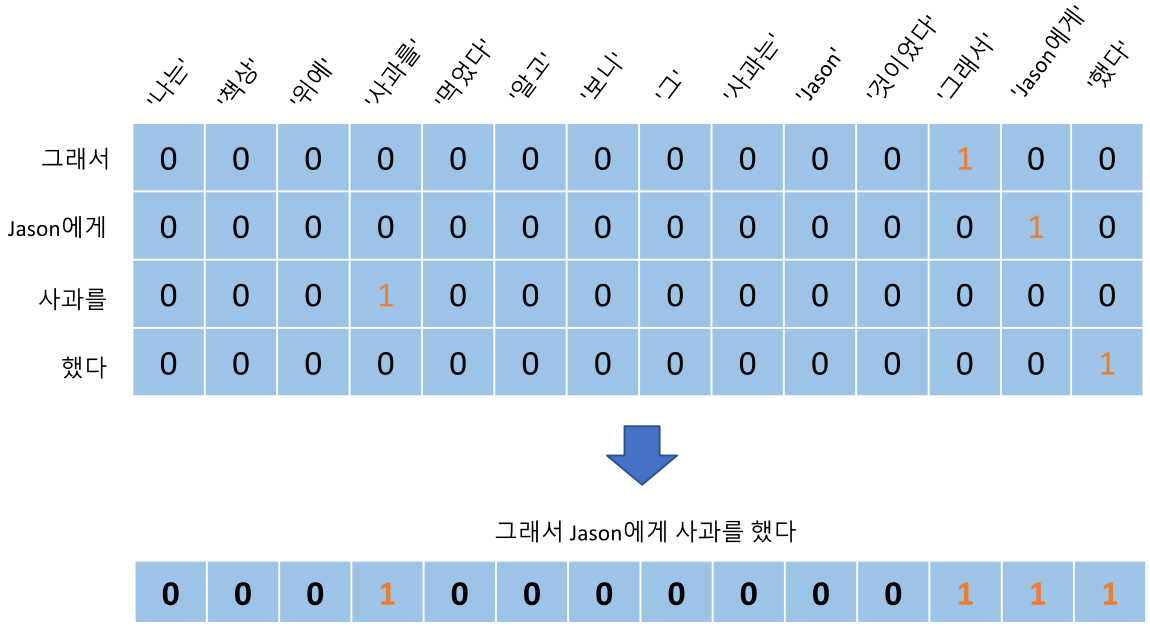
Word Embedding

One-Hot Encoding

Corpus를 모두 Tokenization하여 Vocabulary을 만들고, 각 Token마다 Index를 정함.

단어 사전의 크기를 V = len(token vocabulary)라 하고, V 길이의 모두 0값을 가진 0 벡터를 생성

각 Token은 그에 해당하는 Index의 값만 1의 값을 가진 벡터로 표현



Word Embedding

Dense Representation

One-Hot Encoding을 통한 문자 표현의 문제점 중 한 가지는 변수의 Sparseness (희소성)

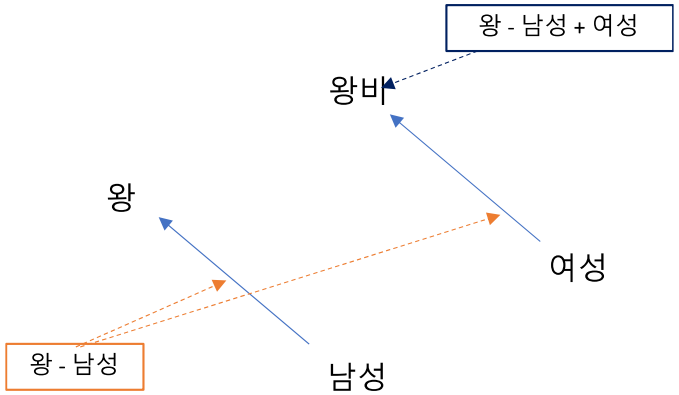
- 모든 Token의 갯수인 V만큼의 벡터를 만드는 것 뿐 아니라, 대부분은 0이면서 극소수의 위치에만 값을 가지는 있기 때문에 상당히 큰 메모리 사용량을 필요로 하지만, 대부분은 0으로 되어있어 비효율적인 사용이라고 볼 수 있기 때문
- 2013년 구글의 논문 Efficient Estimation of Word Representations in Vector Space (Tomas Mikolov et al.) 으로 인해 NLP에서의 문자 표현 방식은 큰 전환기를 맞게 됨

Word Embedding

Dense Representation

- Word2vec

Word2vec이라는 이름으로 소개된 이 방법은 vector("King”) - vector(”Man”) + vector(”Woman") = vector("Queen") 이라는 예제를 통해 관계를 연산으로 설명할 수 있는 벡터 표현이 가능함을 보여주면서 관심을 받게 됨



"Token의 의미는 주변 Token의 정보로 표현된다"라는 가정. 특정 Token 기준으로 주변에 비슷한 Token들이 있다면 해당 Token은 비슷한 위치의 벡터로 표현되도록 학습을 시킴. 예를 들면, "나는 책상 위에 사과를 먹었다", "나는 책상 위에 배를 먹었다", "나는 책상 위에 숙성회를 먹었다" 라는 문장들의 "사과를", "배를", "숙성회를" 빼고는 모두 주변 Token이 같음. 이런 데이터 학습하는 경우, 위 3개의 Token은 서로 비슷한 위치의 벡터로 학습을 시키는 것

Word Embedding

Dense Representation

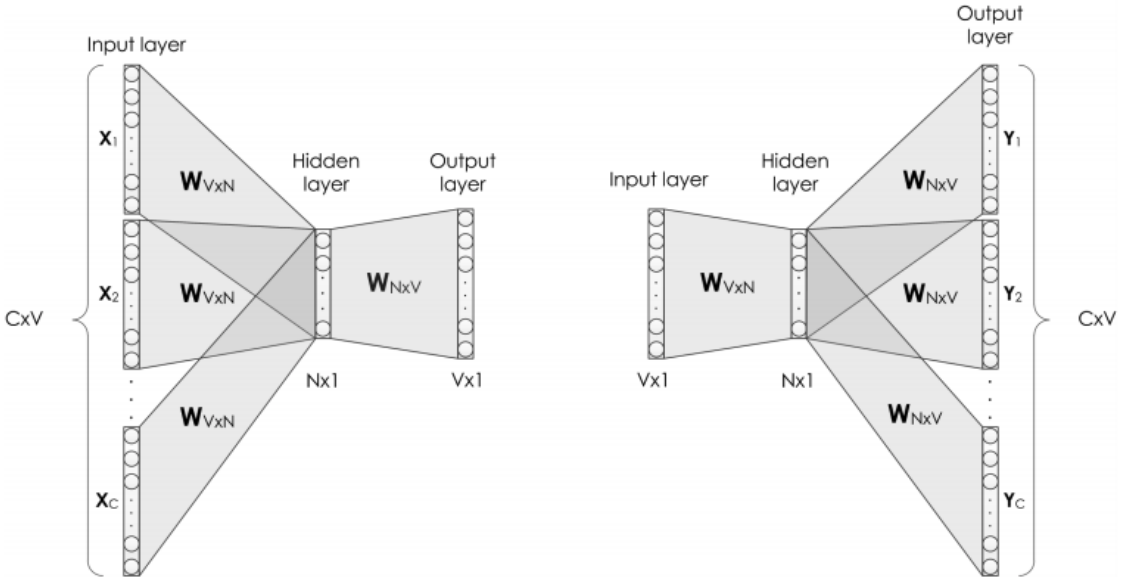
- Word2vec

- 논문에서는 학습하는 과정을 CBOW (Continuous Bag-of-Words Model) 과 Skip-Gram 2가지를 소개
- 두 방법 모두 공통적으로는 문장을 Window 형태로 부분만 보는 것을 기본으로 시작. 기준 Token 양 옆 Token들을 포함한 Window가 이동하면서, Window 속 Token들과 기준 Token의 관계를 학습시키는 과정을 진행. 이 때, 주변 Token들을 Context라고 표현하고, 기준 Token을 Target으로 지칭
- CBoW 는 Context Token들의 벡터로 변환하여 더한 뒤 Target Token를 맞추는 것을 목적으로 하였고, Skip-Gram은 반대로 Target Token을 벡터로 변환한 뒤 Context Token들을 맞추는 것을 목적으로 모델링

Word Embedding

Dense Representation

- Word2vec 모델 구조



CBOW Skip-gram

Word Embedding

Dense Representation

- Word2vec 학습 과정

Notation (Window 크기는 1로 가정)

$$V : Vocabulary\ Size\ (the\ number\ of\ tokens)$$
$$D : Embedding\ Dimension$$
$$W_{V \times D} : Weight\ matrix\ (input - hidden)$$
$$W'_{D \times V} : Weight\ matrix\ (hidden - output)$$
$$target\ token : x_i$$
$$context\ tokens : x_{i-1}, x_{i+1}$$

Word Embedding Vector

x_i 는 Vocabulary에서 Token의 해당 Index 부분만 1을 가지고 나머지는 모두 0인 벡터. 이 벡터와 W를 곱하게 된다면, W 행렬의 해당 index 부분의 행만 남게 되는데 이 행의 값을 Word Embedding Vector 라고 함

