딥러닝을 활용한 자연어 처리

인공신경망 딥러닝

RNN, LSTM, Transformer

https://github.com/hjk7902/nlp

이 슬라이드에서 사용한 서체 :
- Open Sans(https://ko.cooltext.com/Download-Font-Open+Sans)
- KoPubWorld돋움체(http://www.kopus.org/biz/electronic/font.aspx)

과정 안내

딥러닝을 활용한 자연어 처리

딥러닝을 활용한 자연어 처리

- 자연어 전처리와 워드 임베딩에 대해 이해하고 구현할 수 있습니다.
- RNN, LSTM 등 자연어 처리를 위한 인공신경망 구조를 이해하고 구현할 수 있습니다.
- 트랜스포머의 구조를 이해하고 이를 자연어처리에 활용할 수 있습니다.

주요 내용의 흐름

인공지능 개요 → 워드 임베딩 → RNN, LSTM → Transformer

선수지식

- 파이썬, 넘파이(또는 판다스)
- 텐서플로우(케라스), 인공신경망 이론, 딥러닝 구현

1장. 인공지능과 자연어처리

딥러닝을 활용한 자연어 처리

1절. 인공지능

1장. 인공지능과 자연어처리

인공지능의 시작

John McCarthy, 1956, Dartmouth Conference

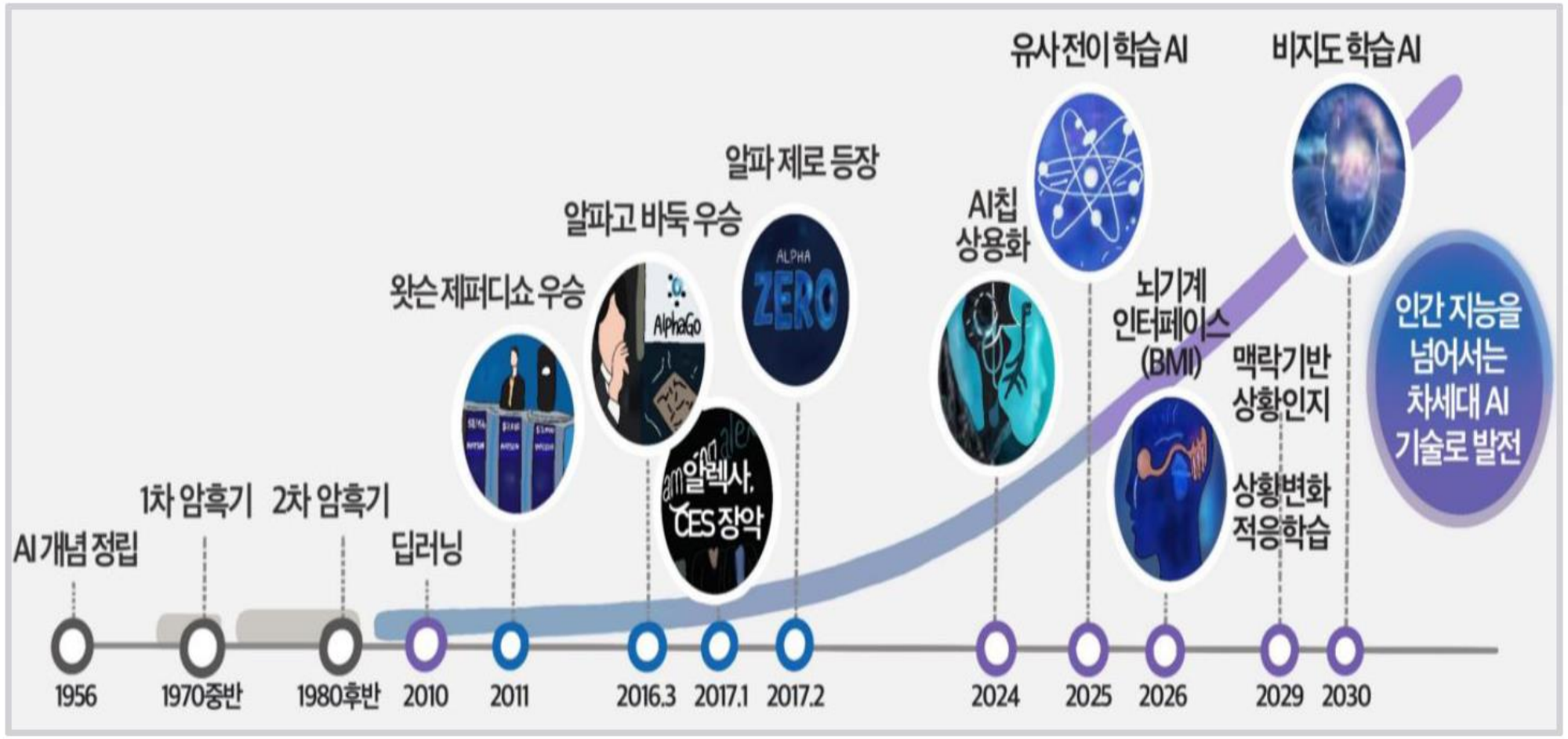
일반인으로 심사 위원 구성 ⇨ 블라인드 상태에서 컴퓨터와 인간이 텍스트 메시지로 대화 ⇨ 컴퓨터를 인간으로 판정한 비율

30% 이상일 경우

인간처럼 사고할 수 있는 컴퓨터로 평가

Alan Turing, Turing Test

인공지능의 역사

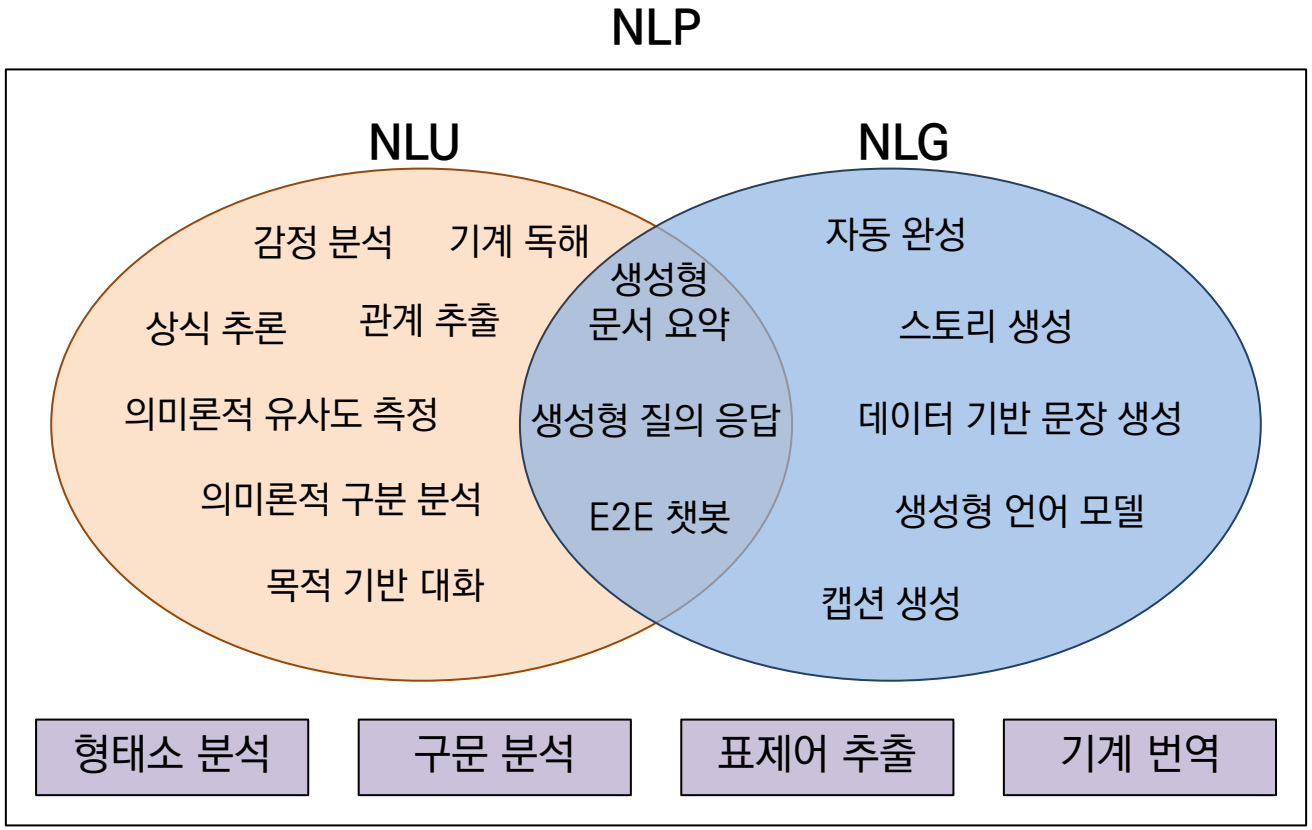

유사전이학습AI
비지도학습AI
알파 제로 등장
알파고 바둑 우승
AI칩 상용화
왓슨 제퍼디쇼 우승
뇌기계 인터페이스 (BMI)
맥락기반 상황인지
인간 지능을 넘어서는 차세대 AI 기술로 발전
ALPHA ZERO
AlphaGo
알렉사, CES 장악
상황변화 적응학습
AI개념 정립
1차 암흑기
2차 암흑기
딥러닝
1956
1970중반
1980후반
2010
2011
2016.3
2017.1
2017.2
2024
2025
2026
2029
2030

2절. 자연어처리

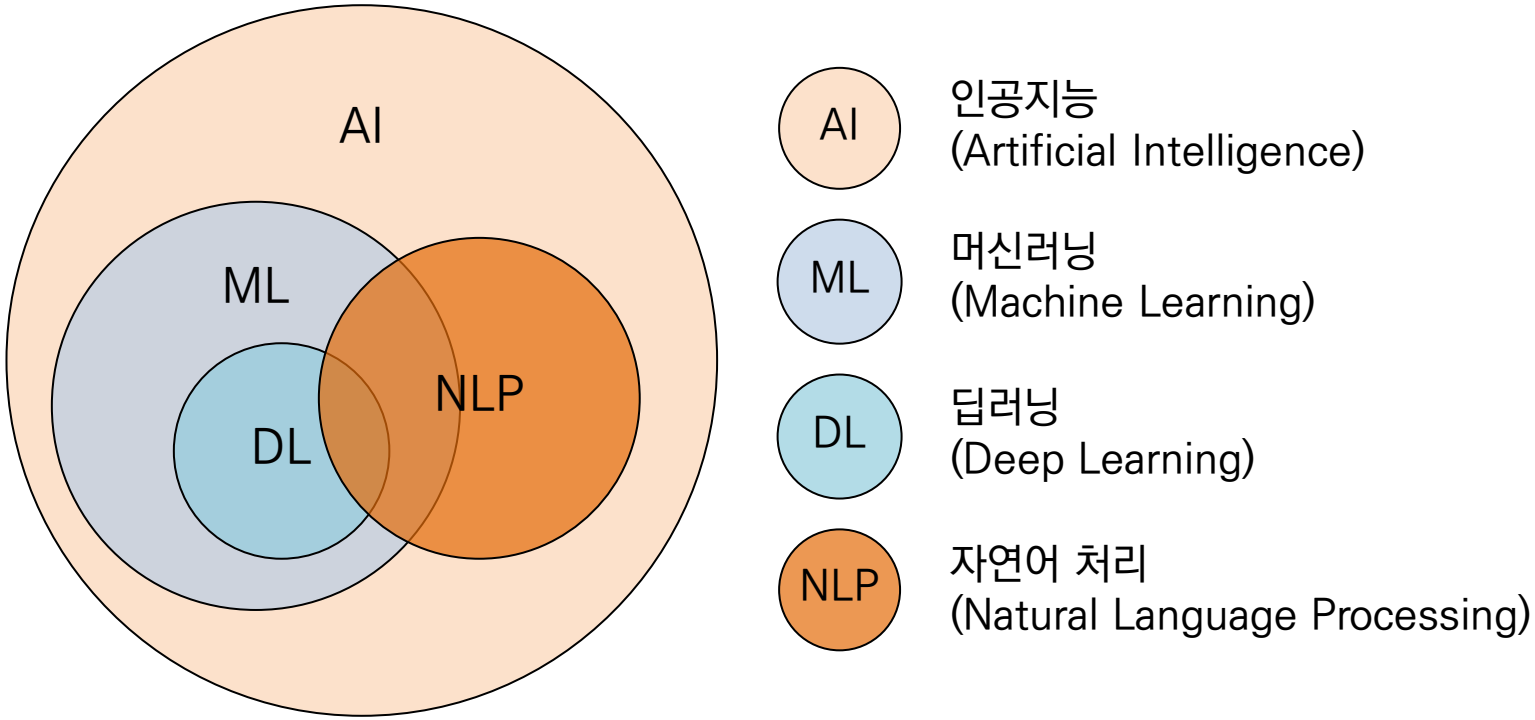
1장. 인공지능과 자연어처리

자연어와 자연어처리 분야

- 자연어는 인간이 상대방과 의사소통을 위해서 사용하는 언어(한국어, 영어, 중국어 등)
- 자연어처리(NLP; Natural Language Processing)는 사람의 자연어를 컴퓨터가 이해하고 처리할 수 있도록 하는 인공지능의 한 분야



자연어처리 역사와 영역


AI
ML
DL
NLP
AI
인공지능
(Artificial Intelligence)
ML
머신러닝
(Machine Learning)
DL
딥러닝
(Deep Learning)
NLP
자연어 처리
(Natural Language Processing)

자연어처리 활용 사례

- 인공지능 번역기
  - 딥러닝을 이용해 국가 간 언어를 번역하는 서비스.
  - 기존 번역 서비스보다 정확도가 한층 높아졌으며, 전후 문맥을 파악하여 자연스러운 번역이 가능해짐
  - 최근에는 딥러닝 음성인식 기술과 융합하여 통역 서비스에 활용
- 인공지능 챗봇
  - 챗봇(Chatbot)은 채팅(Chatting)과 로봇(Robot)이 합성어로 인공지능 기반의 커뮤니케이션 소프트웨어
    사람과의 문자 대화를 통해 질문에 알맞은 답변이나 각종 연관 정보를 제공하며, 다양한 비즈니스 도메인 분야에 활용
- 인공지능 스피커
  - 음성인식 기능과 인공지능 자연어처리 기술 등을 활용한 스피커
  - 인공지능 스피커는 사용자의 음성 명령을 스피커가 서버로 전송하고, 인공지능 플랫폼이 의미분석 및 서비스를 제공
- 인공지능 리걸 테크(Legal Tech)
  - 인공지능 자연어처리, 빅데이터 기술 등을 활용한 법률서비스
  - 기존이 판례나 법령, 변호사들의 자문 빅데이터를 활용
  - 이용자가 기본적인 정보만 입력하면 자동으로 법률 문서가 작성되고 법률 상담 서비스를 제공
- 인공지능 저널리즘(Journalism)
  - 인공지능 기술을 활용해 기사를 작성하는 로봇 저널리즘
  - 머신러닝, 딥러닝 등을 활용한 자연어처리 기술을 활용
  - 방대한 빅데이터에서 중요한 정보를 편집해서 기사화하는 데이터 저널리즘 등에 활용
  - 기사 및 정보의 사실 여부를 평가하여 가짜 뉴스 판별에도 활용함
- 인공지능 컨택서비스(Contact Service)
  - 고도화된 AI 음성 로봇을 활용한 고객 상담센터를 제공
  - 고객들의 상담 대기시간을 획기적으로 줄일 수 있으며, 고객의 문의 사항에 대한 신속한 답변과 안내 사항 등 각종 정보를 제공

3절. 자연어처리를 위한 텍스트 전처리

1장. 인공지능과 자연어처리

자연어 처리를 위한 텍스트 전처리

Corpus : 여러 문장으로 구성된 문서의 집합, 자연어처리에 사용되는 데이터

정제, 불용어 제거, 어간 추출, 토큰화 및 문서 표현 등의 작업을 수행

- 정제(Cleaning): 특수문자 등과 같은 불필요한 노이즈 텍스트를 제거, 대소문자 통일 시킴
- 불용어 제거(Stop word Elimination): 전치사, 관사 등 문장이나 문서의 특징을 표현하는데 불필요한 단어를 제거
- 토큰화(Tokenizer): 코퍼스(Corpus)에서 분리 자(Separator)를 포함하지 않는 연속적인 문자열 단위로 분리(nltk, konlpy 사용)
- 워드 임베딩(Word Embedding): 주어진 문서나 문장을 하나의 벡터로 표현
  단어들을 인덱싱하고 주어진 문서에 존재하는 단어의 빈도수를 사용하여 문서를 표현
  문서 표현 방법은 One-hot 인코딩, TF-IDF, Word2Vec 등이 있음

형태소 분석

NLTK(Natural Language Toolkit, https://www.nltk.org/)

- 인간의 언어 데이터를 파이썬 언어로 분석할 수 있도록 만들어진 패키지
- 분류, 토큰화, 형태소 분석, 품사 태깅, 구문 분석 및 의미론적 추론을 위한 텍스트 처리 라이브러리와 WordNet과 같은 50개가 넘는 말뭉치를 제공

KoNLPy(http://konlpy.org/)

- Hannanum, Kkma, Komoran, Mecab, Okt 형태소 분석기 지원
- 내부 품사 태깅 클래스들의 성능 비교: https://konlpy.org/ko/latest/morph/

임베딩

문자열을 숫자로 변환하여 벡터(Vector) 공간에 표현하는 방법

빈도 기반 : 문서에 등장하는 단어의 빈도를 계산해서 행렬로 표현하거나 가중치를 부여

- TF(Term Frequency), DTM(Document Term Matrix), TF-IDF(Term Frequency-Inverse Document Frequency)

토픽 기반: 문서에 잠재된 주제(Latent Topic)를 추론(Inference)하기 위한 임베딩

- LSA(Latent Semantic Analysis, 잠재 의미분석)

예측(Prediction) 기반: 다음 단어, 주변 단어, 마스킹 된 단어의 예측 등을 위한 임베딩

- Word2Vec, FastText, BERT, ELMo, GPT

워드 임베딩 vs. 언어 모델

워드 임베딩

단어를 기반으로 임베딩을 수행하는 것

워드 임베딩은 서로 다른 문맥의 동음이의어가 같게 임베딩 되는 단점

언어 모델

문맥을 고려하여 문장(Sentence)을 기반으로 임베딩을 수행하는 것

4절. 워드 임베딩

1장. 인공지능과 자연어처리

워드 임베딩

단어를 숫자에 대응

- 의미론적, 구문론적 컨텍스트 또는 단어를 인지하는 데 도움
- 기사, 블로그 등에서 다른 단어와 얼마나 유사하거나 유사하지 않은지 이해하는 데 도움