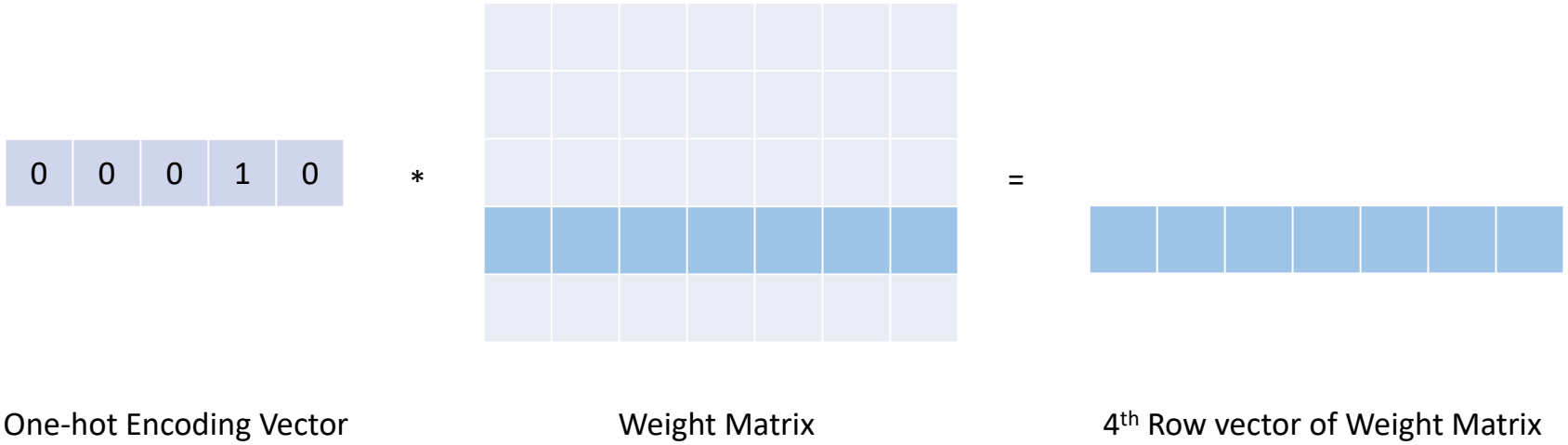
Word Embedding

Dense Representation

- Word2vec 학습 과정

Word Embedding Vector

$$x_i \cdot W = y$$



One-hot Encoding Vector | Weight Matrix | 4th Row vector of Weight Matrix

Word Embedding

Dense Representation

- Word2vec 학습 과정

CBOW

$$hidden\ layer = x_{i-1} \cdot W + x_{i+1} \cdot W$$

이 Hidden layer에 W' 행렬을 곱함으로써 단어 사전 중 어떤 Token이 가장 적합할지 구하는 Score를 구함. 이 벡터에 Softmax를 취함으로써 각 Token에 대한 확률 값을 구할 수 있음. CBoW의 학습과정은 Context Token을 Input으로 하였을 때 Target Token이 나올 수 있도록 하는 것으로 이해할 수 있음

$$P(x_i) = softmax(hidden\ layer \cdot W') = softmax\big((x_{i-1} \cdot W + x_{i+1} \cdot W) \cdot W'\big)$$

Word Embedding

Dense Representation

- Word2vec 학습 과정

Skip-Gram

Target Token의 Embedding Vector를 구해서 Hidden Layer를 구성하고, 여기에 W' 행렬을 곱해서 Context Token을 예측.
CBoW에서 Context Token의 임베딩 벡터를 더하는 과정이 빠지는 대신, Context에 해당하는 Token 마다 예측을 해야하는 과정이 들어감

$$hidden\ layer = x_i \cdot W$$

$$P(x_{i-1}) = softmax(x_i \cdot W \cdot W')$$

$$P(x_{i+1}) = softmax(x_i \cdot W \cdot W')$$

Word Embedding

Dense Representation

- Glove

관련 논문 : GloVe: Global Vectors for Word Representation (Pennington et al., 2014)

관련 링크 : https://github.com/stanfordnlp/GloVe

연구 단체 : Stanford Univ.

핵심 아이디어 : 기존의 Representation 기법에서 사용한 문서 내 모든 단어의 통계 정보와 Word2vec의 Local Context Window 정보를 동시에 사용하는 모델링

Word Embedding

Dense Representation

- Fasttext

관련 논문 : Enriching Word Vectors with Subword Information (Bojanowski et al., 2016)

관련 링크 : https://fasttext.cc/

연구 단체 : Facebook

핵심 아이디어 : 한개의 word에 대한 vector로 n-gram character에 대한 vector 평균을 사용

Word Embedding

Dense Representation

- BERT

관련 논문 : BERT : Pre-training of Deep Bidirectional Transformers for Language Understanding (Jacob Devlin et al., 2018)

관련 링크 : https://github.com/google-research/bert

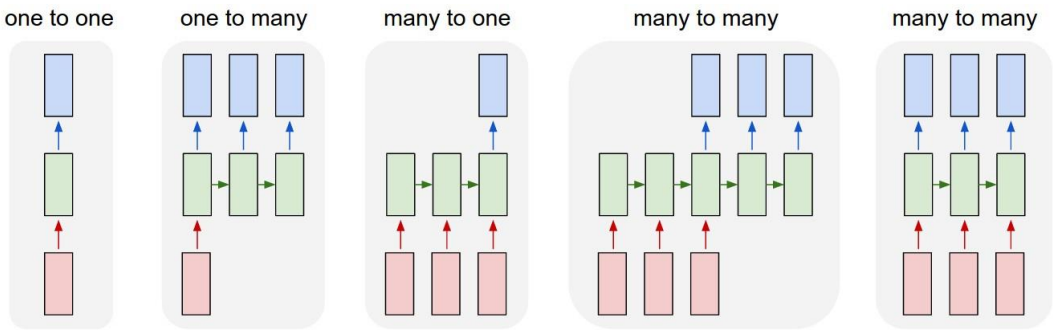
핵심 아이디어 : Transformers라는 Module의 성능의 알려지면서, 이에 관련한 방대한 양의 관련 연구가 이루어짐. 이는 문맥에 따라 단어의 Embedding Vector가 바뀔 수 있는 **Contextual Embedding**이라는 이름으로 기존의 Word Representation 영역의 또 다른 큰 변화를 가져옴.

Deep Learning Models

Recurrent Neural Network (RNN)

- 문장과 같은 문자 데이터 같은 경우, 여러개의 연관성이 깊은 Token들로 분할되어 표현됨.
- 즉, 문장을 Input이나 Output으로 사용할 경우, 한 개의 벡터 (One)가 아닌 다수의 벡터 (Many)의 형태로 사용해야 함.
- 그래서 이런 형태의 Sequential Data (순차 데이터)를 다루는 경우를 One-to-One이 아닌 Many-to-One 혹은 One-to-Many 문제라고 표현(뿐만 아니라 Input과 Output이 모두 Sequential Data인 Many-to-Many 문제도 있음) 이와 같이 문제를 분류할 경우, 각각의 분류에 해당하는 가장 대표적인 문제들은 다음과 같음

  - Many-to-One : 댓글의 악플 가능성 정도를 측정하는 Sentence Classification
  - One-to-Many : 사진 속 내용을 설명하는 글을 만들어 내는 Image Captioning
  - Many-to-Many (token-by-token) : 문장의 모든 token에 대한 품사를 예측하는 Pos Tagging
  - Many-to-Many (Encoder-Decoder) : 입력 문장을 다른 언어의 문장으로 번역해주는 Translation

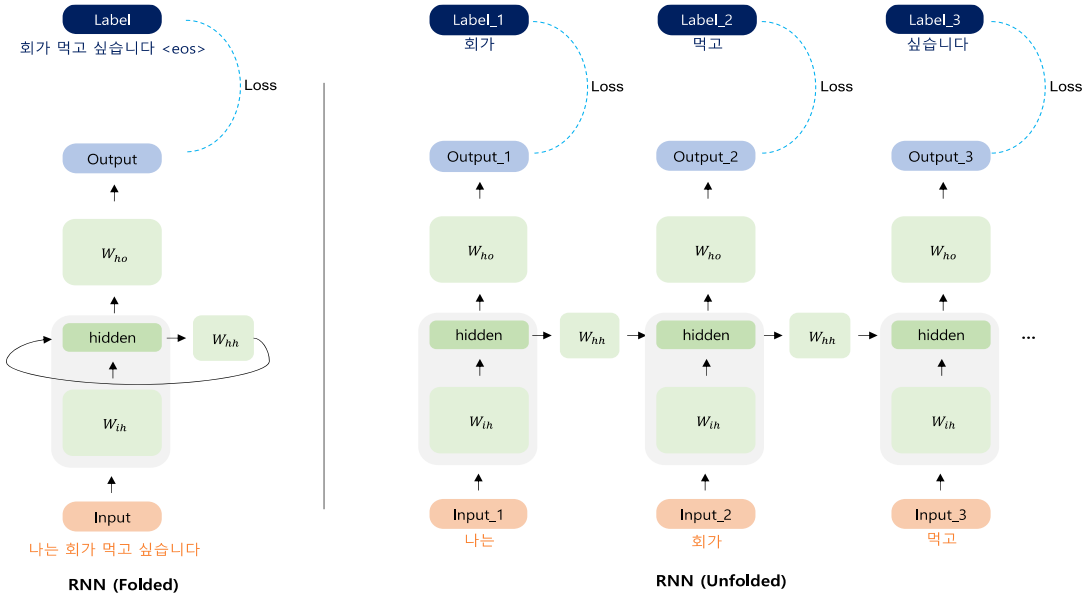


출처 : http://karpathy.github.io/2015/05/21/rnn-effectiveness/

Deep Learning Models

Recurrent Neural Network (RNN)

- Sequential Data을 다루는 가장 대표적인 모델은 Recurrent Neural Network (RNN)
- 단어의 순서대로 입력하고 반복해서 같은 Hidden Layer를 Update하면서 학습하는 구조로 설계
- 마치 사자성어나 속담 이어 말하기 게임처럼 "어물전 망신은"이나 "과유" 까지의 정보를 Hidden Layer에 담고 있으면 다음에 이어질 말은? 이라고 묻는 게임을 한다고 볼 수 있음