인기 있는 워드 임베딩

- 이진 인코딩(Binary Encoding) 또는 원-핫 인코딩
- TF 인코딩(TF Encoding)
- TDM(Term Document Matrix)
- TF-IDF 인코딩(TF-IDF Encoding)
- 잠재 의미분석 인코딩(Latent Semantic Analysis Encoding)
- Word2Vec 임베딩(Word2Vec Embedding)

Term Frequency

문서에서 용어가 나타나는 총 회수로 정의

```
1  text = "John likes to watch movies. Mary likes movies too.\
2   Mary also likes to watch football games."
```

```
1  words = text.replace('.', '').split()
2  print(words)
```

문서에서 구두점을 제외하고 공백으로 분리

```
['John', 'likes', 'to', 'watch', 'movies', 'Mary', 'likes',
'movies', 'too.', 'Mary', 'also', 'likes', 'to', 'watch',
'football', 'games']
```

Term Frequency


```
1  import numpy as np
2  word_count = np.unique(words, return_counts=True)
3  print(word_count)
```

리스트에서 유일한 값의 개수를 셈

```
(array(['John', 'Mary', 'also', 'football', 'games', 'likes',
'movies', 'to', 'too.', 'watch'], dtype='<U8'), array([1, 2, 1, 1,
1, 3, 2, 2, 1, 2]))
```

```
1  word_to_cnt = {}
2  for word, cnt in zip (*word_count):
3      word_to_cnt[word] = cnt
4  print(word_to_cnt)
```

단어-개수 딕셔너리를 만듬

```
{'John': 1, 'Mary': 2, 'also': 1, 'football': 1, 'games': 1,
'likes': 3, 'movies': 2, 'to': 2, 'too.': 1, 'watch': 2}
```

Term Document Matrix

문서별로 단어의 빈도수를 계산해서 행렬로 만들어 놓은 것

예제 텍스트가 두 개의 문장이라면 문장마다 단어의 빈도수를 계산해서 표현할 수 있음

```
corpus = [
    "John likes to watch movies. Mary likes movies too.",
    "Mary also likes to watch football games."
]
```

Term Document Matrix

문서별로 단어의 빈도수를 계산해서 행렬로 만들어 놓은 것

```
1  from sklearn.feature_extraction.text import CountVectorizer
2  vector = CountVectorizer()
3  tdm_array = vector.fit_transform(corpus).toarray()
4  tf_dic = vector.vocabulary_
5  print(tdm_array)
6  print(tf_dic)
```

싸이킷런을 이용하면 TDM 문서를 쉽게 만들 수 있음

```
[[0 0 0 1 2 1 2 1 1 1]
 [1 1 1 0 1 1 0 1 0 1]]
{'john': 3, 'likes': 4, 'to': 7, 'watch': 9, 'movies': 6, 'mary':
5, 'too': 8, 'also': 0, 'football': 1, 'games': 2}
```

Term Document Matrix

TDM 행렬을 데이터프레임으로 만듬

```
1  import pandas as pd
2  tf_dic_sorted = dict(sorted(tf_dic.items(),
                               key=lambda item: item[1]))
3  tdm = pd.DataFrame(tdm_array, columns=tf_dic_sorted.keys())
4  print(tdm)
```

	also	football	games	john	likes	mary	movies	to	too	watch
0	0	0	0	1	2	1	2	1	1	1
1	1	1	1	0	1	1	0	1	0	1

TF-IDF

- Term Frequency-Inverse Document Frequency
- 문서별로 단어의 빈도수를 계산해서 행렬로 만들어 놓은 것
- IDF는 전체 문서에서 용어가 얼마나 자주 발생하는지 정의
- 전체 문서 세트에서 발생하는 용어의 가중치 균형을 맞추는 데 사용

TF-IDF

$$TF(w) = \text{문서에서 } w \text{의 빈도} / \text{총 단어수}$$

$$IDF(w) = \log(\text{용어를 포함하는 문서 수}) / (\text{총 문서 수} + 1)$$

$$TF-IDF(w) = TF(w) * IDF(w)$$

- TF-IDF 점수가 클수록 해당 용어가 문서에서 더 관련성이 높음
- 단어가 기사에서 등장한 횟수에 비례/이 단어가 전체 영역에서 등장한 횟수에 반비례

TF-IDF


```
from sklearn.feature_extraction.text import TfidfVectorizer

tfidf_vec = TfidfVectorizer()
tfidf_array = tfidf_vec.fit_transform(corpus).toarray()
tfidf_dic = tfidf_vec.vocabulary_
tfidf_dic_sorted = dict(sorted(tfidf_dic.items(),
                               key=lambda item: item[1]))
tfidf_dtm = pd.DataFrame(tfidf_array,
                         columns=tfidf_dic_sorted.keys())
print(tfidf_dtm)
```

	also	football	games	john	likes	mary	movies	to	too	watch
0	0.000000	0.000000	0.000000	0.323699	0.460629	0.230315	0.647398	0.230315	0.323699	0.230315
1	0.446101	0.446101	0.446101	0.000000	0.317404	0.317404	0.000000	0.317404	0.000000	0.317404

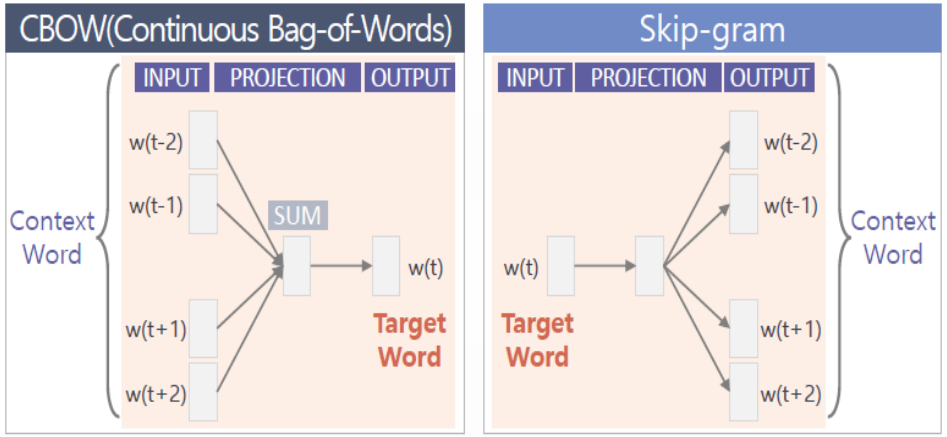
Word2Vec

워드 임베딩 모델을 생성하는 데 사용되는 얕은 2계층 신경망

많은 어휘 기반 데이터 세트를 입력으로 수집하고 사전의 각 단어가 고유한 벡터에 대응되도록 벡터 공간을 출력하는 방식으로 작동

CBOW(Continuous Bag of Words)와 Skip-Gram의 두 가지 모델이 있음

- CBOW는 주변 단어를 임베딩하여 중심 단어를 예측
- Skip-gram은 중심 단어를 임베딩하여 주변 단어를 예측



윈도우와 슬라이딩 윈도우

윈도우: 중심 단어(Target Word)를 기준으로 해서 참조하고자 하는 주변 단어(Context Word)의 범위

- 만일 윈도우 크기가 2인 경우 “quality is more import than quantity”에서
- 중심 단어가 ‘quality’인 경우 주변 단어는 ‘is’와 ‘more’이며, 중심 단어가 ‘more’인 경우 앞의 두 단어 ‘quality’와 ‘is’, 뒤의 두 단어 ‘important’와 ‘than’ 이렇게 4개 단어가 주변 단어 됨

슬라이딩 윈도우(Sliding Window)는 전체 학습 문장에 대해 윈도우를 이동하며 학습하기 위해 중심 단어와 주변 단어 데이터셋을 추출하는 과정

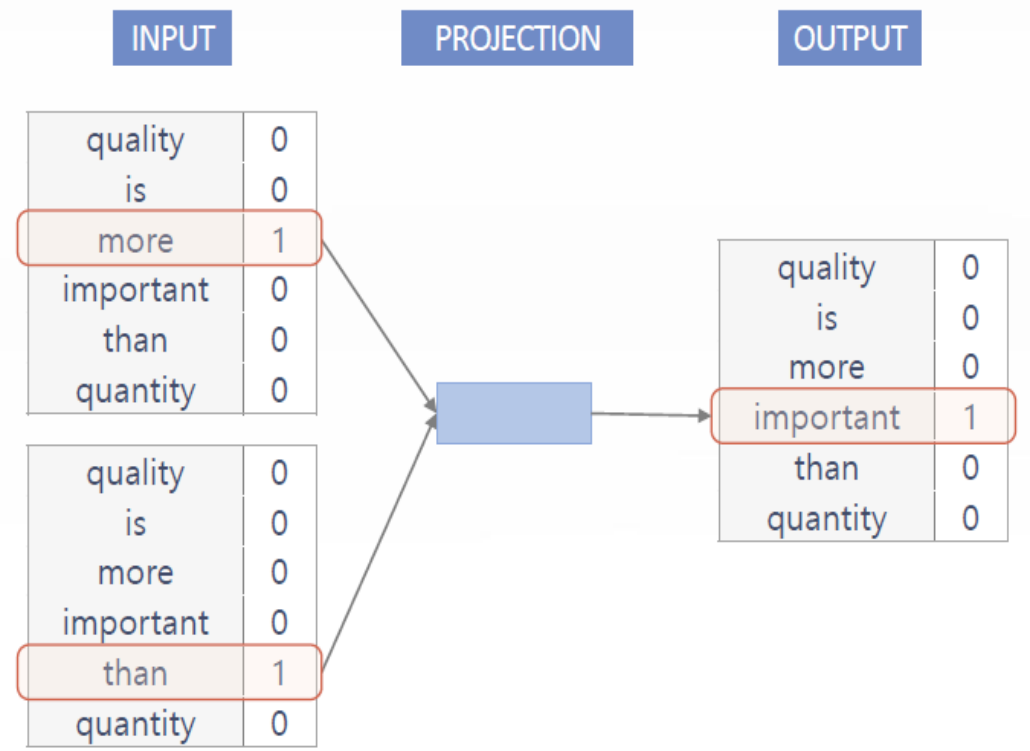
1	quality	is	more	important	than	quantity
2	quality	is	more	important	than	quantity
3	quality	is	more	important	than	quantity
4	quality	is	more	important	than	quantity
5	quality	is	more	important	than	quantity
6	quality	is	more	important	than	quantity

Sliding	Target Word	Context Word
1	[1, 0, 0, 0, 0, 0]	[0, 1, 0, 0, 0, 0], [0, 0, 1, 0, 0, 0]
2	[0, 1, 0, 0, 0, 0]	[1, 0, 0, 0, 0, 0], [0, 0, 1, 0, 0, 0], [0, 0, 0, 1, 0, 0]
3	[0, 0, 1, 0, 0, 0]	[1, 0, 0, 0, 0, 0], [0, 1, 0, 0, 0, 0], [0, 0, 0, 1, 0, 0], [0, 0, 0, 0, 1, 0]
4	[0, 0, 0, 1, 0, 0]	[0, 1, 0, 0, 0, 0], [0, 0, 1, 0, 0, 0], [0, 0, 0, 0, 1, 0], [0, 0, 0, 0, 0, 1]
5	[0, 0, 0, 0, 1, 0]	[0, 0, 1, 0, 0, 0], [0, 0, 0, 1, 0, 0], [0, 0, 0, 0, 0, 1]
6	[1, 0, 0, 0, 0, 1]	[0, 0, 0, 0, 1, 0], [0, 0, 0, 1, 0, 0]

CBOW(Continuous Bag-of-Words)

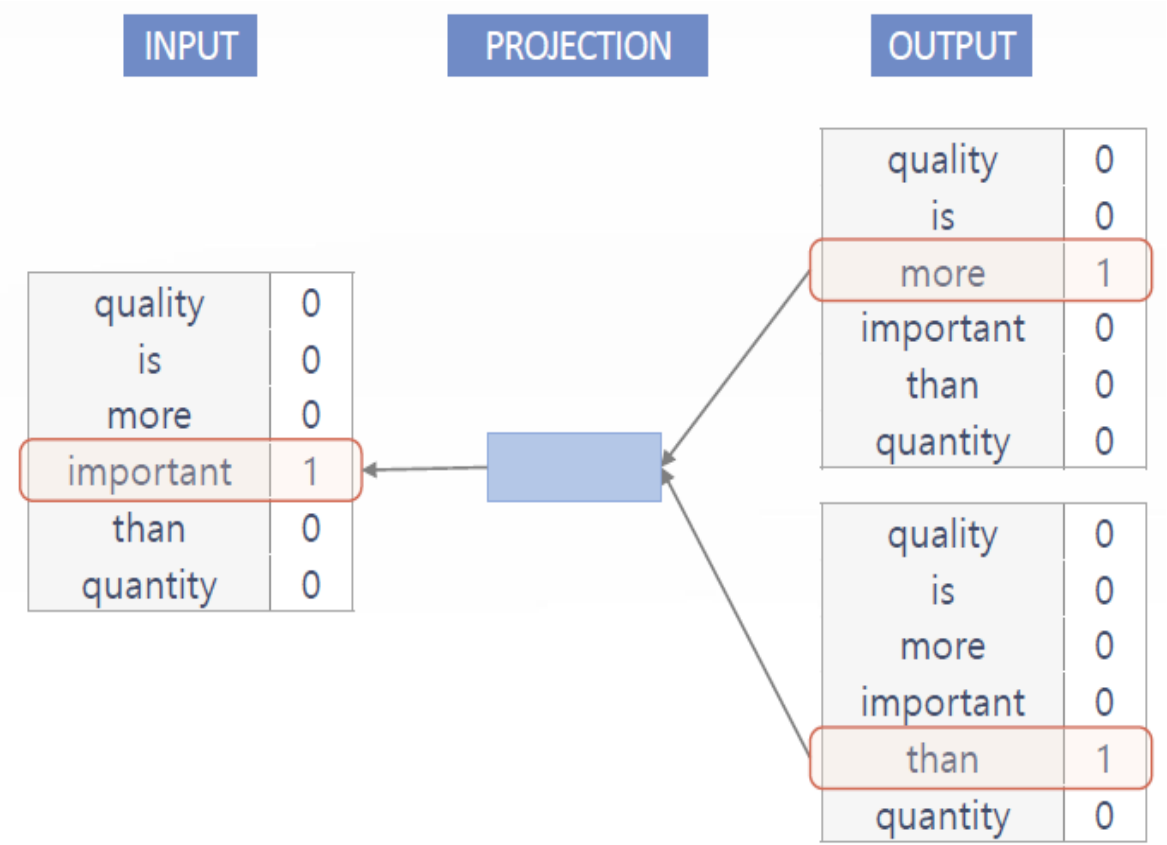
주변 단어(Context Word)를 임베딩하여 중심 단어(Target Word)를 예측

- CBOW는 소규모 데이터베이스 작업에 적합한 모델
- RAM 메모리 요구사항도 필요하지 않음



Skip-gram

중심 단어(Target Word)를 임베딩 하여 주변 단어(Context Word)를 예측



CBOW vs. Skip-gram

두 모델은 대칭을 보여주지만, 아키텍처와 성능 면에서 다름

- CBOW 모델은 주변 단어를 사용하여 중심 단어를 예측

- CBOW 모델은 더 빠름
- 더 적은 RAM이 필요하며 더 작은 데이터베이스에 적합
- 자주 사용되지 않는 단어의 처리를 보장하지 않음

- Skip-Gram 모델은 단어의 맥락적 유사성에 의존하여 특정 단어를 둘러싼 단어를 예측

- Skip-Gram 모델은 빈도가 낮은 단어에 대해 더 정확
- 더 큰 데이터베이스에 적합하며 작동하려면 더 많은 RAM이 필요

CBOW, Skip-gram 등의 Word2Vec 적용을 위한 파이썬 라이브러리

gensim.models.Word2Vec(data, sg, size, window, min_count)

- data : 리스트 형태의 데이터입니다.
- sg : 0이면 CBOW, 1이면 Skip-gram방식을 사용합니다.
- size : 벡터의 크기를 지정합니다.
- window : 윈도우의 크기입니다. 3이면 앞/뒤 3단어를 포함합니다.
- min_count : 사용할 단어의 최소 빈도입니다. 3이면 3회 이하 단어는 무시합니다.

```
!pip install --upgrade gensim
!pip install "scipy<1.11"
```

ImportError: cannot import name 'triu' from 'scipy.linalg'오류가 발생할 경우 scipy를 다운그래이드하세요. - 2024.07

gensim


```
1  corpus = ["John likes to watch movies. Mary likes movies too",
2            "Mary also likes to watch football games."]
3
4  word_list = []
5  for word in corpus:
6      word_list.append(word.replace('.', '').split())
7
8  from gensim.models import Word2Vec
9  model = Word2Vec(word_list, sg=0, vector_size=100,
10                   window=3, min_count=1)
11
12 print(model.wv.most_similar('likes'))
13 print(model.wv.similarity('movies', 'games'))
```

Word2Vec 모델을 이용하여 특정 단어와 가장 유사한 단어를 찾거나 두 단어의 유사도를 확인하는 예

```
[('John', 0.21617142856121063), ('also', 0.09291722625494003), ('too', 0.027057476341724396), ('football', 0.016134677454829216), ('Mary', -0.010840574279427528), ('to', -0.02775036357343197), ('movies', -0.05234673246741295), ('games', -0.059876296669244766), ('watch', -0.111670583486557)]
0.064089775
```

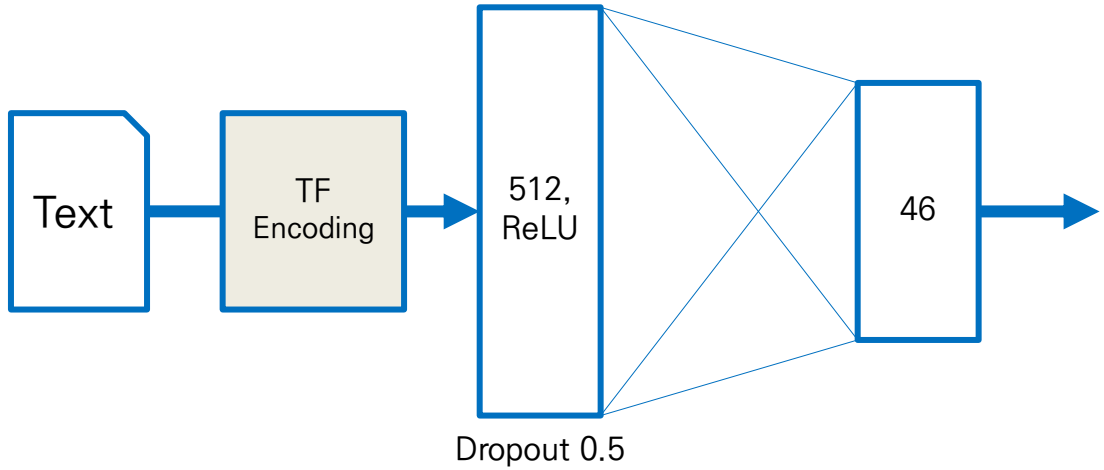
2장. 자연어처리 인공신경망

딥러닝을 활용한 자연어 처리

1절. 순환신경망

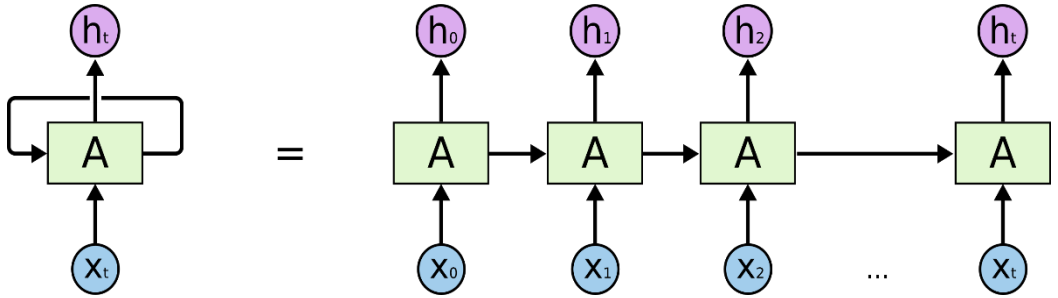
2장. 자연어처리 인공신경망

DNN을 기억하시나요?