Deep Learning Models

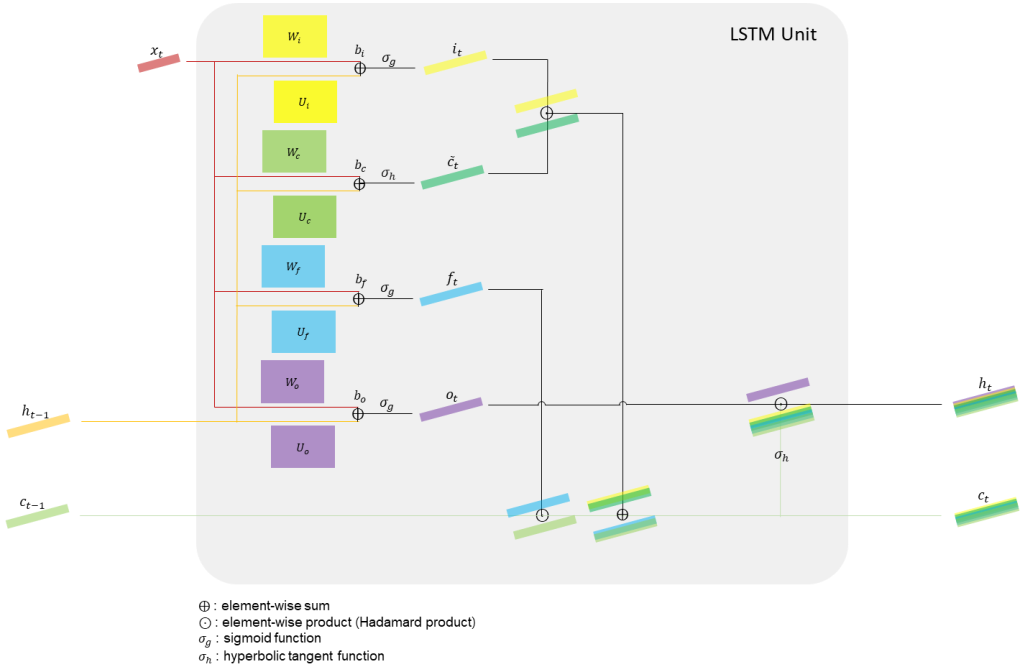
Recurrent Neural Network (RNN)

- 과도한 Back Propagation으로 인한 Gradient이 발산하거나 없어지는 현상인 Gradient Vanishing (or Exploding) 현상이 일어날 확률이 높음.
- 문장의 초반 Token에 대한 Weight의 Update 하기 위한 Gradient는 Chain Rule에 의해 문장 후반의 Gradient까지 모두 곱하여 만들어 짐. 따라서 문장이 길어지게 된다면 Gradient가 너무 작아지거나 커지는 Gradient Vanishing (or Exploding) 현상으로 이어지게 되고, 이런 경우 정상적인 학습이 어려워짐.
- 이를 해결하기 위해 해당 Token의 과거 지정 개수의 Token까지만 부분적으로 Update를 하는 Truncated Back Propagation이나 너무 큰 Gradient은 조정해서 Update하는 Gradient clipping 방법을 사용하거나 초기값을 잘 설정하거나 Activation Function을 ReLU로 설정하는 방법 등이 사용됨
- 또 한 가지의 문제점은 장기 기억력이 떨어진다는 점. 연속된 정보를 기억할 수 있는 모델 구조임에도 불구하고, 위와 같은 너무 작은 Gradient 영향으로 인해 모든 문장의 정보를 담기 어려웠음. 따라서 문장의 길이가 길어지는 경우 RNN 모델을 학습하는 것이 오래 걸리는 것 뿐 아니라 성능이 확실히 떨어지는 것을 확인할 수 있음.

Deep Learning Models

Long Short-Term Memory (LSTM)

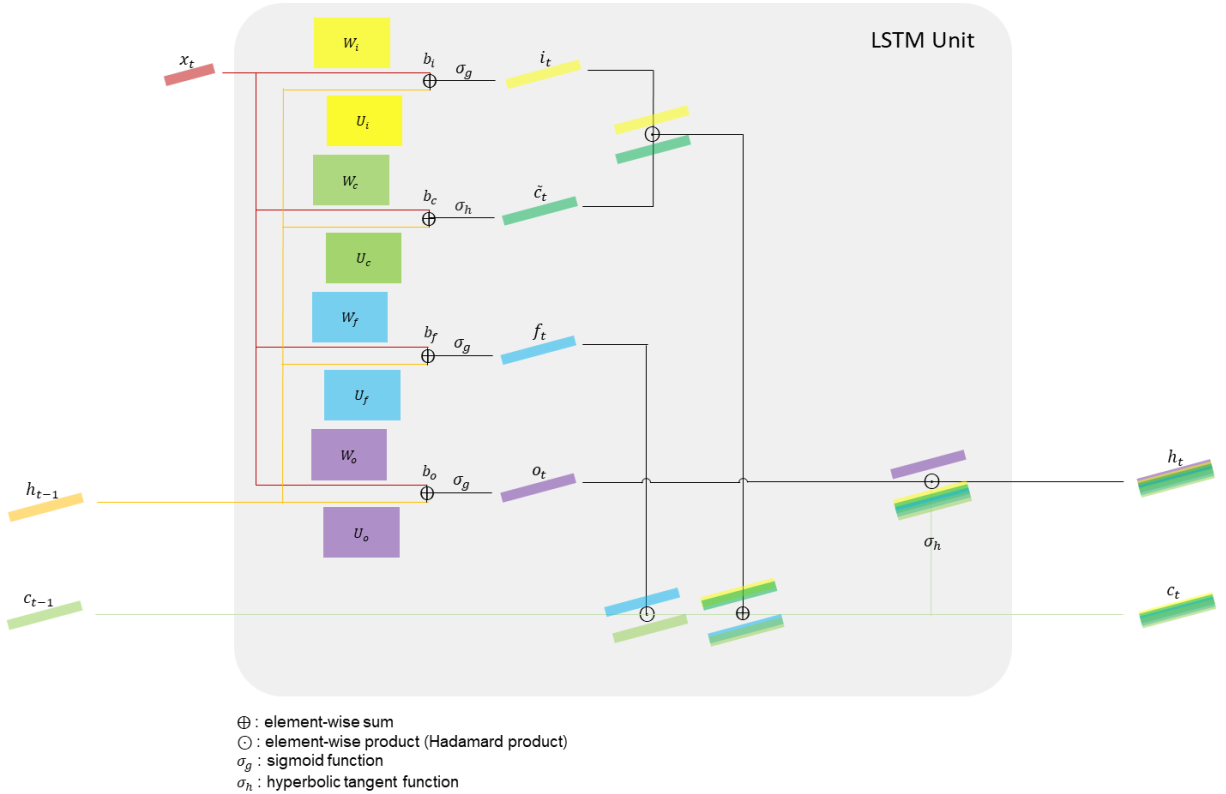
- 모델은 기존 RNN 모델의 순서 정보를 Hidden Layer에 담는 과정을 발전시켜 RNN의 큰 단점이었던 단기 기억만 가능하다는 부분을 개선시킴
- LSTM의 핵심은 Cell이라고 불리우는 c_t와 Gate $(i_t, f_t, \tilde{c_t}, o_t)$들을 통한 정보 필터링이라고 볼 수 있습니다. 특정 시점 t에 대한 데이터 x_t 와 그 이전 시점까지의 정보h_{t-1}로 각각의 Gate와 현재 시점의 Cell 정보 C_t를 만들고, 여기에 한 번 처리를 거쳐 최종형태의 정보 h_t를 만들어냅니다..



Deep Learning Models

Long Short-Term Memory (LSTM)

$$
\begin{aligned}
f_t &= \sigma_g(W_f x_t + U_f h_{t-1} + b_f) \\
i_t &= \sigma_g(W_i x_t + U_i h_{t-1} + b_i) \\
o_t &= \sigma_g(W_o x_t + U_o h_{t-1} + b_o) \\
\tilde{c}_t &= \sigma_h(W_c x_t + U_c h_{t-1} + b_c) \\
c_t &= f_t \circ c_{t-1} + i_t \circ \tilde{c}_t \\
h_t &= o_t \circ \sigma_h(c_t)
\end{aligned}
$$

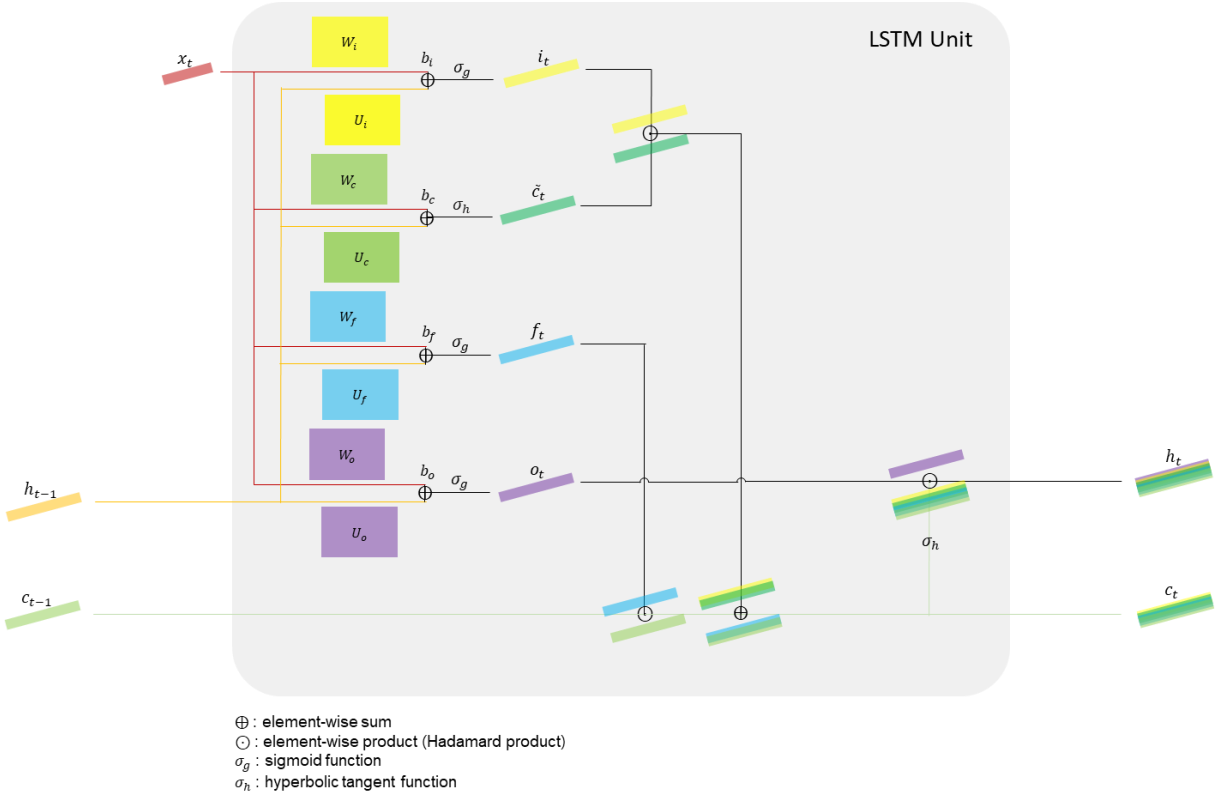


Deep Learning Models

Long Short-Term Memory (LSTM)

$$\tilde{c}_t = \sigma_h(W_c x_t + U_c h_{t-1} + b_c)$$
$$c_t = f_t \circ c_{t-1} + i_t \circ \tilde{c}_t$$

- c_t 는 이전 정보까지의 정보를 담고 있는 Cell (c_{t-1})에서 얼마나 잊을지를 나타내는 Gate f_t 를 통과한 값과 현재 Token에 대한 정보($\tilde{c}_t$)를 얼마나 가져올지 정하는 Gate i_t 를 통과한 값의 합산.
- 간단히 표현하자면 과거의 모든 정보 중 불필요하다고 판단되는 것은 지우고, 현재 Token에 대한 정보를 적당히 합산해서 정보를 만드는 과정이라고 볼 수 있음.