순환신경망 개요

- 순환신경망은 1986년 데이비드 루멜하트(David Rumelhart)가 개발한 알고리즘
- 순환신경망은 시계열 데이터를 학습하는 딥러닝 기술
- 순환신경망은 기준 시점(t)과 다음 시점(t+1)에 네트워크를 연결함
- 순환신경망은 AI 번역, 음성인식, 주가 예측의 대표적 기술임



RNN 종류

One to Many : 입력이 하나이고 출력이 여러 개 생성

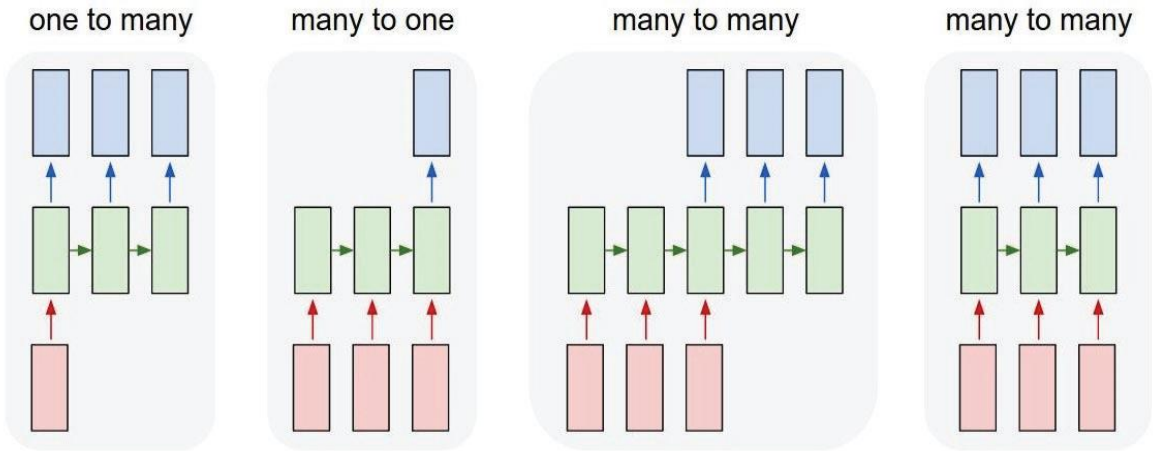
- One to Many는 영상에 캡션을 달아야 할 때 사용

Many to One : 여러 입력이며, 하나의 출력을 생성

- 영화평을 분류할 경우 Many to One이 사용

Many to Many : 여러 입력이며, 출력도 여러 개 생성

- 3번째 Many to Many 구조는 번역에 사용

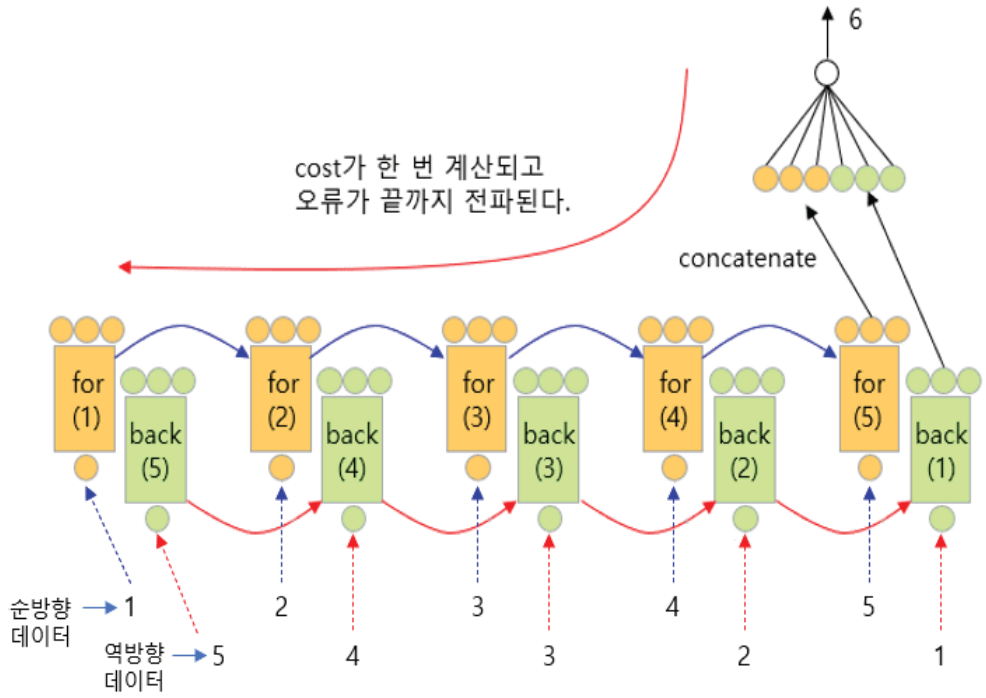


Bidirectional RNN

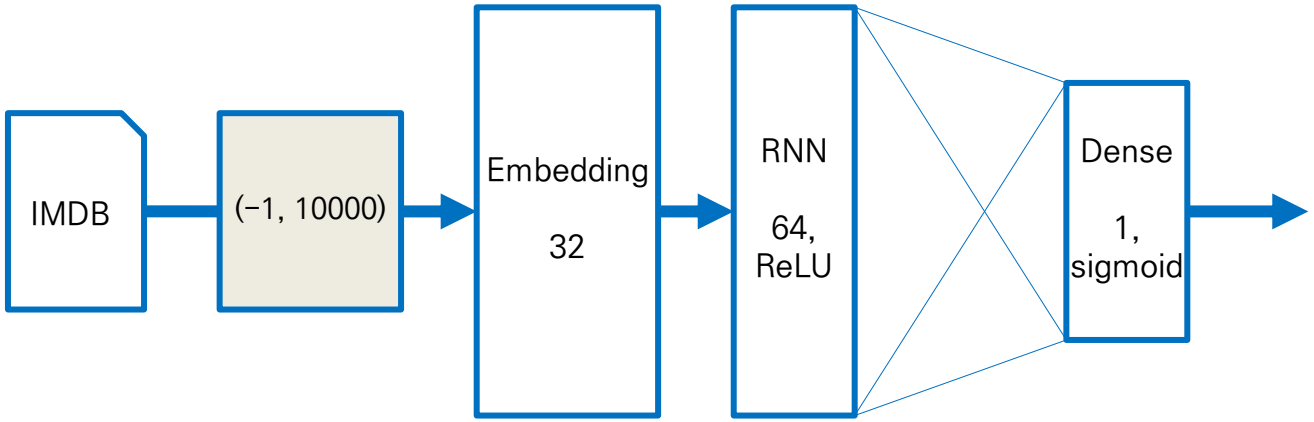
양방향 RNN은 두 개의 RNN을 하나로 묶음

하나는 순방향으로 가중치를 수정하며, 다른 하나는 역방향으로 가중치를 수정함

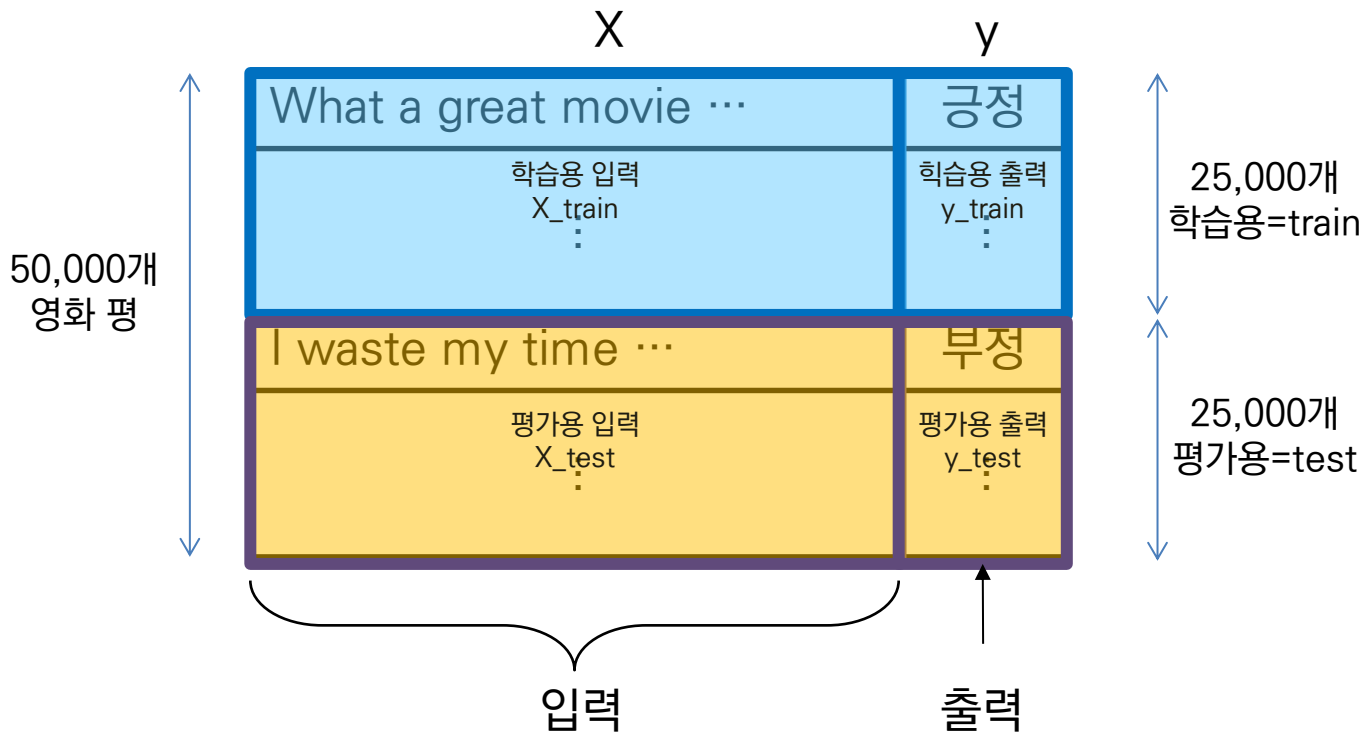
- RNN 단어가 길면 과거의 일을 잊는 것을 방지하고 또한 미래의 사실을 과거에 반영할 수 있는 구조



실습 – RNN으로 영화평 분류하기 모델 구조



실습 – RNN으로 영화평 분류하기 (imdb 데이터셋 구조)



실습 – RNN으로 영화평 분류하기 (RNN 구조 구현)


```python
from tensorflow.keras import Sequential, layers

model = Sequential([
    layers.Input(shape=(80,)),
    layers.Embedding(input_dim=10000, output_dim=32), # 10000*32개 파라미터
    layers.SimpleRNN(64), # 32x64(x에 대한 w) + 64x64(h에 대한 w) + 64(b)
    layers.Dense(2, activation='softmax') # loss='sparse_categorical_crossentropy'
    # layers.Dense(1, activation='sigmoid') # loss='binary_crossentropy'
])
model.summary() # 0.77
```

Model: "sequential"

Layer (type)	Output Shape	Param #
embedding (Embedding)	(None, 80, 32)	320,000
simple_rnn (SimpleRNN)	(None, 64)	6,208
dense (Dense)	(None, 2)	130

Total params: 326,338 (1.24 MB)
Trainable params: 326,338 (1.24 MB)
Non-trainable params: 0 (0.00 B)

parameter = (input_dim + units) x units + units

실습 – RNN으로 영화평 분류하기 (RNN 구조 구현)


```
model.compile(loss='sparse_categorical_crossentropy', optimizer='adam',metrics=['accuracy'])
```

```
from tensorflow.keras.datasets import imdb
(X_train, y_train), (X_test, y_test) = imdb.load_data(num_words=10000)
print(X_train.shape, y_train.shape, X_test.shape, y_test.shape)
# 0: 부정적인 리뷰(negative review), 1: 긍정적인 리뷰(positive review)
```

```
(25000,) (25000,) (25000,) (25000,)
```

```
from tensorflow.keras.preprocessing.sequence import pad_sequences
X_train_pad = pad_sequences(X_train, maxlen=80, truncating='post', padding='post')
X_test_pad = pad_sequences(X_test, maxlen=80, truncating='post', padding='post')
```

```
%%time
model.fit(X_train_pad, y_train, epochs=10, batch_size=200)
```

실습 – RNN으로 영화평 분류하기 (RNN 구조 구현)


```
1 model.evaluate(X_test_pad, y_test)
```

```
782/782 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━
[1.0030674934387207, 0.767520010471344]
```

```
1 import numpy as np
2 pred = model.predict(X_test_pad)
3 # pred = (pred > 0.5).astype(int)
4 pred = np.argmax(pred, axis=1)
```

```
782/782 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 8s 10ms/step
```

```
1 from sklearn.metrics import confusion_matrix
2 print(confusion_matrix(y_test, pred))
```

```
[[9826 2674]
 [3138 9362]]
```

```
1 from sklearn.metrics import accuracy_score
2 accuracy_score(y_test, pred)
```

```
0.76752
```