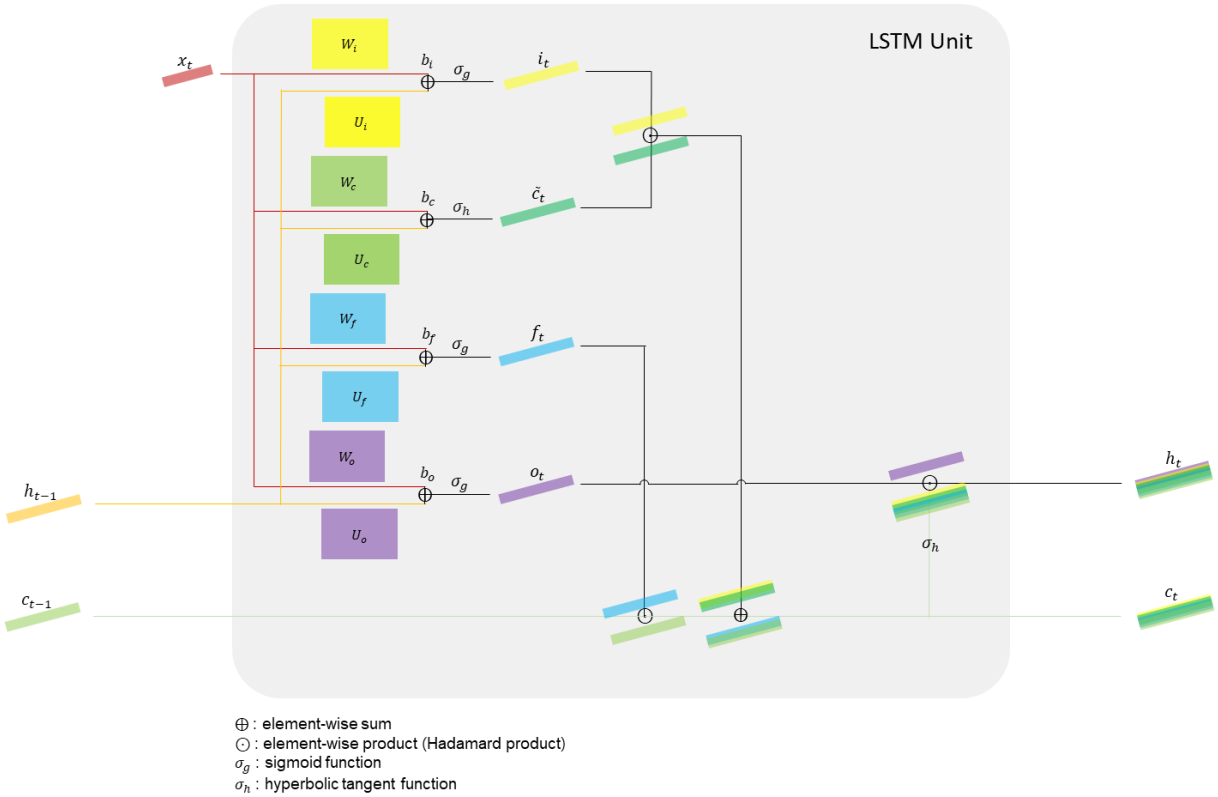


Deep Learning Models

Long Short-Term Memory (LSTM)

$$f_t = \sigma_g(W_f x_t + U_f h_{t-1} + b_f)$$
$$i_t = \sigma_g(W_i x_t + U_i h_{t-1} + b_i)$$

- 정보의 필터링을 담당하는 Gate들인 f_t 와 i_t 를 자세히 살펴보면, 이전까지의 정보 h_{t-1} 와 현재 Token x_t 값을 조합해서 구한다고 볼 수 있음
- Activation Function이 약간 다른 것을 볼 수 있는데, Sigmoid 함수를 사용하여 Component마다 0~1의 값을 가지게 되고 이를 Component-Wise하게 곱해서 벡터 각각의 Component 정보를 얼마나 쓸 것인지 정한다고 볼 수 있음

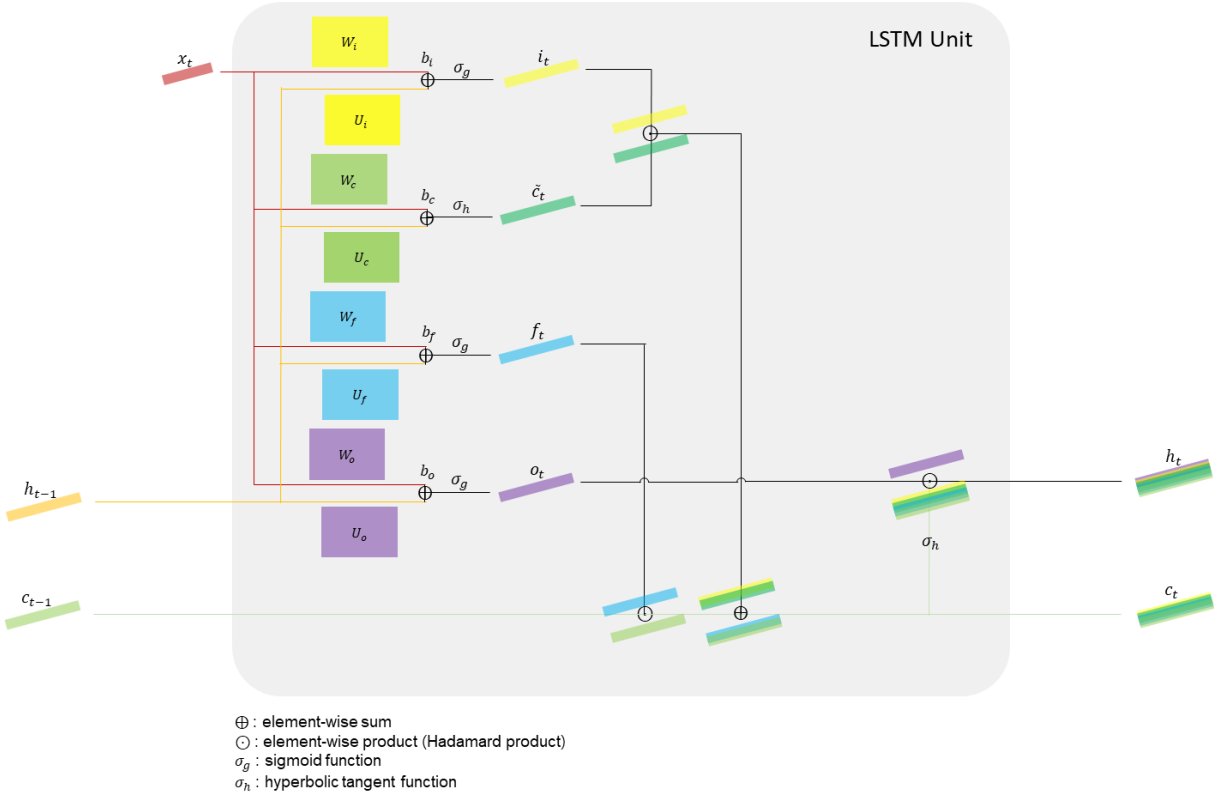


Deep Learning Models

Long Short-Term Memory (LSTM)

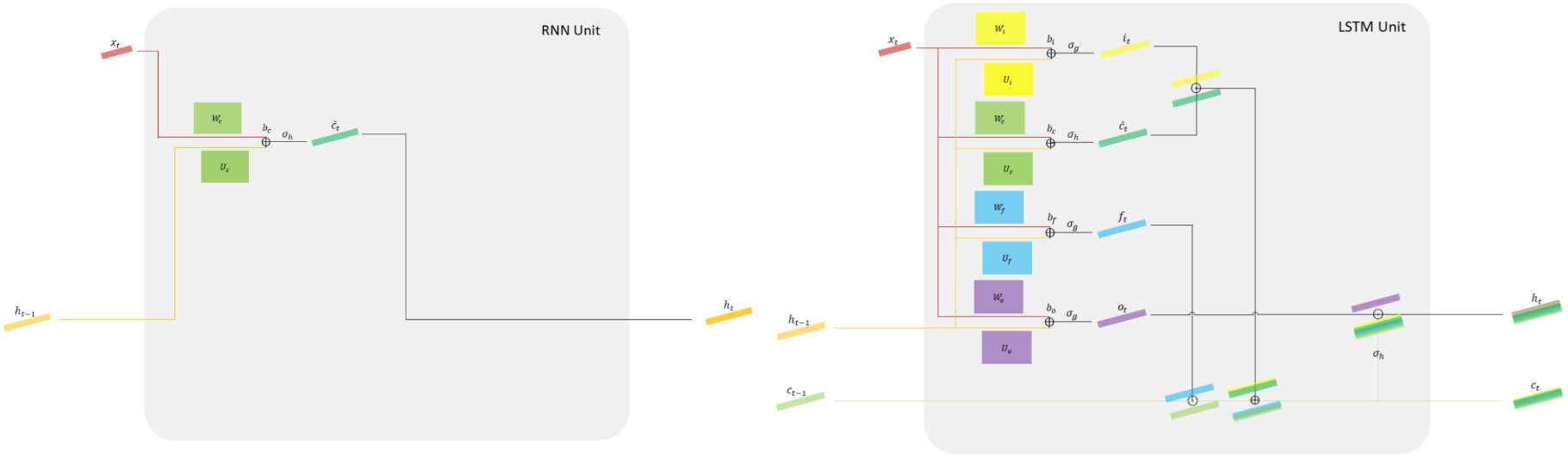
$$o_t = \sigma_g(W_o x_t + U_o h_{t-1} + b_o)$$
$$h_t = o_t \circ \sigma_h(c_t)$$

- Cell 정보는 Output Gate(o_t)를 거쳐가면서 또 한번의 정보 수정을 진행하여 최종적인 Hidden Layer (h_t)를 구합니다



Deep Learning Models

Long Short-Term Memory (LSTM) vs RNN

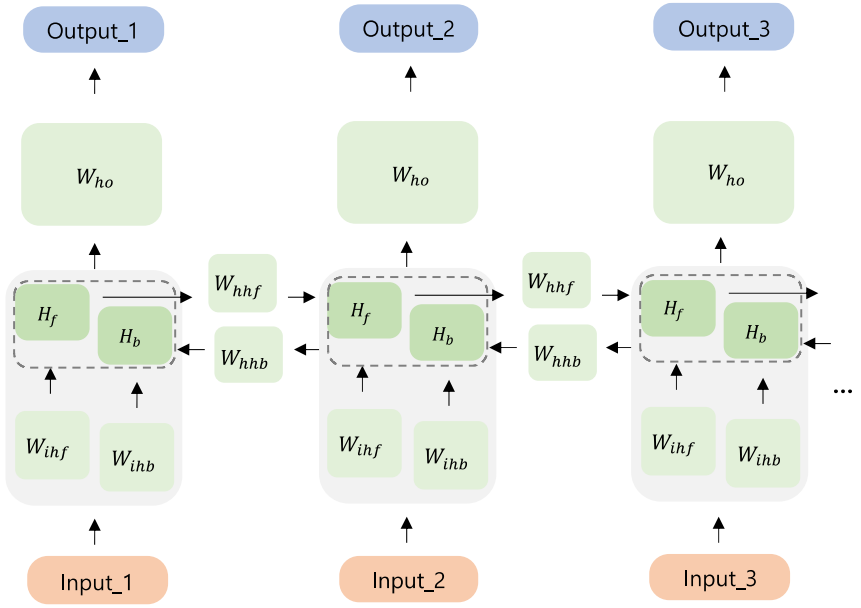


- RNN에서 Gradient Vanishing이 일어나게 되는 주된 문제는 Token 사이의 거리가 먼 경우의 Hidden Layer h_t 간의 Chain Rule 연산에서 나타남
- Token간 사이가 멀어질수록 Gradient가 사라지는 형태가 되어 학습이 잘 되지 않는 현상이 벌어짐
- LSTM은 Cell c_t 를 이용하여 곱해지는 값의 차이가 심각하지 않게 만듦
- 위와 같은 상황에서 LSTM의 경우에는 f_t 값만 계속 곱하게 되는 형태인데, f_t 의 Activation Function인 Sigmoid 값이 1에 가까울수록 Gradient 값이 작아지는 상황을 줄여줄 것. 이는 f_t 의 Gate가 다 열린다는 뜻으로, 과거 데이터들을 모두 가져오겠다는 뜻과 유사하다고도 해석할 수 있음

Deep Learning Models

Bidirectional Recurrent Neural Networks (Bi-RNNs)

- 논문 : Bidirectional Recurrent Neural Networks, M. Schuster and K. K. Paliwal, 1997
- 기존의 RNN류의 모델이 모두 왼쪽에서 오른쪽 방향으로의 정보를 담는 형태의 모델들이었다면, 오른쪽에서 왼쪽으로의 정보도 같이 담아서 양방향 정보를 모두 이용하자는 것이 핵심 아이디어
- 양방향의 Hidden Layer (Forward, Backward)를 Concatenate해서 사용
- 모델 구조



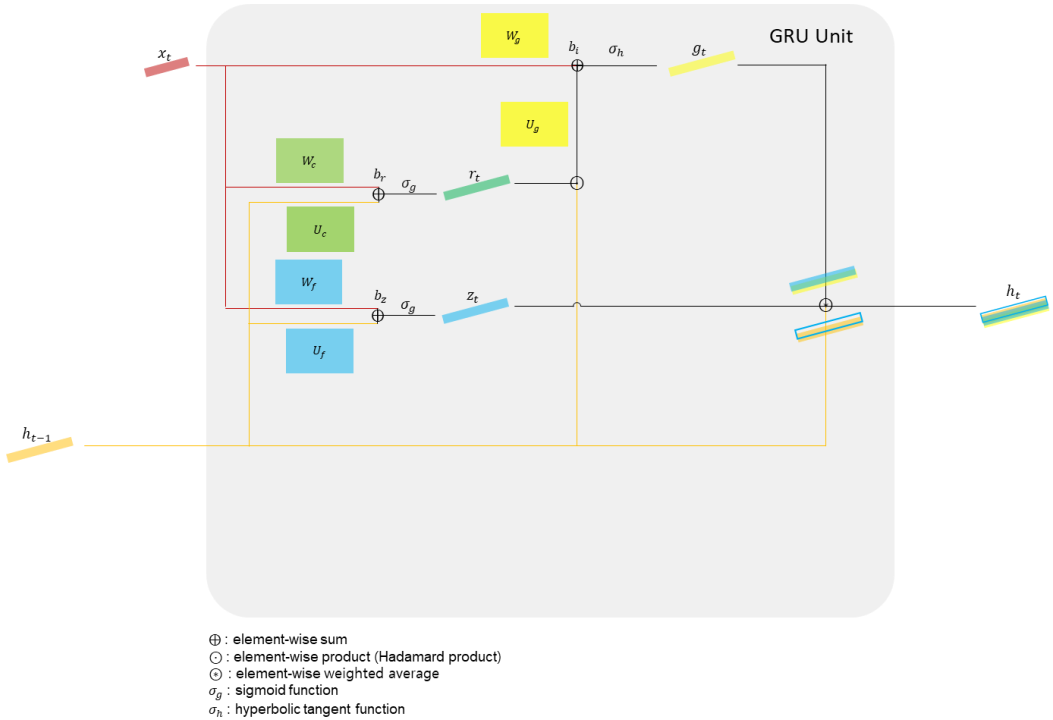
Deep Learning Models

Gated recurrent units (GRUs)

- 논문 : Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling, Cho et al., 2014
- LSTM의 Output 부분을 제거한 간소화 version으로 알려진 모델
- 최종 Hidden Layer를 과거의 누적 정보(h_(t-1))와 현재 정보((h_t) ̂)의 Weighted Average를 이용해서 표현
- 모델 수식

$$z_t = \sigma_g(W_z x_t + U_z h_{t-1} + b_z)$$
$$r_t = \sigma_g(W_r x_t + U_r h_{t-1} + b_r)$$
$$\widehat{h_t} = \phi_h(W_h x_t + U_h(r_t \odot h_{t-1}) + b_h)$$
$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \widehat{h_t}$$

- 모델 구조



Deep Learning Models

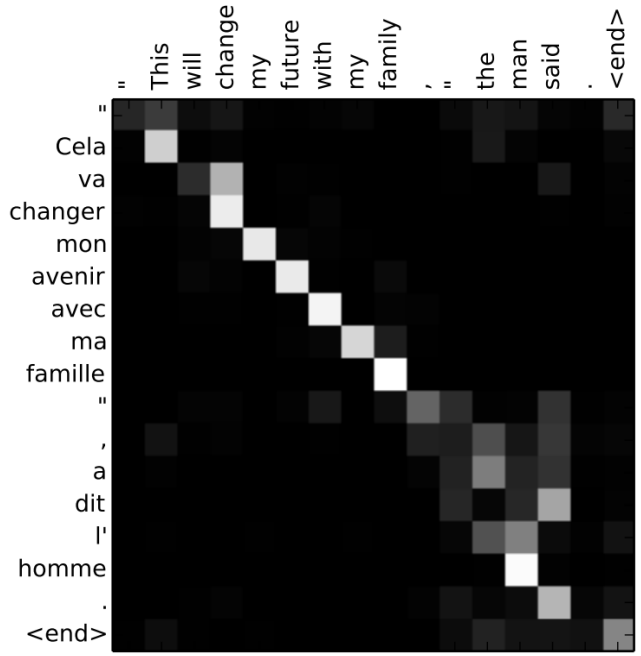
Attention Mechanism

- 논문 : NEURAL MACHINE TRANSLATION BY JOINTLY LEARNING TO ALIGN AND TRANSLATE, Bahdanau et al., 2015
- NLP에서의 Attention Mechanism 활용의 시작을 알린 논문
- Neural Machine Translation 을 만드는 경우 주로 RNN Encoder-Decoder 구조의 모델을 사용하였는데 아래 2가지의 Module로 구성
- 번역하고자 하는 문장 Source Sentence 의 정보를 저장하는 RNN Encoder
- 위에서 저장된 정보를 토대로 변역된 문장을 생성하는 RNN Decoder
- 결과
  - 번역문을 만드는 Decoder에서 원 문장의 어떤 단어에 집중해서 단어를 만들어낼 것인지에 대한 정보 Attention Score를 이용함으로써 긴 문장에서도 좋은 성능을 보여줌.
  - Attention을 이용해서 다른 언어이지만 같은 뜻을 나타내는 Pair에 대한 정보를 활용할 수 있었으며, 이를 시각화를 통해 증명.
  - 해당 논문을 시작으로 NLP에서 Attention Mechanism이 활발히 연구 되었으며, NLP에서 딥러닝 모델의 큰 변화를 가져온 Transformer Module의 핵심 아이디어가 되기도 함,
  - Attention Mechanism의 연구 흐름 참고(https://lilianweng.github.io/lil-log/2018/06/24/attention-attention.html)

Deep Learning Models

Attention Mechanism

- 아래 그림은 번역하고자 하는 문장(영어)과 번역한 문장(불어)의 각 Token별 Attention score가 높은 부분을 밝게 표현한 그림
- 한마디로 번역문의 Token을 생성할 때 번역하기전 문장의 어떤 Token에 집중해서 생성했는지를 볼 수 있는 그림.
- 대부분 어순에 따라 Score가 높은 것을 볼 수 있는데, 논문 저자는 영어 "the"를 불어 " l’ "로 번역되는 부분을 주의깊게 봄.
- " the " 를 바꿔주는 과정에서 뒷 단어가 " man " 임을 Attention으로 동시에 보고 있었고, 이를 불어로 번역하는 과정에서 남자에 대한 관사 " l’ " 를 선택했다는 부분이 흥미로운 부분.