실습 – RNN으로 영화평 분류하기 (RNN층 추가하기)


```
from tensorflow.keras import Sequential, layers

model = Sequential([
    layers.Input(shape=(80,)),
    layers.Embedding(input_dim=10000, output_dim=32), # 10000*32개 파라미터
    layers.SimpleRNN(64, return_sequences=True), # 32x64(x에 대한 w) + 64x64(h에 대한 w) + 64(b)
    layers.SimpleRNN(128), # 64x128(x에 대한 w) + 128x128(h에 대한 w) + 128(b)
    layers.Dense(2, activation='softmax') # loss='sparse_categorical_crossentropy'
    # layers.Dense(1, activation='sigmoid') # loss='binary_crossentropy'
])
model.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Param #
embedding (Embedding)	(None, 80, 32)	320,000
simple_rnn (SimpleRNN)	(None, 80, 64)	6,208
simple_rnn_1 (SimpleRNN)	(None, 128)	24,704
dense (Dense)	(None, 2)	258

Total params: 351,170 (1.34 MB)
Trainable params: 351,170 (1.34 MB)
Non-trainable params: 0 (0.00 B)

return_sequences=True로 설정하여 출력 시퀀스를 반환하도록 설정합니다. 이는 다음 층인 SimpleRNN(128)이 시퀀스 데이터를 받을 수 있도록 합니다.

실습 – RNN으로 영화평 분류하기 (모형 수정 해보기)

앞의 모형에서 아래의 내용을 바꿔보고 그 결과를 기록해 보세요.

① 옵티마이저를 sgd로 바꿔보세요. accuracy:

② 전체 단어의 개수를 1000개로 바꿔보세요. accuracy:

③ 영화평의 길이를 200개로 바꿔보세요. accuracy:

④ pad_sequence의 truncating과 padding을 pre로 바꿔보세요. accuracy:

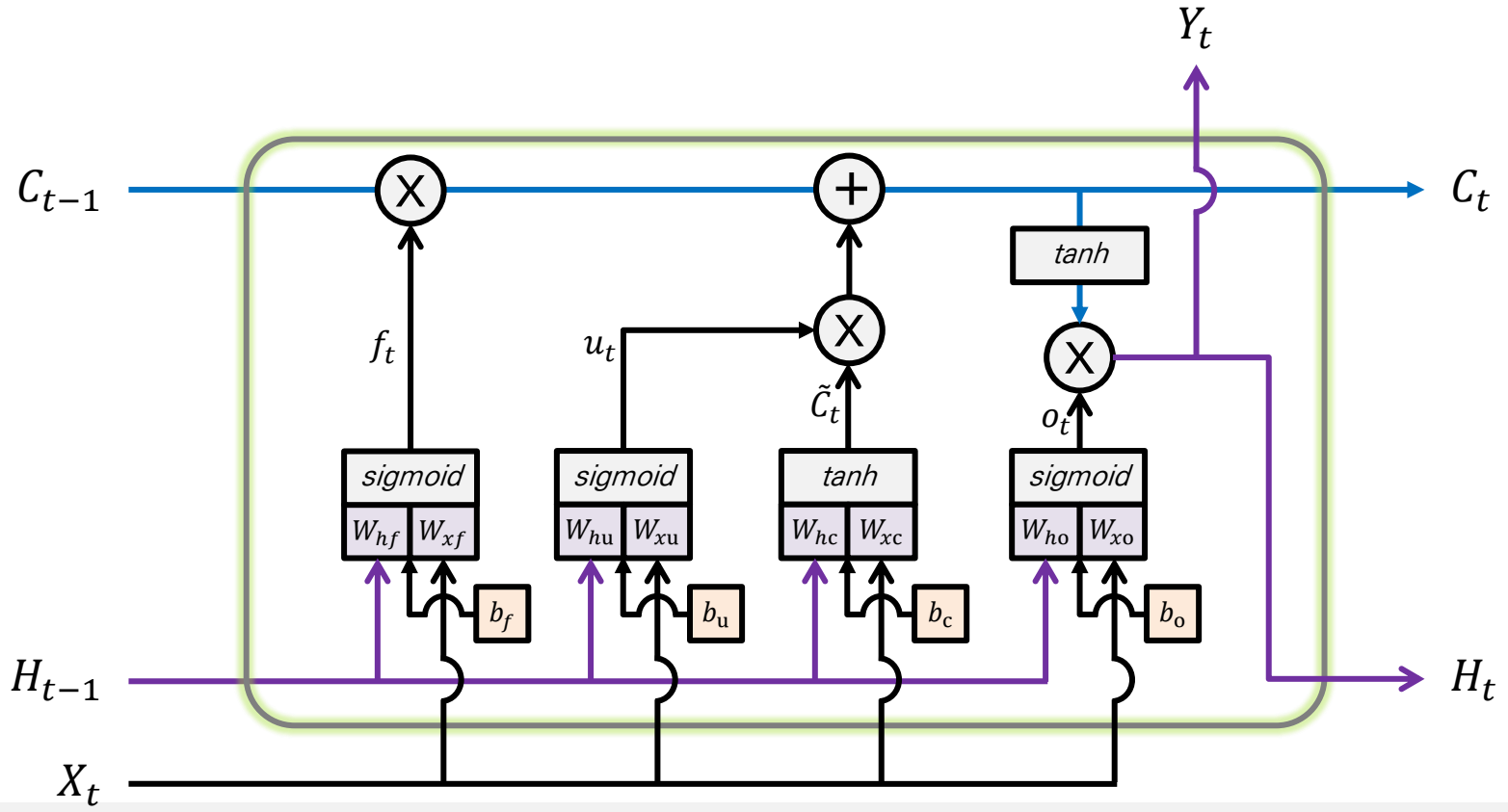
⑤ RNN 층(뉴런 128개)을 하나 더 추가해 보세요. accuracy:

2절. LSTM과 GRU

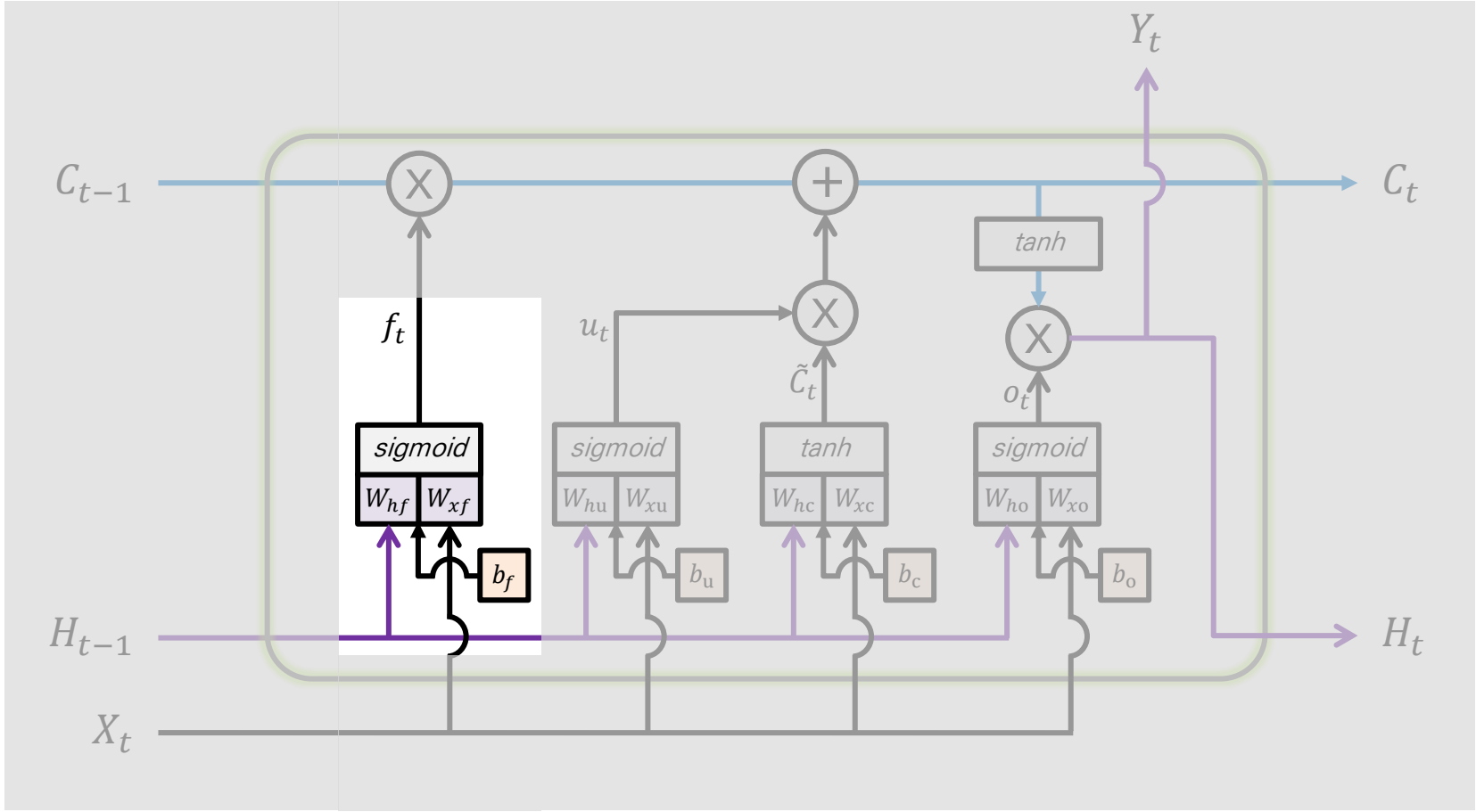
2장. 자연어처리 인공신경망

RNN의 단기 기억(Short-Term Memory) 문제를 해결하기 위해 만들어진 것

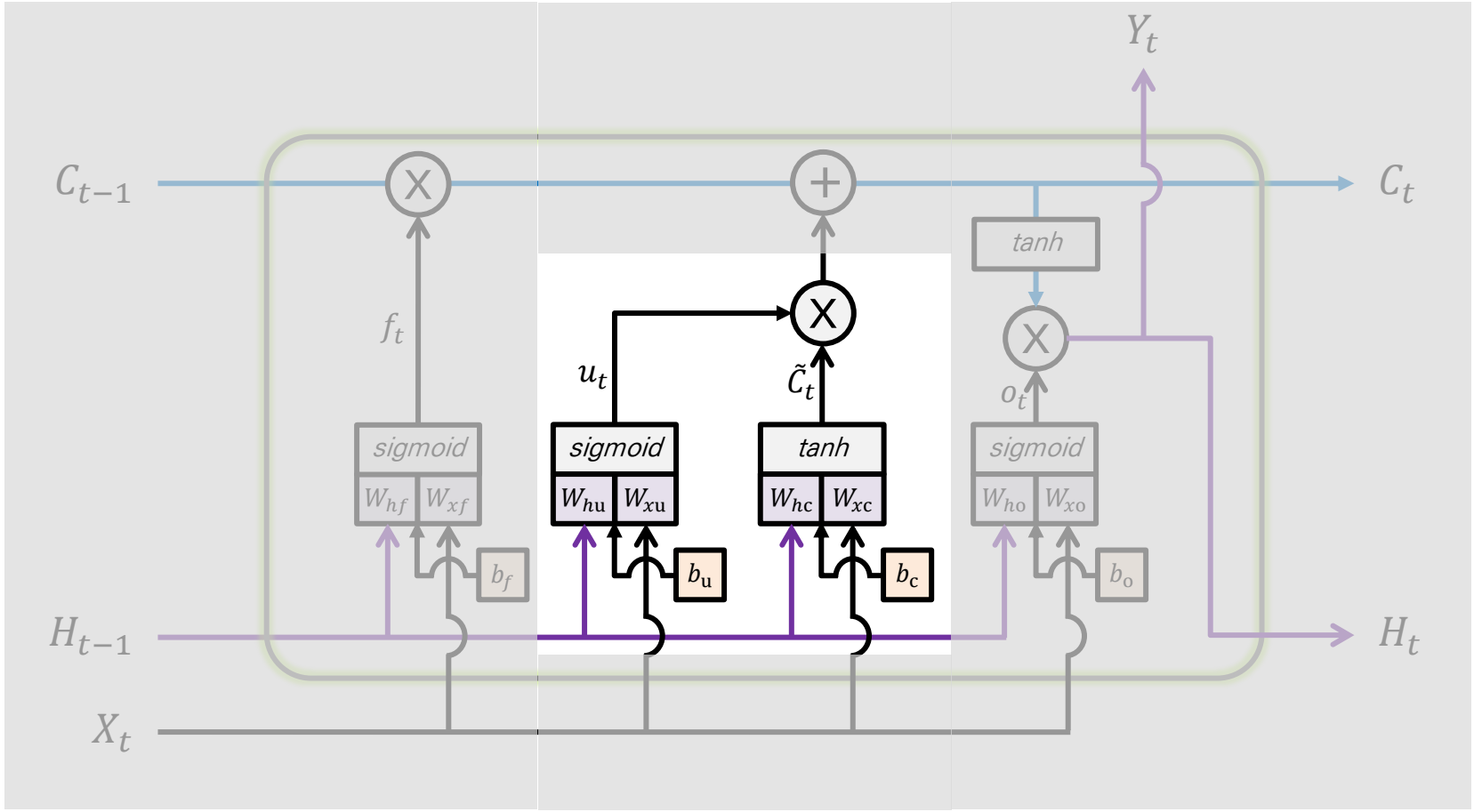
최근의 기억은 유지하지만 오래된 기억은 전달되지 않는 문제를 해결하고자 하는 것



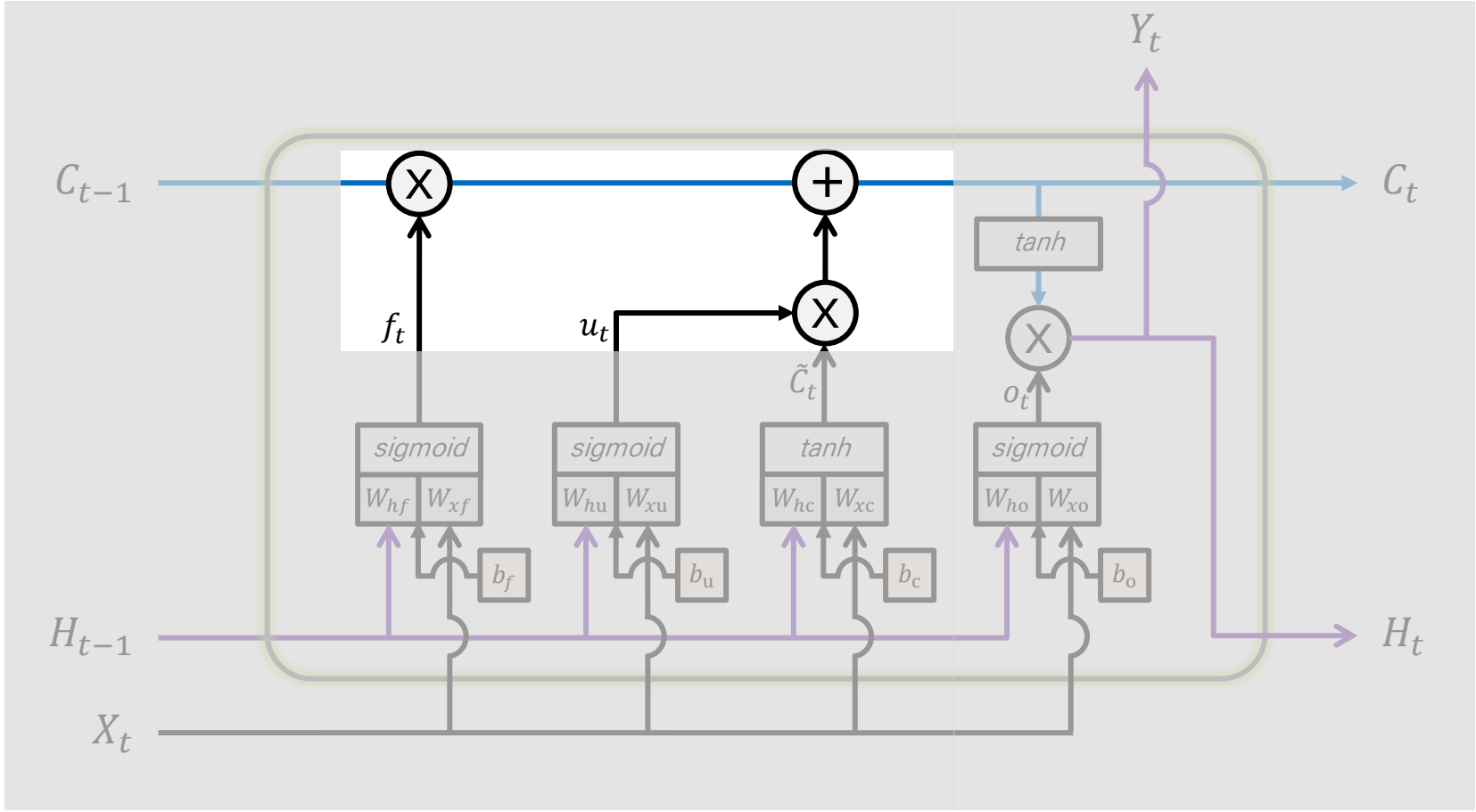
Forget Gate


Y_t
C_{t-1}
C_t
f_t
u_t
$\tilde{C}_t$
o_t
tanh
sigmoid
W_{hf} W_{xf}
b_f
sigmoid
W_{hu} W_{xu}
b_u
tanh
W_{hc} W_{xc}
b_c
sigmoid
W_{ho} W_{xo}
b_o
H_{t-1}
H_t
X_t

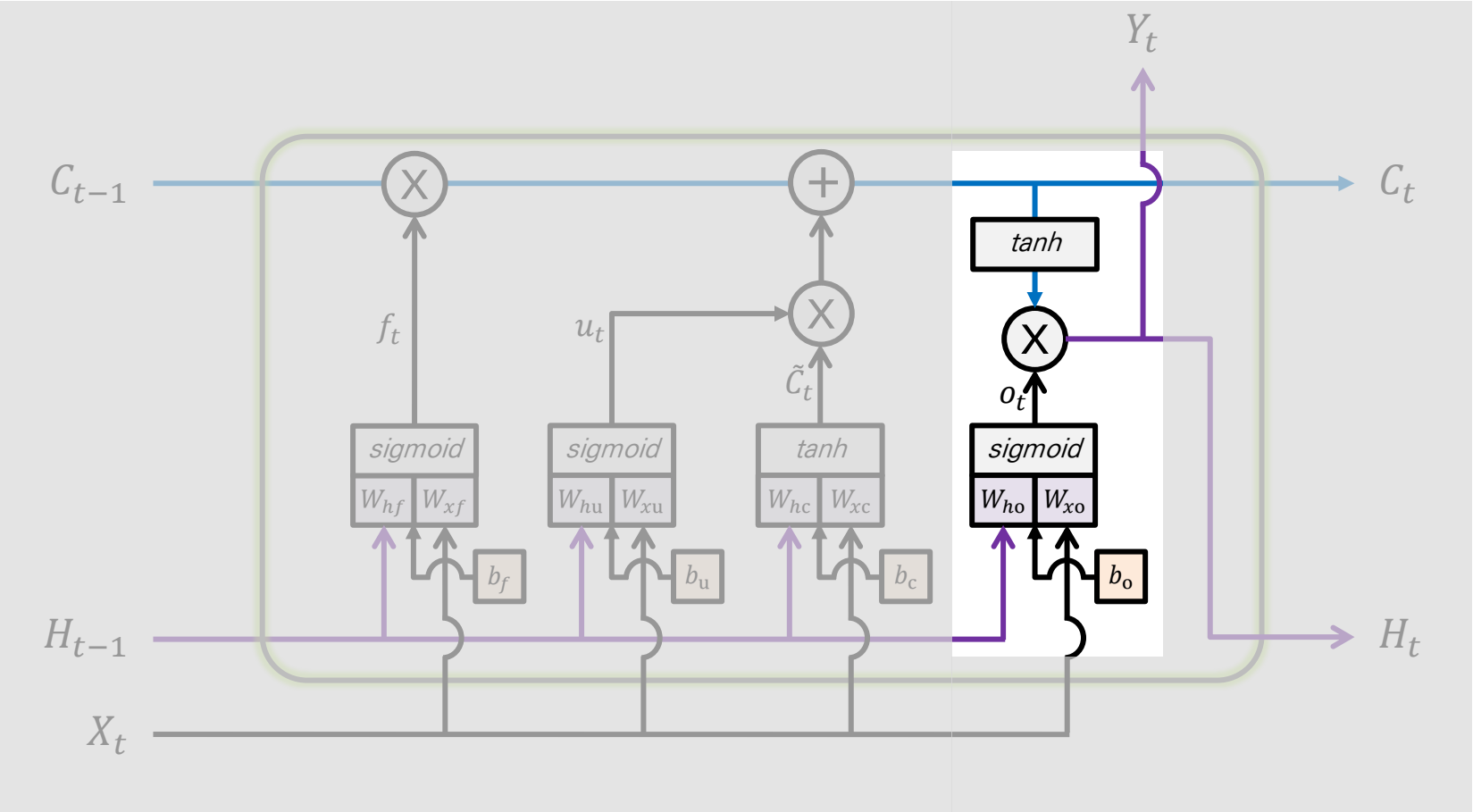
Input Gate


Y_t
C_{t-1}
C_t
f_t
u_t
$\tilde{C}_t$
o_t
tanh
sigmoid
sigmoid
tanh
sigmoid
W_{hf} W_{xf}
W_{hu} W_{xu}
W_{hc} W_{xc}
W_{ho} W_{xo}
b_f
b_u
b_c
b_o
H_{t-1}
H_t
X_t

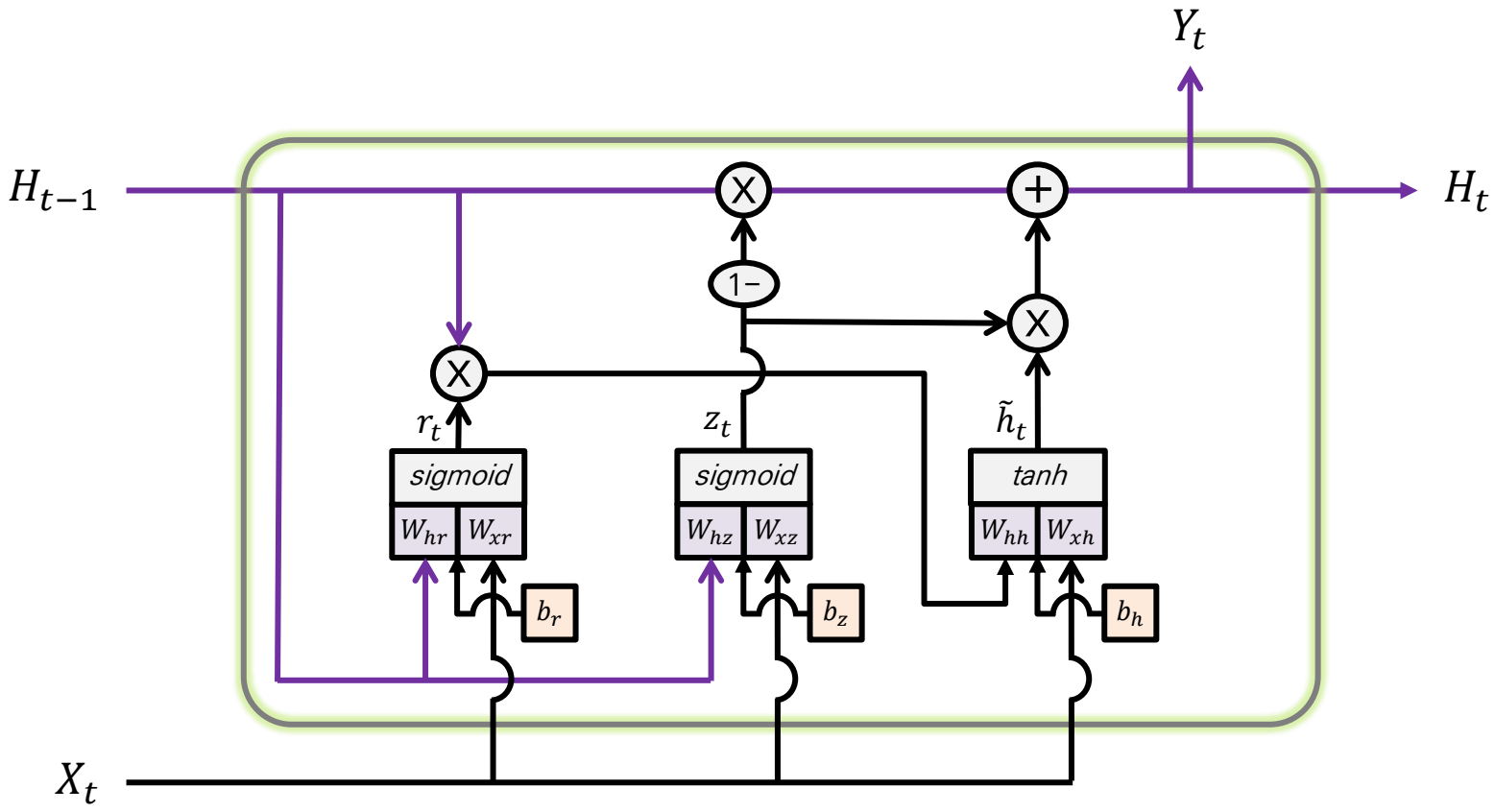
Cell State Update Gate



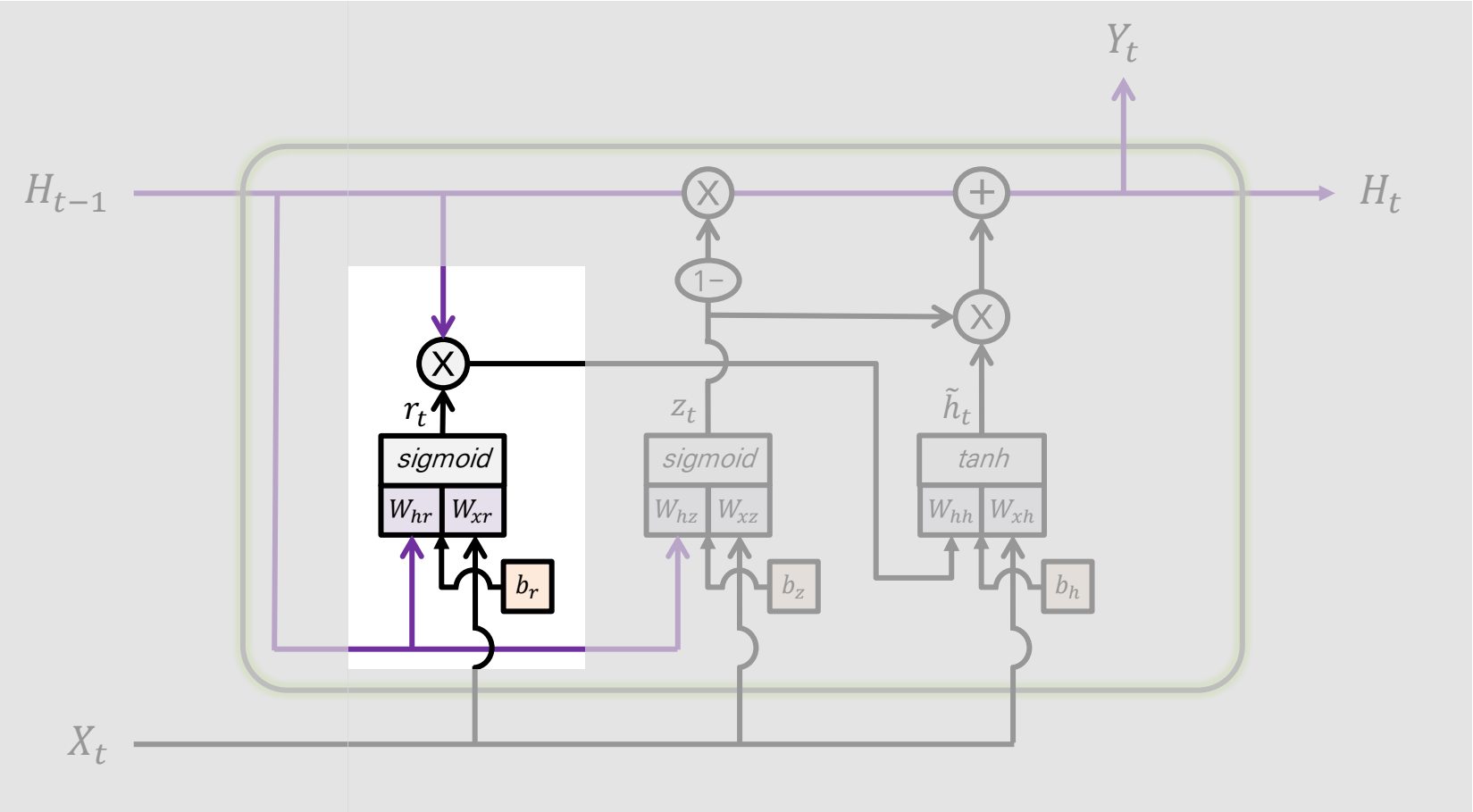
Output Gate


Y_t
C_{t-1}
C_t
f_t
u_t
$\tilde{C}_t$
o_t
tanh
sigmoid
sigmoid
tanh
sigmoid
W_{hf} W_{xf}
W_{hu} W_{xu}
W_{hc} W_{xc}
W_{ho} W_{xo}
b_f
b_u
b_c
b_o
H_{t-1}
H_t
X_t

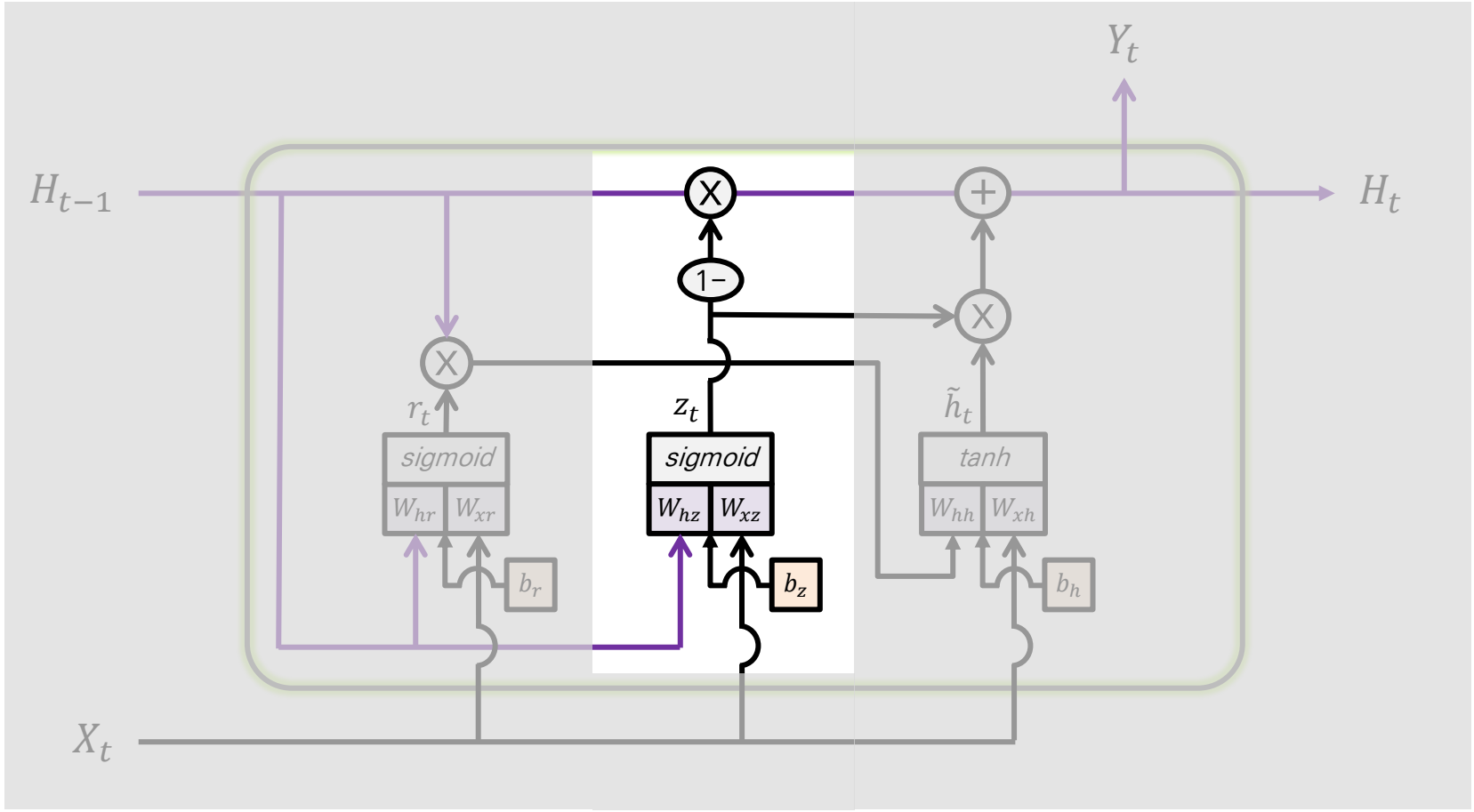
GRU(Gated Recurrent Unit): LSTM의 게이트를 단순화시켜 속도를 빠르게 개선한 것



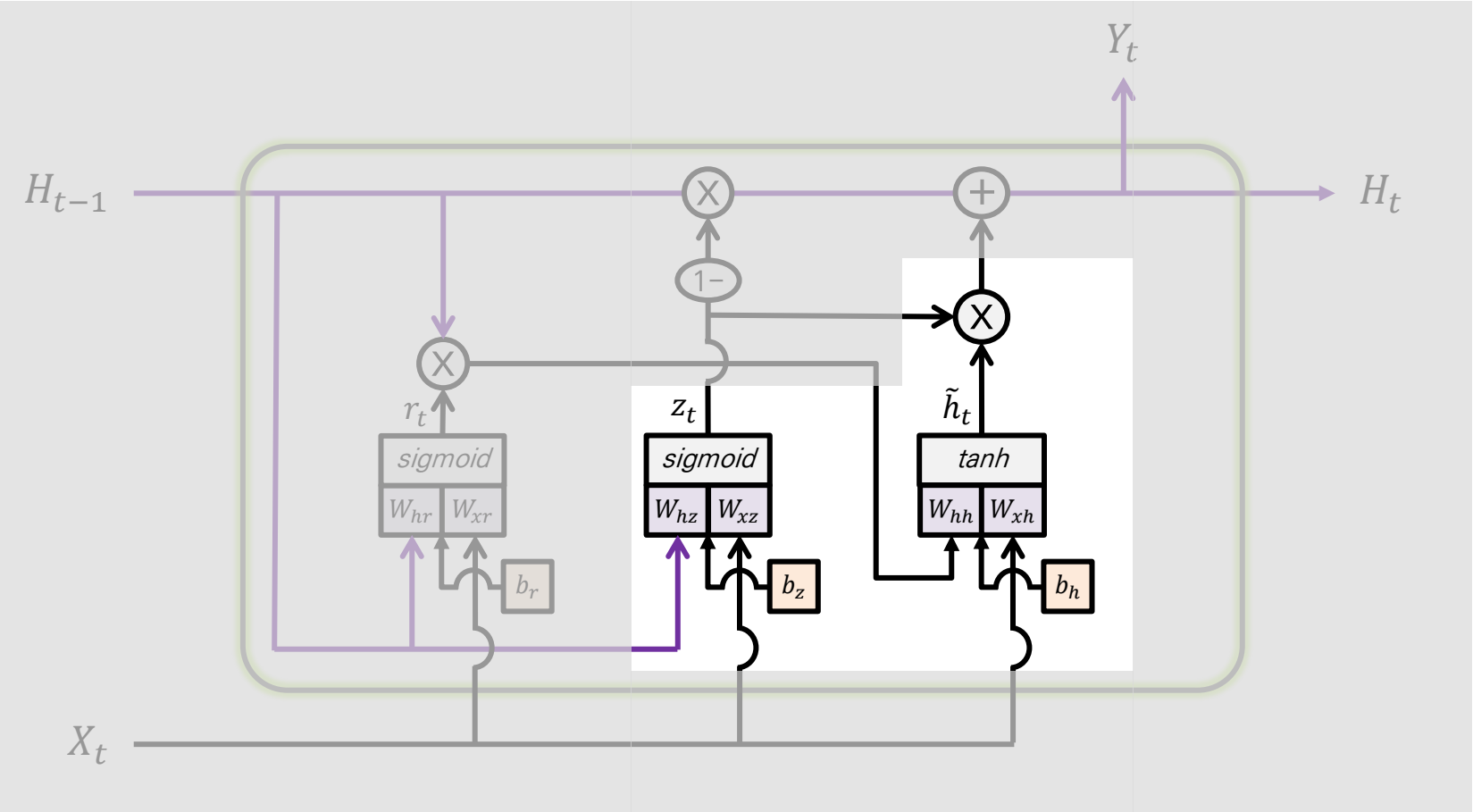
Reset gate


Y_t
H_{t-1}
H_t
X_t
1−
r_t
z_t
$\tilde{h}_t$
sigmoid
sigmoid
tanh
W_{hr}
W_{xr}
W_{hz}
W_{xz}
W_{hh}
W_{xh}
b_r
b_z
b_h

Update gate

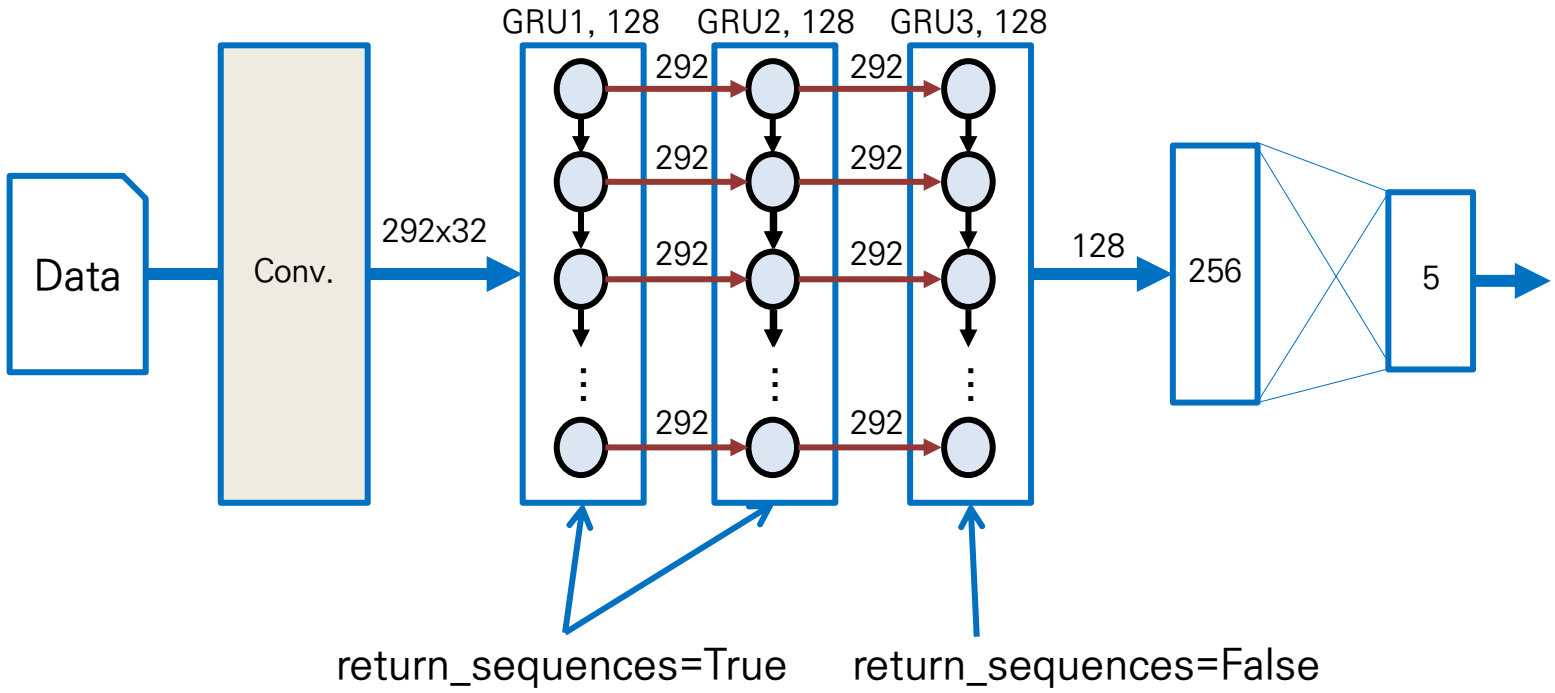

Y_t
H_{t-1}
H_t
X_t
1−
r_t
z_t
$\tilde{h}_t$
sigmoid
sigmoid
tanh
W_{hr}
W_{xr}
W_{hz}
W_{xz}
W_{hh}
W_{xh}
b_r
b_z
b_h

Input gate

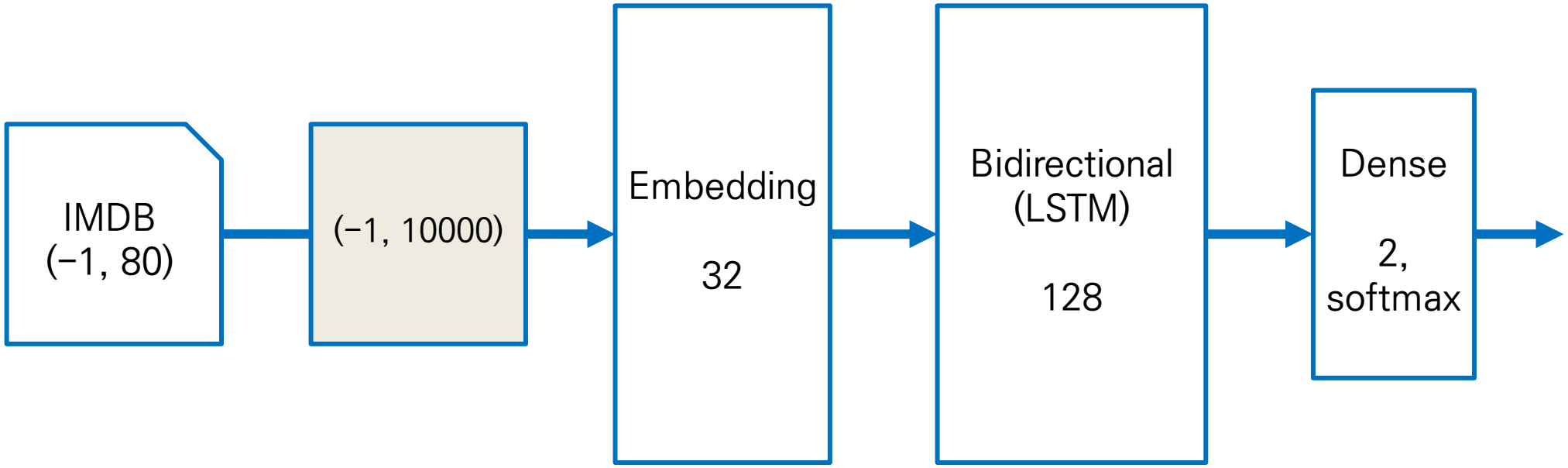

Y_t
H_{t-1}
H_t
1−
r_t
z_t
$\tilde{h}_t$
sigmoid
sigmoid
tanh
W_{hr}
W_{xr}
W_{hz}
W_{xz}
W_{hh}
W_{xh}
b_r
b_z
b_h
X_t

GRU Stacking

GRU를 쌓아 시계열 데이터의 학습 효과를 높일 수 있음



실습 – LSTM을 이용한 영화평 분류하기



실습 – LSTM을 이용한 영화평 분류하기