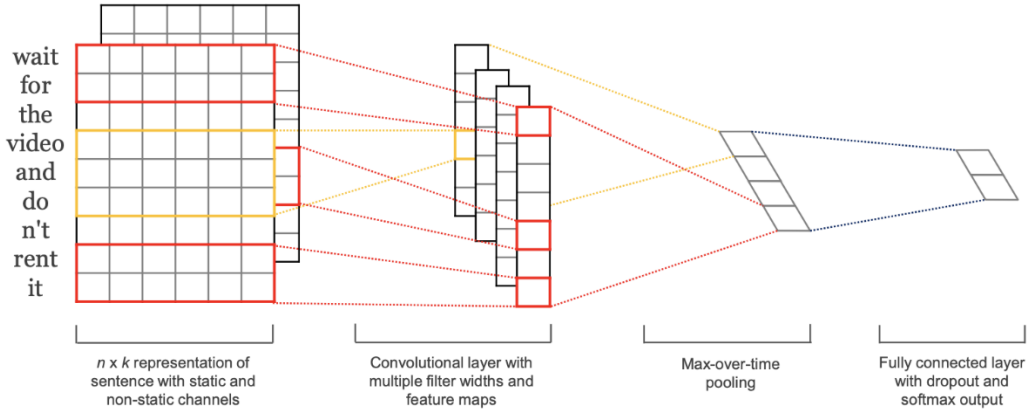


출처 : NEURAL MACHINE TRANSLATION BY JOINTLY LEARNING TO ALIGN AND TRANSLATE (https://arxiv.org/abs/1409.0473)

Deep Learning Models

Convolution Neural Network for Text classification

- NLP에서의 딥러닝 모델은 RNN 종류의 사용이 주를 이루었음.
- 하지만 이미지 데이터에서 애용하던 CNN을 NLP에 적용하여 좋은 결과를 보여준 모델도 존재.
- 아래 논문을 시작으로 RNN 계열이 아닌 CNN의 좀 더 복잡한 모형으로의 발전과 적용 등으로 연구가 이어짐.
  - 논문 : Convolutional Neural Networks for Sentence Classification, Yoon Kim, 2014
  - RNN의 구조적 한계로 인해 할 수 없었던 병렬 처리를 CNN을 이용하여 해결
  - 모델 구조



출처 : Convolutional Neural Networks for Sentence Classification (https://www.aclweb.org/anthology/D14-1181/)

Pre-Trained Model의 시대

Attention is all you need (The Transformer)

- "Attention만 있으면 된다"는 제목답게 이 논문에서 제시하고 있는 기계 번역 모델은 오직 Attention 개념만 가지고 만들어져 있음.
- Attention은 보통 RNN류의 모델의 보조 정보 혹은 보조 장치로써의 좋은 성능을 보여옴.
- 하지만 이 모델은 RNN 계열의 Module을 아예 쓰지 않는 형태로 구조를 설계.
- 그럼에도 불구하고 기존 최고의 성능을 보였던 모델 결과 대비 좋은 성능을 보여줌
- 모델구조
  - 모델의 큰 구조는 기계번역에서 자주 사용하는 Encoder-Decoder 형식.
  - 번역하고자 하는 문장 (Source Sentence)을 Input으로 입력하여 Encoder가 특정 벡터로 정보를 저장한 뒤, Decoder가 해당 정보를 통해 번역문을 만들어내면 정답 번역 문장(Label)과 Loss 구해서 학습하는 구조.
  - 하지만 보통의 Encoder와 Decoder는 RNN류의 LSTM이나 GRU Module을 사용하고, Attention을 적용하는 방식을 사용했는데, The Transformer는 해당 부분을 RNN을 전혀 쓰지 않고 여러 개의 Module을 이어서 만듦
  - 모든 Token을 순서대로 입력 받는 RNN과 다르게 모든 Token을 한번에 받아 처리하는 방식을 사용하여서, 병렬처리가 가능하다는 장점을 가지고 있음.

Pre-Trained Model의 시대

Attention is all you need (The Transformer)

- Multi-Head Attention이라는 Module과 Feed Forward 변환을 ResNet에서 사용된 Shortcut으로 묶어놓은 기본 Module을 사용.
- 아래 그림에서 보듯 여러개의 해당 Module들을 N번 쌓아둔 Encoder와 해당 정보를 받아 비슷한 구조의 Module들을 또 여러겹 쌓아둔 Decoder로 구성되어 있음

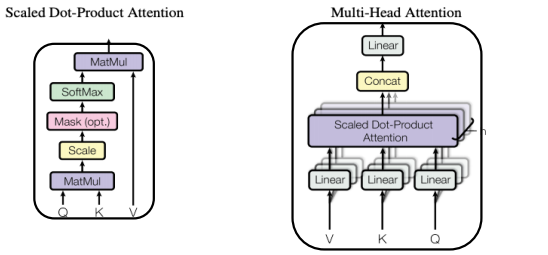


Figure 2: (left) Scaled Dot-Product Attention. (right) Multi-Head Attention consists of several attention layers running in parallel.

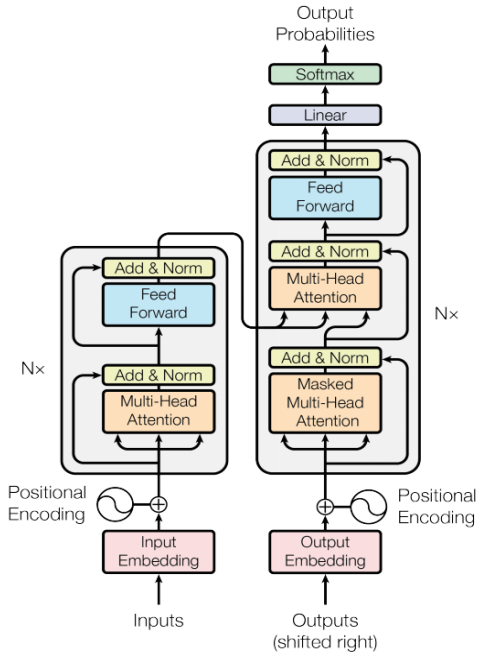


Figure 1: The Transformer - model architecture.

출처 : Attention is all you need, Ashish Vaswani et al., 2017 (https://arxiv.org/abs/1706.03762)

Pre-Trained Model의 시대

Attention is all you need (The Transformer)

- Multi-Head Attention Module은 이름에서 볼 수 있듯이 여러 개의 세부 Module Scaled Dot-Product Attention의 결과물을 모아서 하나의 Layer를 통과시켜 정보를 전달. 그래서 Multi-Head Attention이해에 앞서 Scaled Dot-Product Attention에 대한 Module의 이해가 선행되어야 함.

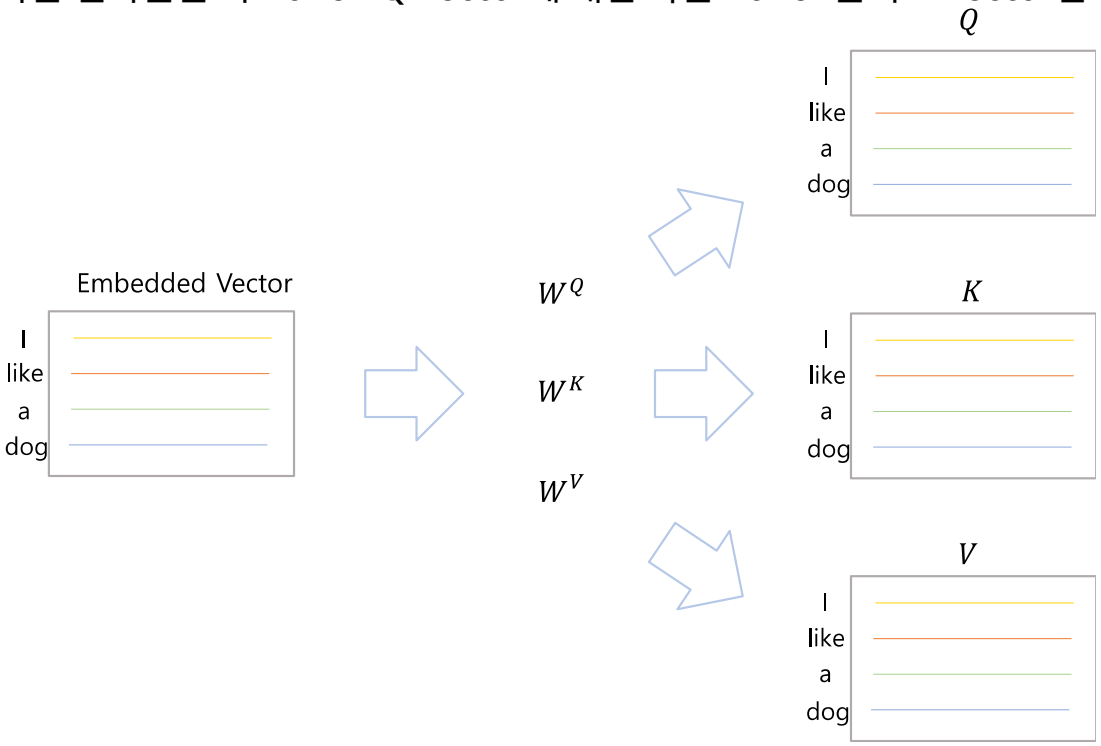
$$Attention(Q, K, V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