```
model = Sequential([
    layers.Input(shape=(80,)),
    layers.Embedding(input_dim=10000, output_dim=32), # 10000*32개 파라미터
    layers.Bidirectional(layers.LSTM(64)), # 32x64x2x4(x에 대한 w) + 64x64x2x4(h에 대한 w) + 64x2x4(b)
    layers.Dense(2, activation='softmax') # loss='sparse_categorical_crossentropy'
   # layers.Dense(1, activation='sigmoid') # loss='binary_crossentropy'
])
model.summary() # 0.767
```

Model: "sequential_5"

Layer (type)	Output Shape	Param #
embedding_5 (Embedding)	(None, 80, 32)	320,000
bidirectional_2 (Bidirectional)	(None, 128)	49,664
dense_5 (Dense)	(None, 2)	258

Total params: 369,922 (1.41 MB)
Trainable params: 369,922 (1.41 MB)
Non-trainable params: 0 (0.00 B)

3절. Seq2Seq

2장. 자연어처리 인공신경망

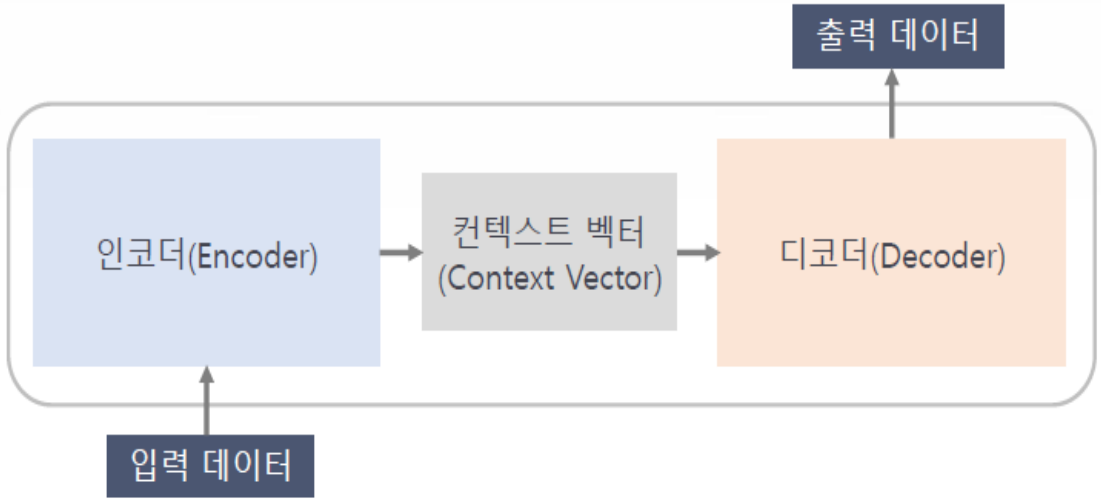
PBMT 기술과 GNMT 기술로 나뉨

PBMT(Phrase-based Machine Translation)

- 문장을 구절 단위로 나누고 번역

GNMT(Google Neural Machine Translation)

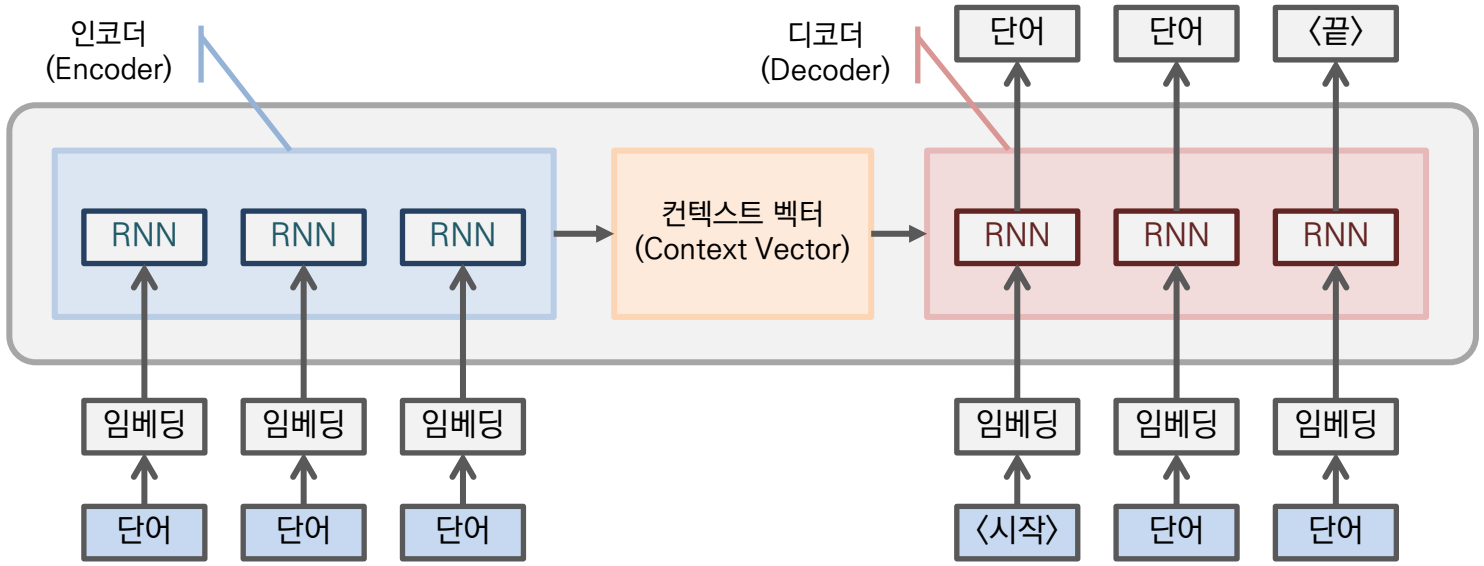
- 인코더 입력, 디코더 입력, 그리고 디코더 출력으로 나눠짐



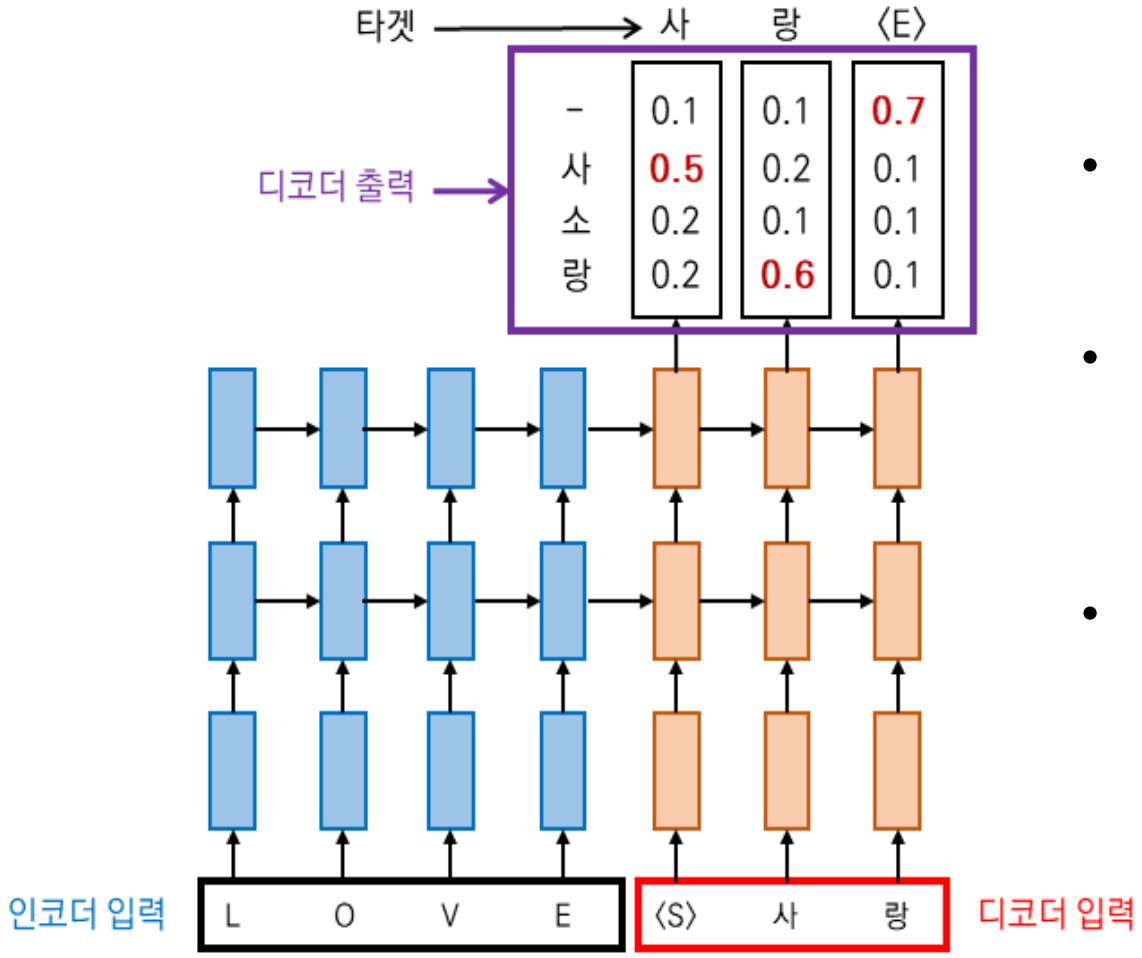
Seq2Seq

RNN을 연결하여 하나의 시퀀스 데이터를 다른 시퀀스 데이터로 변환하는 모델

- 시퀀스(Sequence)는 연관된 일련의 데이터를 의미
- 자연어 문장이 시퀀스에 해당
- 입력을 처리하기 위한 인코더(Encoder)와 출력을 위한 디코더(Decoder)가 연결되는 구조



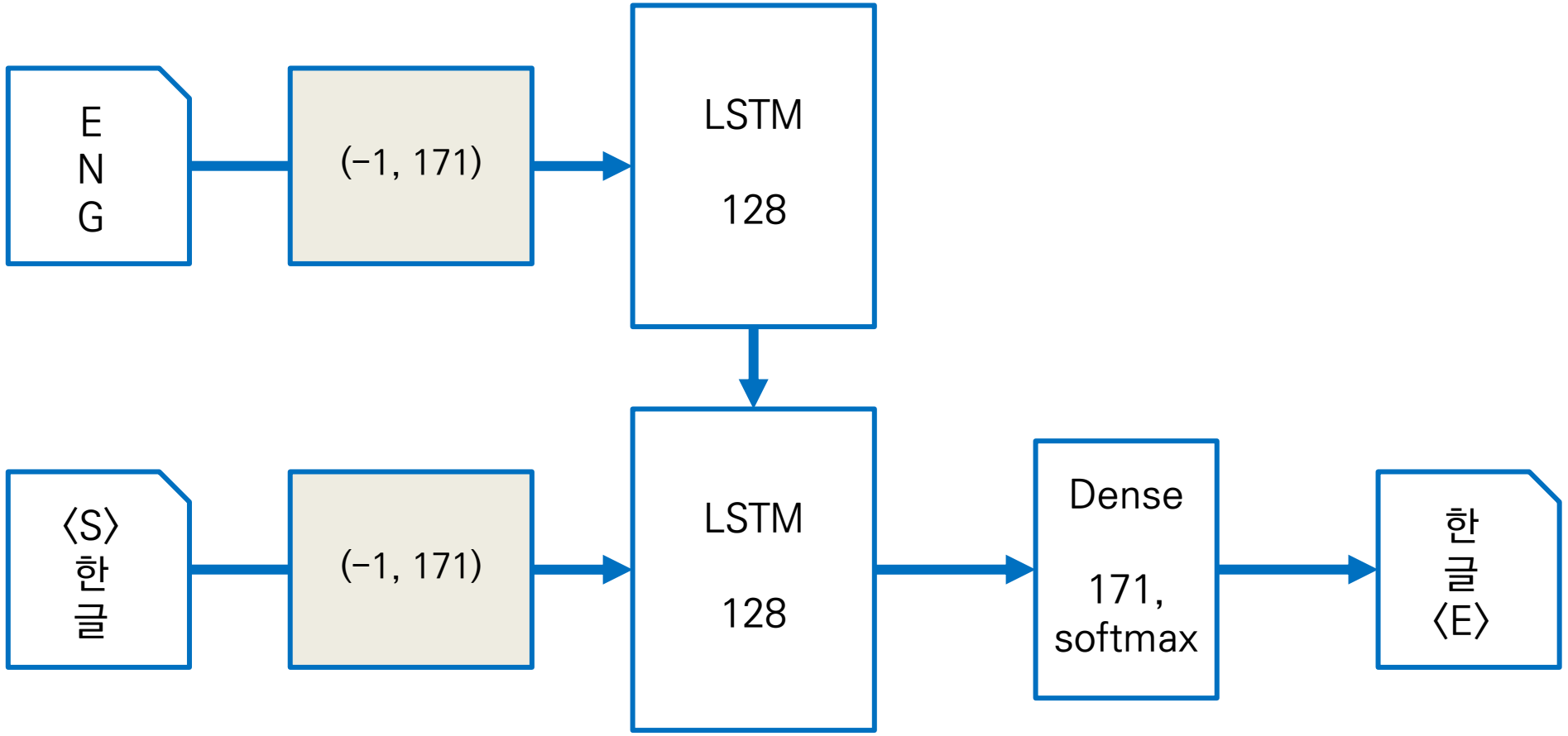
GNMT



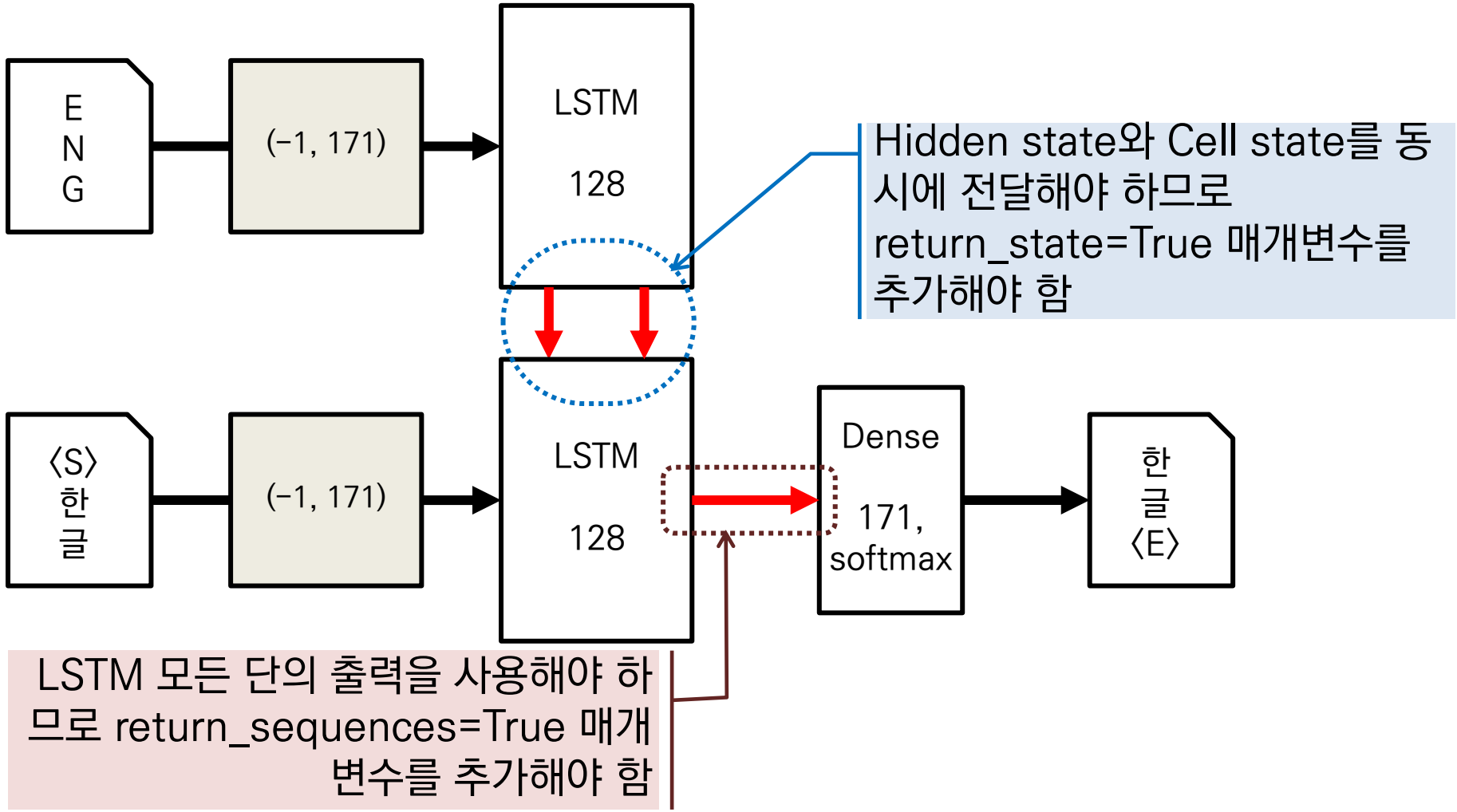
- 인코더는 차례로 입력된 문장들의 모든 단어를 압축하여 컨텍스트 벡터를 생성
- 컨텍스트 벡터는 차대로 입력된 문장의 모든 단어의 정보를 압축한 벡터이며 잠재 벡터(Latent Vector)라고도 함
- 디코더는 입력된 컨텍스트 벡터를 이용하여 출력 시퀀스를 생성하고 출력

예제 – 3장. 4절. 영-한 번역기

실습 – Seq2Seq를 이용한 영–한 번역기



실습 – Seq2Seq를 이용한 영–한 번역기 (Seq2Seq와 LSTM)



실습 – Seq2Seq를 이용한 영-한 번역기 (1. 모델 구축)


```python
from tensorflow.keras import Model, layers

# 입력 단어는 4글자 영단어, 전체문자수는 171개(SEP 포함)
enc_input = layers.Input(shape=(4, 171))
_, state_h, state_c = layers.LSTM(128, return_state=True)(enc_input)

dec_input = layers.Input(shape=(3, 171)) # 한글 단어 2자와 start 토큰 1개
lstm_out = layers.LSTM(128, return_sequences=True)(dec_input, initial_state=[state_h, state_c])

dec_output = layers.Dense(171, activation='softmax')(lstm_out)

model = Model(inputs=[enc_input, dec_input], outputs=dec_output)
model.summary()
```

1. 모델 구축
2. 입력값 토큰화, 임베딩
3. 훈련을 정의하고 학습
4. 평가
5. 예측하기 위한 데이터를 토큰화, 임베딩후 predict
6. 라벨 출력

```python
model.compile(loss='sparse_categorical_crossentropy',
              optimizer='adam') # metrics가 없으면 훈련시 loss만 출력됨
```

Model: "functional_1"

Layer (type)	Output Shape	Param #	Connected to
input_layer (InputLayer)	(None, 4, 171)	0	-
input_layer_1 (InputLayer)	(None, 3, 171)	0	-
lstm (LSTM)	[(None, 128), (None, 128), (None, 128)]	153,600	input_layer[0][0]
lstm_1 (LSTM)	(None, 3, 128)	153,600	input_layer_1[0][0], lstm[0][1], lstm[0][2]
dense (Dense)	(None, 3, 171)	22,059	lstm_1[0][0]

Total params: 329,259 (1.26 MB)
Trainable params: 329,259 (1.26 MB)
Non-trainable params: 0 (0.00 B)

실습 – Seq2Seq를 이용한 영-한 번역기 (2. 입력값 토큰화, 임베딩)


```
# 문자배열 생성
import pandas as pd
import numpy as np
# <START>, <END>, <PAD>
arr1 = [c for c in 'SEPabcdefghijklmnopqrstuvwxyz']
arr2 = pd.read_csv('korean.csv', header=None)
# print(arr2[0].values.tolist())
num_to_char = arr1 + arr2[0].values.tolist()
print(len(num_to_char))
char_to_num = {char:i for i, char in enumerate(num_to_char)}
# print(char_to_num)
```

```
171
```

```
# 학습용 단어셋 불러오기
raw = pd.read_csv('translate.csv', header=None)
eng_kor = raw.values.tolist()
print(len(eng_kor)) # 학습할 전체 단어 수 110개
```

```
110
```

```
# 단어를 숫자 배열로 변환
temp_eng = 'love'
temp_eng_n = [char_to_num[c] for c in temp_eng]
print(temp_eng_n)
temp_kor = '사랑'
np.eye(171)[temp_eng_n].shape
```

```
[14, 17, 24, 7]
(4, 171)
```

실습 – Seq2Seq를 이용한 영-한 번역기 (3. 학습)


```
# 단어를 원-핫 인코딩된 배열로 변환
def encode(eng_kor):
    enc_in = []
    dec_in = []
    rnn_out = [] # decoder output
    for seq in eng_kor:
        eng = [char_to_num[c] for c in seq[0]]
        enc_in.append(np.eye(171)[eng])

        kor = [char_to_num[c] for c in ('S'+seq[1])]
        dec_in.append(np.eye(171)[kor])

        target = [char_to_num[c] for c in (seq[1] + 'E')]
        rnn_out.append(target)

    enc_in = np.array(enc_in)
    dec_in = np.array(dec_in)
    rnn_out = np.array(rnn_out)
    rnn_out = np.expand_dims(rnn_out, axis=2)
    return enc_in, dec_in, rnn_out
```

```
X_enc, X_dec, y_rnn = encode(eng_kor)
```

```
%%time
model.fit([X_enc, X_dec], y_rnn, epochs=500)
```

```
Epoch 1/500
4/4 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 3s 11ms/step - loss: 5.1339
Epoch 2/500
4/4 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - loss: 5.0933
```

1. 모델 구축
2. 입력값 토큰화, 임베딩
3. 훈련을 정의하고 학습
4. 평가
5. 예측하기 위한 데이터를 토큰화, 임베딩후 predict
6. 라벨 출력

실습 – Seq2Seq를 이용한 영-한 번역기 (4. 예측)


```
enc_in, dec_in, _ = encode([['tall', 'PP']])
# print(enc_in)
pred = model.predict([enc_in, dec_in])
# print(pred.shape)
word = np.argmax(pred[0], axis=-1)
# print(word)
num_to_char[word[0]], num_to_char[word[1]]
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 246ms/step
('크', '다')
```

```
from numpy.random import randint
pick = randint(0, len(eng_kor), 5)
choose = [ [eng_kor[i][0], 'PP'] for i in pick]
print(choose)
```

```
[['thin', 'PP'], ['wing', 'PP'], ['ring', 'PP'], ['goal', 'PP'], ['pick',
```

```
enc_in, dec_in, _ = encode(choose)
pred = model.predict([enc_in, dec_in])
print(pred.shape)
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 251ms/step
(5, 3, 171)
```

```
for i in range(len(choose)):
    eng = choose[i][0]
    word = np.argmax(pred[i], axis=-1)
    kor = ''
    for j in range(2):
        kor = kor + num_to_char[word[j]]
    print(eng, kor)
```

```
thin 얇은
wing 날개
ring 반지
goal 목적
pick 선택
```

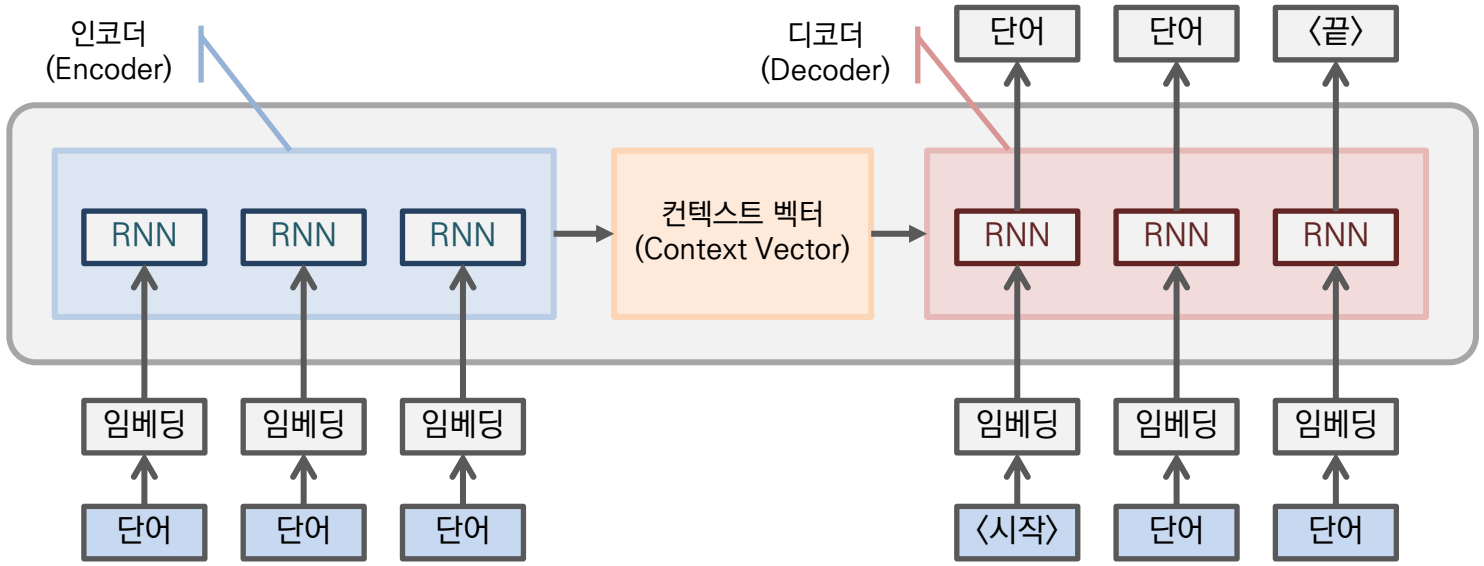
4절. 어텐션

2장. 자연어처리 인공신경망

Seq2Seq의 문제점

인코더가 입력 시퀀스를 하나의 벡터로 압축하는 과정에서 입력 시퀀스의 일부 정보 손실

- RNN 구조의 근본적인 문제점
- 경사 소실(Vanishing Gradient)이 발생할 가능성이 있음
- 입력 데이터의 길이가 길어지면 성능이 저하되는 현상이 발생할 가능성이 있음



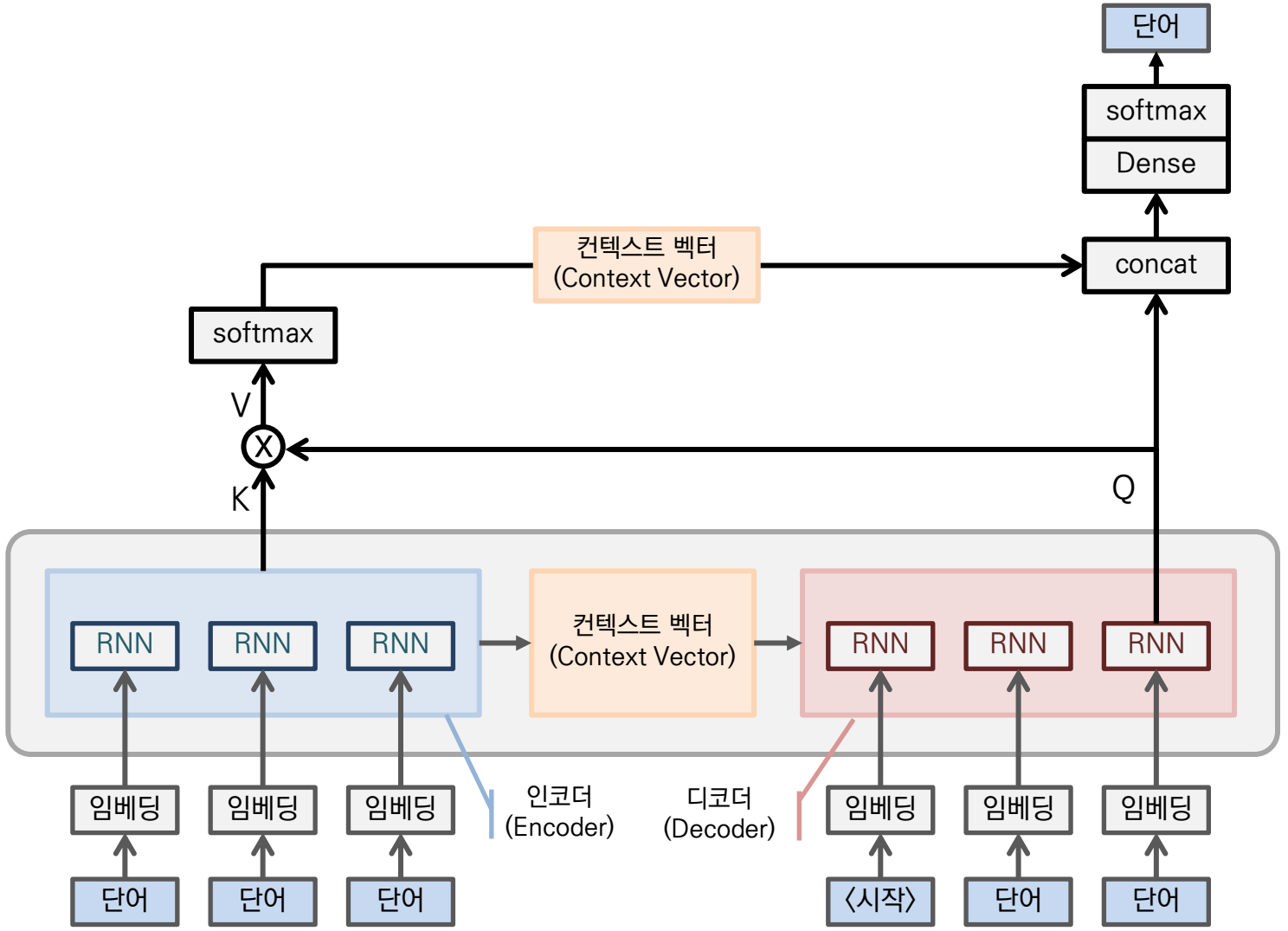
어텐션 함수는 Q(Query), K(Keys), V(Values)를 매개변수로 사용

- Q(Query)는 특정 시점에서의 디코더 셀의 은닉 상태
- K(Key)는 모든 시점에서의 인코더 셀의 Q를 반영하기 전 은닉 상태
- V(Value)는 모든 시점에서의 인코더 셀의 Q 반영 후 은닉 상태

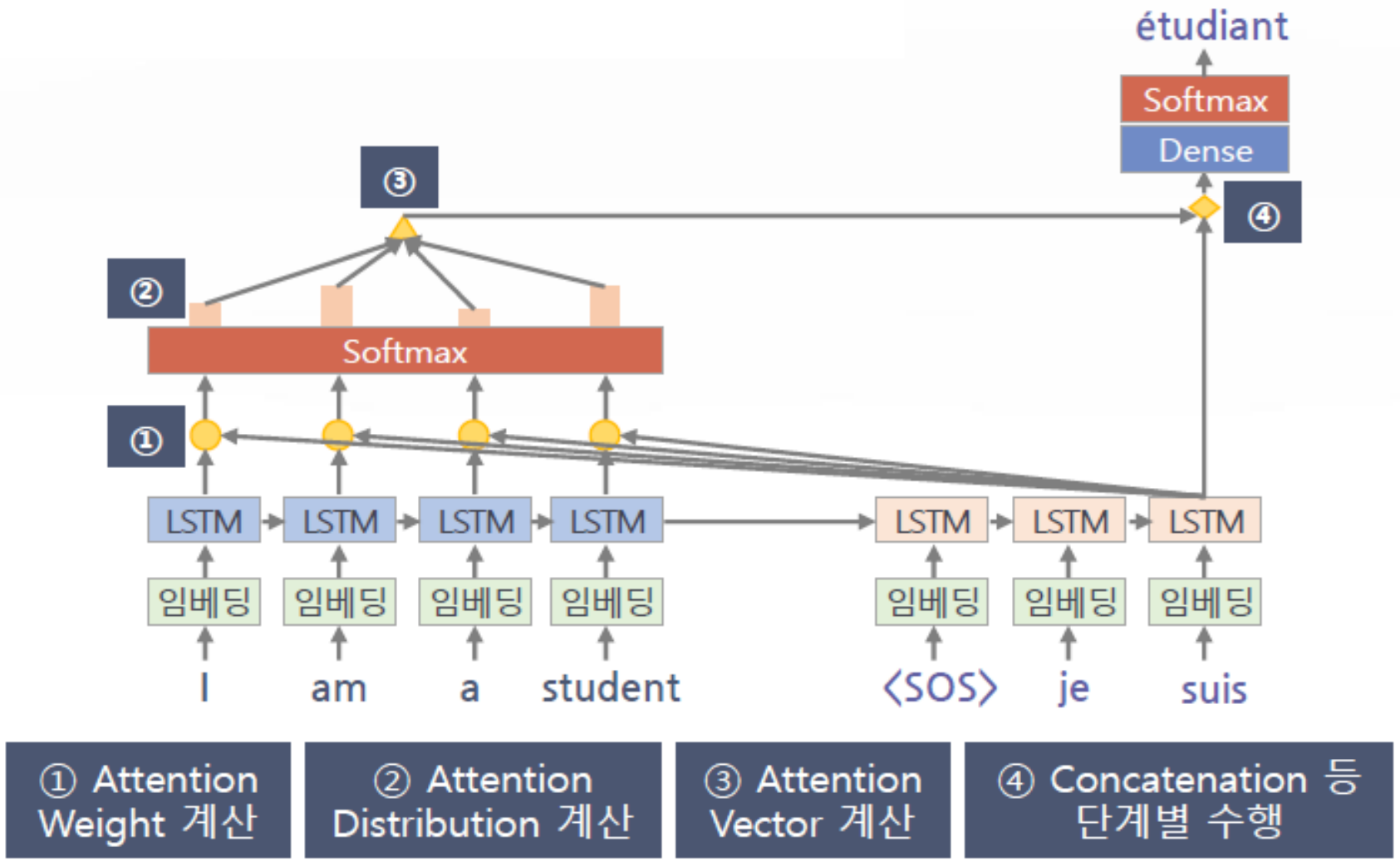
$$Attention\ Value = Attention(Q, K, V)$$

- 어텐션 함수는 주어진 질의(Query)에 대해서 모든 키(Key)와 각각의 유사도를 계산
- 계산된 유사도를 키와 매핑되어 있는 각각의 값(Value)에 반영
- 유사도가 반영된 값(Value)을 모두 어텐션 값(Attention Value)으로 반환

어텐션


단어
softmax
Dense
concat
컨텍스트 벡터
(Context Vector)
softmax
V
X
K
Q
RNN
RNN
RNN
컨텍스트 벡터
(Context Vector)
RNN
RNN
RNN
임베딩
임베딩
임베딩
인코더
(Encoder)
디코더
(Decoder)
임베딩
임베딩
임베딩
단어
단어
단어
〈시작〉
단어
단어

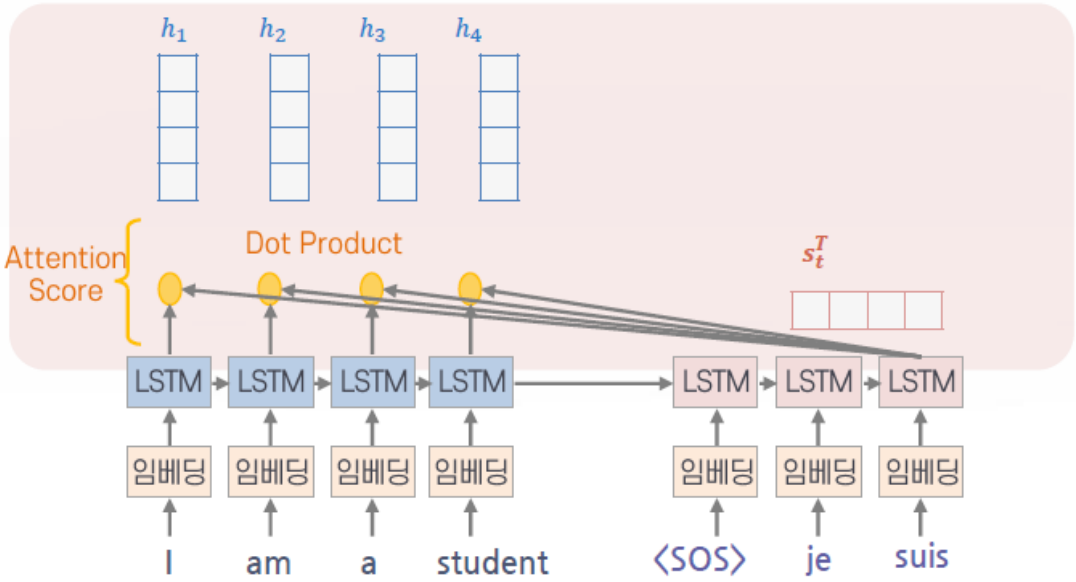
어텐션


étudiant
Softmax
Dense
③
④
②
Softmax
①
LSTM
LSTM
LSTM
LSTM
LSTM
LSTM
LSTM
임베딩
임베딩
임베딩
임베딩
임베딩
임베딩
임베딩
I
am
a
student
〈SOS〉
je
suis
① Attention Weight 계산
② Attention Distribution 계산
③ Attention Vector 계산
④ Concatenation 등 단계별 수행

어텐션 적용 과정

① 어텐션 가중치(Attention Weights) 계산
② 어텐션 분포(Attention Distribution) 계산
③ 어텐션 벡터(Attention Vector) 계산
④ 연결(Concatenation)

어텐션 가중치 계산

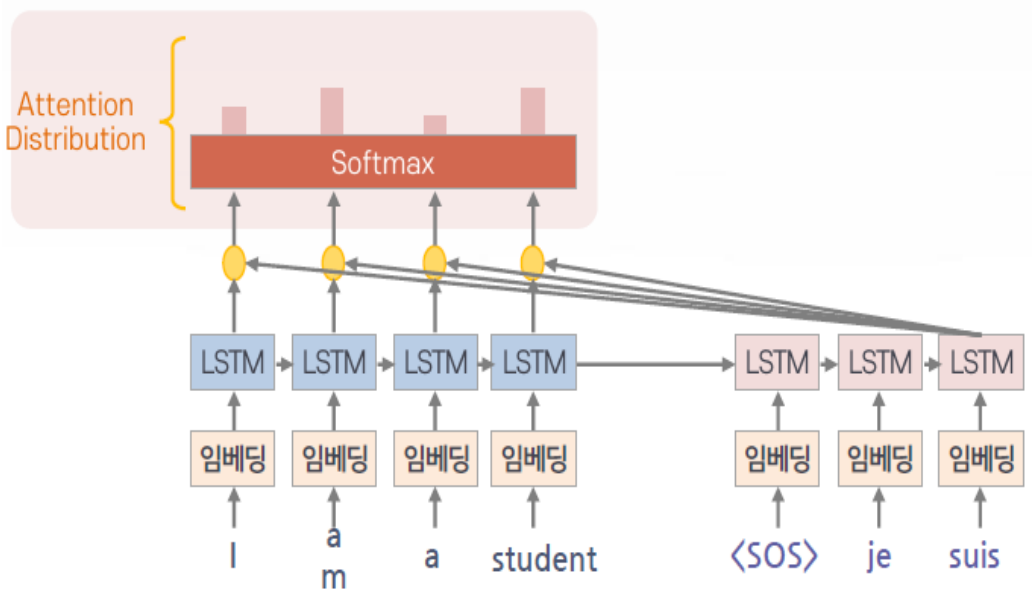


- 디코더의 t 시점 Timestep의 은닉 상태와 인코더의 모든 Timestep의 은닉 상태와 행렬 곱셈 연산을 수행
- 행렬 곱 연산의 결과로 어텐션 가중치를 계산

어텐션 적용 과정

① 어텐션 가중치(Attention Weights) 계산
② 어텐션 분포(Attention Distribution) 계산
③ 어텐션 벡터(Attention Vector) 계산
④ 연결(Concatenation)

어텐션 분포 계산

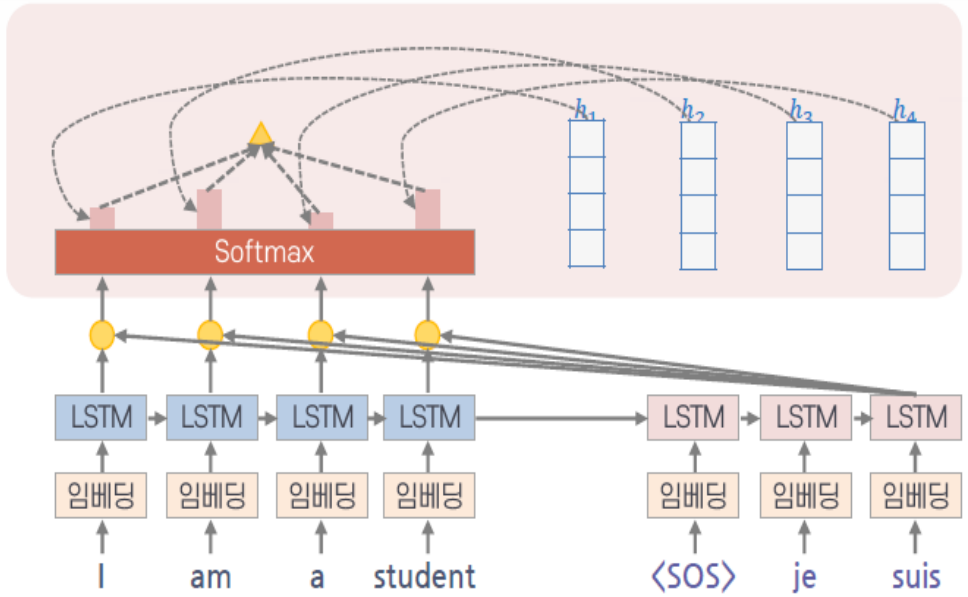


- 인코더의 모든 은닉 상태의 어텐션 가중치의 벡터에 소프트맥스(softmax) 함수를 적용
- 어텐션 분포(Attention Distribution)는 소프트맥스 함수를 적용한 후 각각 어텐션 가중치의 벡터 합이 1이 되는 확률 분포
- 각각의 값은 어텐션의 가중치에 해당

어텐션 적용 과정

① 어텐션 가중치(Attention Weights) 계산
② 어텐션 분포(Attention Distribution) 계산
③ 어텐션 벡터(Attention Vector) 계산
④ 연결(Concatenation)

어텐션 벡터 계산

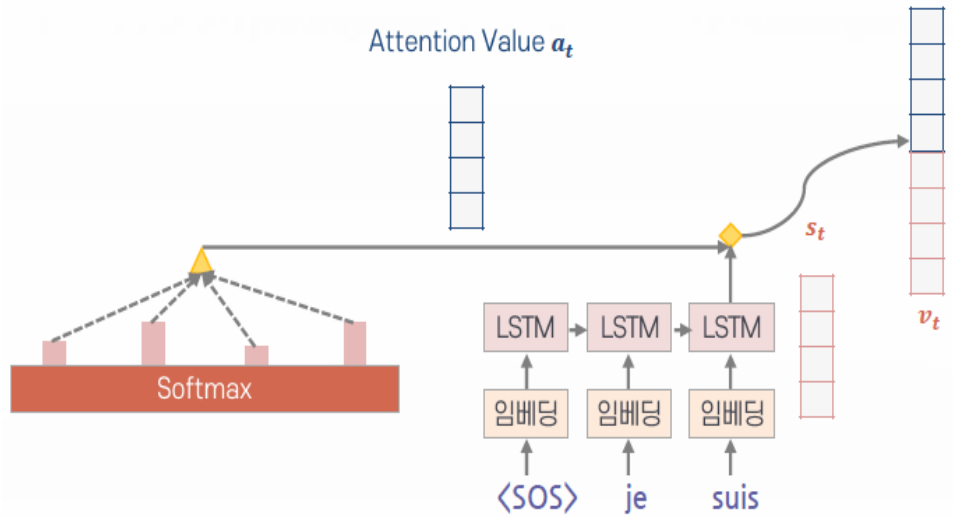


- 인코더의 은닉 상태와 어텐션 가중치를 곱한 후 모두 합하여 어텐션 벡터(Attention Vector)를 계산
- 어텐션 벡터는 인코더의 컨텍스트를 포함하는 컨텍스트 벡터(Context Vector)임

어텐션 적용 과정

① 어텐션 가중치(Attention Weights) 계산
② 어텐션 분포(Attention Distribution) 계산
③ 어텐션 벡터(Attention Vector) 계산
④ 연결(Concatenation)

연결



- 어텐션 벡터와 디코더의 t Timestep의 은닉 상태를 연결(Concatenation)
- 어텐션 벡터와 디코더의 t Timestep의 은닉 상태를 연결한 벡터는 디코더에서 예측을 수행하기 위해 Fully Connected Layer의 입력 및 다음 Timestep의 입력으로 사용

실습 – 어텐션을 이용한 영–한 번역기

2장. 자연어처리 인공신경망 / 4절. 어텐션