- Module의 Input은 각각 Query(Q), Key(K), Value(V)라고 표현.
- Tokenized된 문장에 대해 기본 Embedding Vector와 Position Encoding Vector를 합해 표현하고, 이에 Q, K, V에 대한 각각의 행렬을 곱하여 Q, K, V 행렬로 변환시킴. 같은 문장이지만 다른 표현 방법으로 표현된 행렬들이라고 생각할 수 있음

Pre-Trained Model의 시대

Attention is all you need (The Transformer)

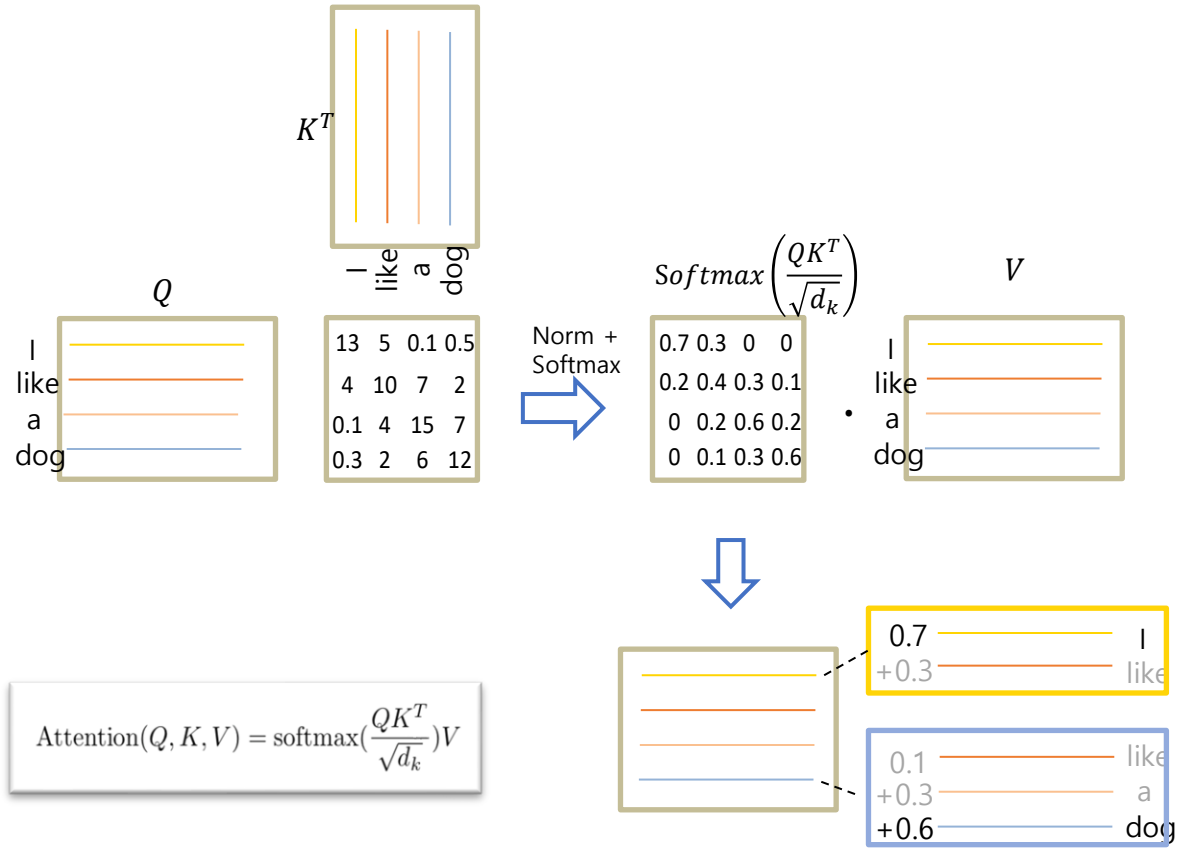
- Q와 K 행렬의 행렬곱 연산에 대한 해석. 행렬곱 연산을 한 결과 행렬의 요소들은 각 Token Vector의 Dot-Product라고 볼 수 있으며, 일반적으로 벡터 간 거리를 의미한다고 볼 수 있음. 그리고 나누어주는 벡터의 크기 (d_k) 값은 벡터 크기에 대한 영향력을 줄여주는 요소라고 보면 되고, Row-By-Softmax를 취해줌으로서 나온 결과물은 각 Token Q Vector에 대한 다른 Token들의 K Vector간의 유사도를 나타내는 상대점수라고 해석할 수 있음



Pre-Trained Model의 시대

Attention is all you need (The Transformer)

- "I"라는 Token Q Vector에 대해서 "I" Token의 K Vector는 0.7 정도의 유사성을 "like"라는 Token에 대해서는 0.3 정도의 유사성을 가지고 있다고 볼 수 있음.
- 그렇게 만들어진 행렬과 V 행렬을 다시 한번 행렬곱 연산을 함. 이 때, V 행렬에 대해 Row측면에서 바라보면 각 Token Vector 입장에서 볼 수 있게 됨.
- 해당 연산은 Token Vector에 대한 상대 점수 Weighted Average라고 해석할 수 있음.
- 우측 그림을 보면 위 설명했던 "I" vs ["I", "like"]간의 상대 점수 0.7, 0.3만큼의 Weighted Vector로 최종 결과가 나타나는 것을 확인 할 수 있음



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

Pre-Trained Model의 시대

Attention is all you need (The Transformer)

- 문장 벡터가 Scaled Dot-Product Attention Module을 통과한다는 것의 의미는 문장 내 다른 유사 Token의 정보도 적절히 포함하고 있는 Weighted Vector로 변환하는 과정이라고 해석할 수 있음.
- Word2vec의 아이디어인 목적 Token의 표현을 주변 Token의 정보의 합산으로 표현하는 것과 유사하다고도 볼 수 있음.
- 아래 그림과 식을 보면 여러 개의 Scaled Dot-Product Attention Module를 보면 서로 다른 h개 만큼 만들어지는 Token 간 Weighted Average 결과물을 붙여서(Concatenate) 다양성을 챙기고, 그 다양한 정보를 Linear Layer 한 번 더 통과시켜 정보를 다시 정제하여 최종 Output을 만들어 냄

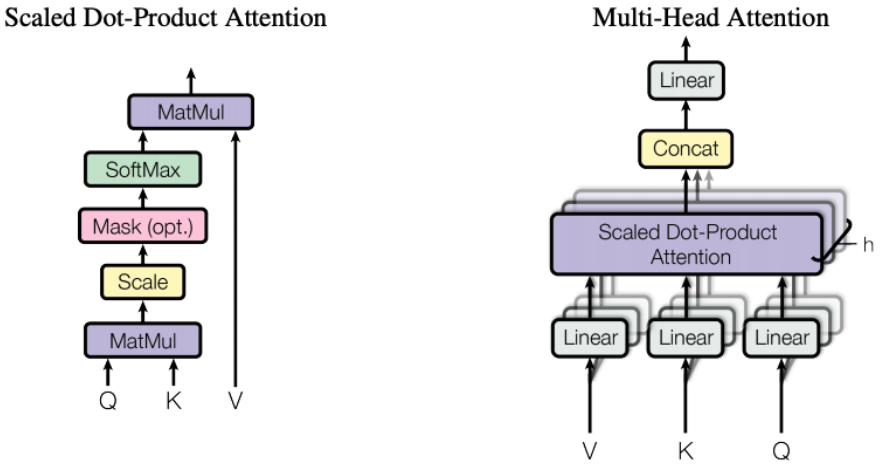


Figure 2: (left) Scaled Dot-Product Attention. (right) Multi-Head Attention consists of several attention layers running in parallel.

$$MultiHead(Q,K,V) = Concat(head_1,\dots,head_h)W_O$$
$$head_i = Attention(QW_i^Q, KW_i^K, VW_i^V)$$
$$W_i^Q \in R^{d_{model}\times d_k}, W_i^K \in R^{d_{model}\times d_k}, W_i^V \in R^{d_{model}\times d_k}, W_O \in R^{hd_v\times d_{model}}$$

출처 : Attention is all you need (https://arxiv.org/abs/1706.03762)

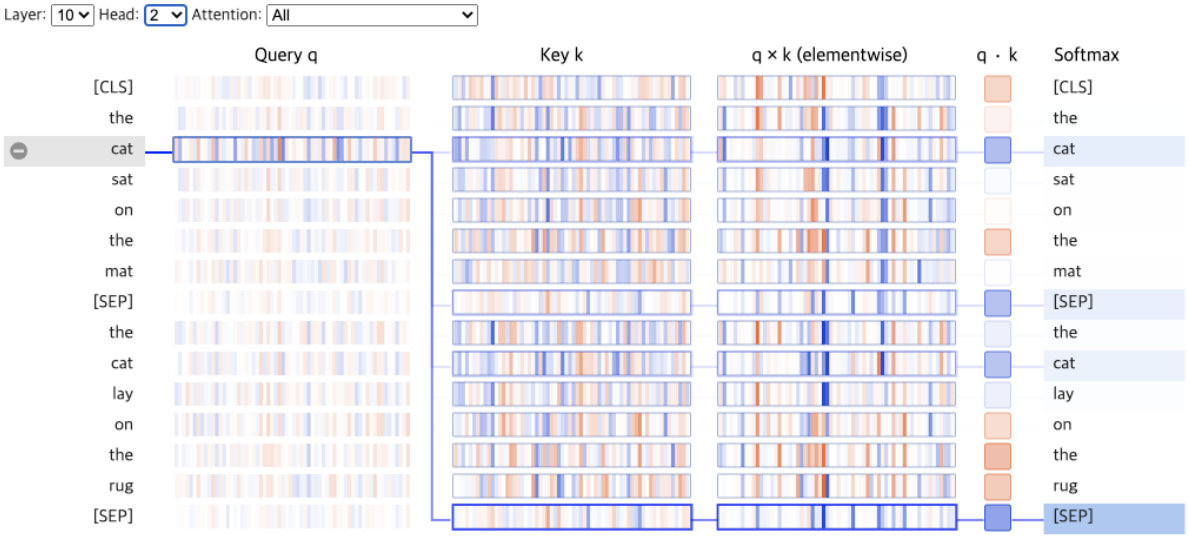
Pre-Trained Model의 시대

Attention is all you need (The Transformer)

- Multi-Head Attention을 통과하는 과정은 "다양한 Token별 관계에 대한 Weight Average 결과를 종합"하는 것이라고 볼 수 있음.
- The Transformer의 Encoder는 이와 같은 Module을 여러겹 이어 두어 정보의 재조합을 여러번 작업하게끔 하였고, 끝에 나오는 Output을 Decoder에게 넘겨주는 방식으로 모델이 구성되어 있음
- 해당 논문의 큰 성과는 기존 기계번역 모델들에서 사용하는 RNN류의 Module을 벗어났다는 점과 Attention 정보만 가지고도 좋은 성능을 낼 수 있다는 점이라고 볼 수 있음

*참고

- 모델에 대한 시각화가 잘 되어 있는 사이트 (https://github.com/jessevig/bertviz)
- github의 Neuron View (Colab)를 통해 학습된 Transformer 모델의 핵심인 Scaled Dot-product Attention Module을 더 잘 이해할 수 있음.
- 해당 사이트를 통해 Token 별 Q, K 표현의 관계가 어떻게 변하는지, 추가로 Layer와 Head 별 차이도 눈으로 확인



출처 : https://github.com/jessevig/bertviz

Pre-Trained Model의 시대

Pre-training of Deep Bidirectional Transformers for Language Understanding (BERT)

- NLP에서의 Pre-trained 모델을 학습하고, 이를 Fine-tuning하는 모델의 가능성과 동시에 높은 성능을 보여줌.
- 논문 발표할 시의 BERT는 11개 Task에서 state-of-the-art 의 성능을 달성하였고, 기존 성능 대비 높은 향상을 보여주면서 많은 관심을 받게 됨

- 모델 학습
  - 일반 문서에서 Feature를 학습하는 Unsupervised Pre-training 과정과 Pre-trained 모델을 가지고 각각의 특정 Task에서 한번 더 학습을 시키는 Fine-tuning 과정을 거침.
  - Unsupervised Pre-training 과정에서는 일반 문장들만 Input으로 사용하고 특정한 답이 없기 때문에 Task과 정답 Label을 만들어줘야 함.
  - BERT는 문장에서 랜덤으로 몇개의 Token을 가리고 주변 문맥으로 해당 Token을 맞추는 Task인 Masked LanguageModel (MLM)과 연이은 문장 Pair인지 의도적으로 랜덤으로 매칭시킨 Pair 문장인지를 구분하는 Task인 Next Sentence Prediction (NSP) 두 가지에 대한 학습을 진행시킴.
  - Pre-training 과정에서는 단어간 관계와 문장단위의 이해를 중점으로 학습을 시키고, Fine-tuning에서는 주어진 다양한 Task에 대한 새로운 데이터를 다시 Input으로 받아서 학습시켜 Task별 최종 모델을 구성

Pre-Trained Model의 시대

Pre-training of Deep Bidirectional Transformers for Language Understanding (BERT)

- 모델 구조
  - 기본적으로는 BERT는 The Transformer를 사용.
  - 하지만 번역을 위한 모델이었던 Transformer와는 달리 BERT는 일반적인 Language Model이기 때문에 Transformer의 Encoder 부분만 떼어와서 모델을 구성.
  - 모델 구조적으로는 전에 설명한 The Transformer의 내용과 다르지 않음
  - Transformer의 Token Embedding과 Position Embedding은 그대로 사용하고, 두 개의 문장을 이어서 Input을 받기 때문에 문장을 구분할 수 있는 Segment Embedding을 추가함.
  - 그리고 한가지 [CLS], [SEP] 라는 특별한 Token을 추가하는데, [SEP]는 문장의 끝을 알리는 문장 구분의 목적으로, [CLS]는 문장의 시작을 알리는 용도로 추가. 그리고 [CLS] Token은 학습하는 동안 문장 전체의 정보를 담는 목적으로 사용.

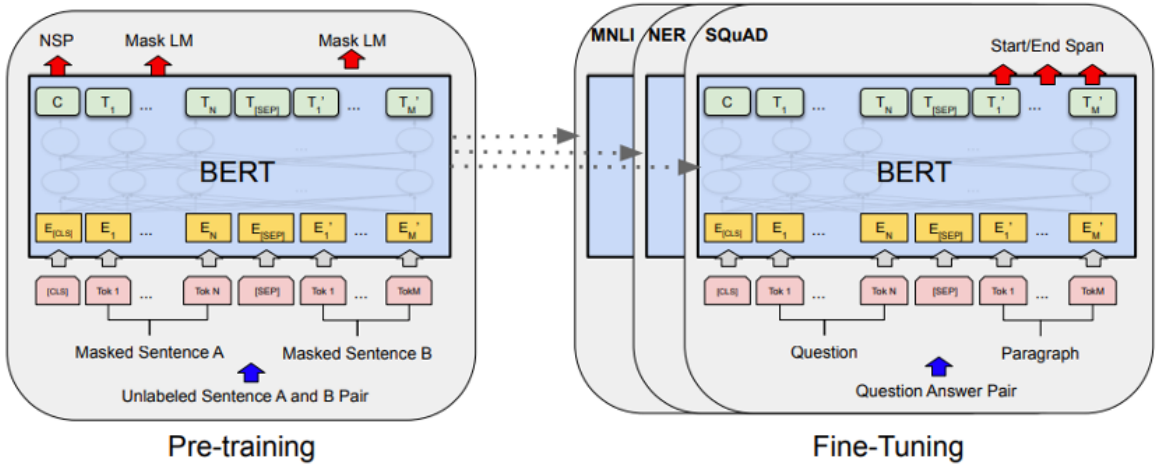


Figure 1: Overall pre-training and fine-tuning procedures for BERT. Apart from output layers, the same architectures are used in both pre-training and fine-tuning. The same pre-trained model parameters are used to initialize models for different down-stream tasks. During fine-tuning, all parameters are fine-tuned. [CLS] is a special symbol added in front of every input example, and [SEP] is a special separator token (e.g. separating questions/answers).

출처 : BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding (https://arxiv.org/abs/1810.04805)

Pre-Trained Model의 시대

Pre-training of Deep Bidirectional Transformers for Language Understanding (BERT)

- 모델 구조
  - Output은 Token 개수만큼의 벡터들로 구성되어 있고, 이 값들은 각 Token들마다 해당 문장에서 연관성 있는 Token들끼리의 정보를 잘 혼합해 놓은 것.
  - 그리고 연구진들은 각 Token의 Output 으로는 Mask가 되어 있는 Token을 예측하는 MLM Task를 해결하는데 사용하였고, [CLS] Token의 Output 값을 이용해서는 Sentence끼리의 관계를 맞추는 NSP Task를 해결하는데 사용.

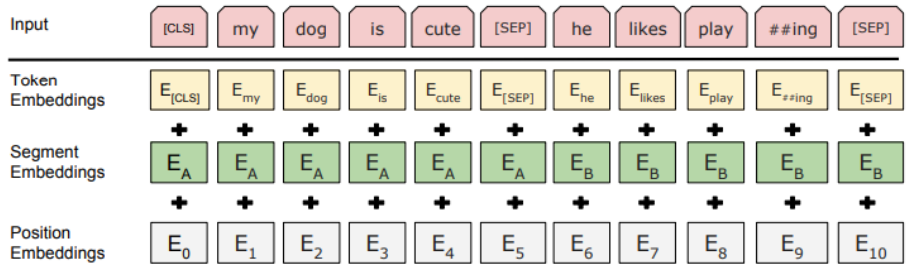


Figure 2: BERT input representation. The input embeddings are the sum of the token embeddings, the segmentation embeddings and the position embeddings.

BERT의 input 처리

출처 : BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding (https://arxiv.org/abs/1810.04805)

Pre-Trained Model의 시대

Pre-training of Deep Bidirectional Transformers for Language Understanding (BERT)

- 앞서 설명한 모델과 데이터를 통해 Pre-Training이 된 모델의 결과물은 새로운 형태의 Embedding이라고 볼 수 있음.
- 서로 다른 Task마다 디테일한 구성은 다르지만 Pre-Trained Model Output들에 간단한 FC만 추가하여 조금 더 학습시키는 Fine-tuning을 거치면 각 Task마다 좋은 성능을 내는 최종 모델을 얻을 수 있었기 때문.
- 이런 점이 NLP에서 자주 사용하였던 Pre-Trained Embedding과 유사하였지만, 기존의 word2vec 형태의 Fixed Embedding과는 다르게 BERT는 문장의 문맥에 따라 Embedding Vector가 달라지는 특성 때문에 Contextual Embedding이라고 표현하고 있음.
- 이 덕분에 서로 다른 문맥에서 사용되는 동음이의어 문제도 해결할 수 있었음

Pre-Trained Model의 시대

Pre-training of Deep Bidirectional Transformers for Language Understanding (BERT)

- BERT의 연구 성과
  - BERT 모델은 모델의 성능이 SOTA를 여러 개 갱신했다는 점과 몇개의 Task에서는 인간의 능력을 넘어섰다는 점에서도 큰 성과를 냈다고 볼 수 있음
  - 하지만 BERT의 진짜 성과는 NLP 영역에서의 Pre-Training & Fine-tuning 형태의 학습 방법을 성공적으로 보여줬다는 점.
  - Computer Vision 영역에서만 적용되는 것이 아니라 NLP에서도 잘 적용된다는 모습을 보여줬을 뿐 아니라 이 이후 활발한 연구를 이끌어 냄.
  - BERT 이후에 BERT보다 당연히 좋은 성능의 모델들은 많이 나왔지만, 대부분의 모델은 BERT의 핵심적인 요소들을 품고 있음.
  - BERT는 최신 NLP 연구를 위해서는 반드시 알고 넘어가야 하는 요소.