```python
import tensorflow as tf
from tensorflow.keras import Model, layers

# 인코더 입력 정의: 영문 단어가 입력, 각 단어는 4자 길이, 모든 문자의 수는 171개
enc_input = layers.Input(shape=(4, 171))

# 인코더 LSTM 정의: 모든 타임스텝의 출력을 반환하도록 설정
enc_output, state_h, state_c = layers.LSTM(128, return_sequences=True, return_state=True)(enc_input)

# 디코더 입력 정의
dec_input = layers.Input(shape=(3, 171))  # 한글 단어는 2자 길이 + <Start> 토큰, 모든 문자의 수는 171개

# 디코더 LSTM 정의: 모든 시퀀스의 출력을 반환하도록 설정
dec_lstm_output, _, _ = layers.LSTM(128, return_sequences=True, return_state=True)(dec_input, initial_state=[state_h, state_c])

# 어텐션 메커니즘 정의
context_vector = layers.Attention()([dec_lstm_output, enc_output])

# 컨텍스트 벡터와 디코더 LSTM 출력을 결합
context_and_lstm_output = layers.Concatenate()([context_vector, dec_lstm_output])

# 디코더 출력층 정의: 출력 크기는 모든 문자의 수인 171, softmax 활성화 함수를 사용
output = layers.Dense(171, activation='softmax')(context_and_lstm_output)

# 모델 정의: 인코더 입력(enc_input)과 디코더 입력(dec_input)을 모델의 입력으로, 디코더 출력을 모델
model = Model(inputs=[enc_input, dec_input], outputs=[output])

# 모델 요약 출력
model.summary()
```

- Seq2Seq 예제 코드의 모델을 수정하세요.
- 나머지 코드는 같습합니다.

Model: "functional"

Layer (type)	Output Shape	Param #	Connected to
input_layer (InputLayer)	(None, 4, 171)	0	-
input_layer_1 (InputLayer)	(None, 3, 171)	0	-
lstm (LSTM)	[(None, 4, 128), (None, 128), (None, 128)]	153,600	input_layer[0][0]
lstm_1 (LSTM)	[(None, 3, 128), (None, 128), (None, 128)]	153,600	input_layer_1[0][0], lstm[0][1], lstm[0][2]
attention (Attention)	(None, 3, 128)	0	lstm_1[0][0], lstm[0][0]
concatenate (Concatenate)	(None, 3, 256)	0	attention[0][0], lstm_1[0][0]
dense (Dense)	(None, 3, 171)	43,947	concatenate[0][0]

Total params: 351,147 (1.34 MB)
Trainable params: 351,147 (1.34 MB)
Non-trainable params: 0 (0.00 B)

5절. 트랜스포머

2장. 자연어처리 인공신경망

트랜스포머

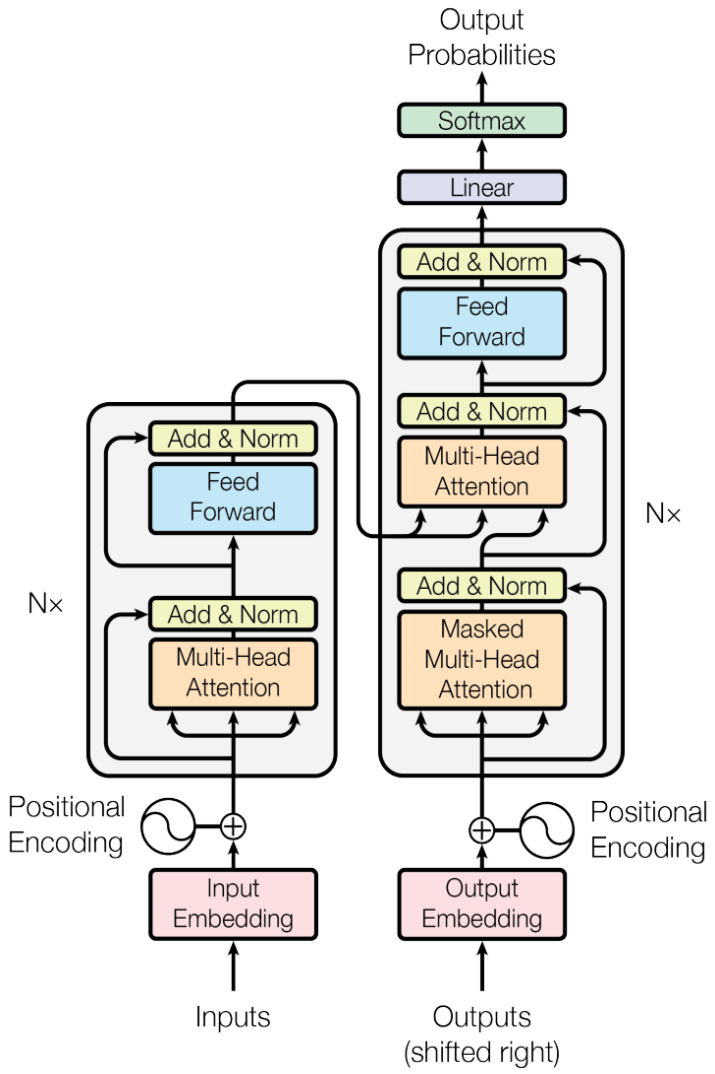
2017년 구글이 “Attention is all you need” 논문에서 발표한 모델

https://arxiv.org/abs/1706.03762

인코더-디코더 구조로 되어있으면서 RNN을 제외하고 어텐션만으로 구현한 모델

입력 문장에 대한 처리 후 출력 문장을 출력

트랜스포머는 인코더와 디코더 및 인코더와 디코더를 연결하기 위한 Connection을 구성

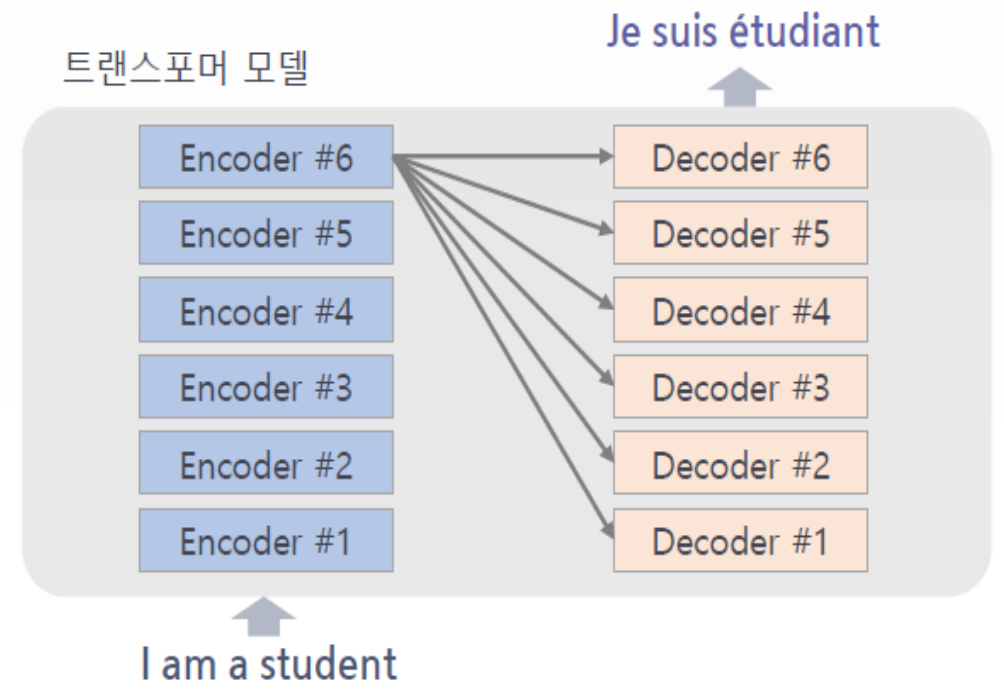


트랜스포머의 인코더와 디코더 구조

인코딩 컴포넌트는 여러 인코더로 구성되어 있음

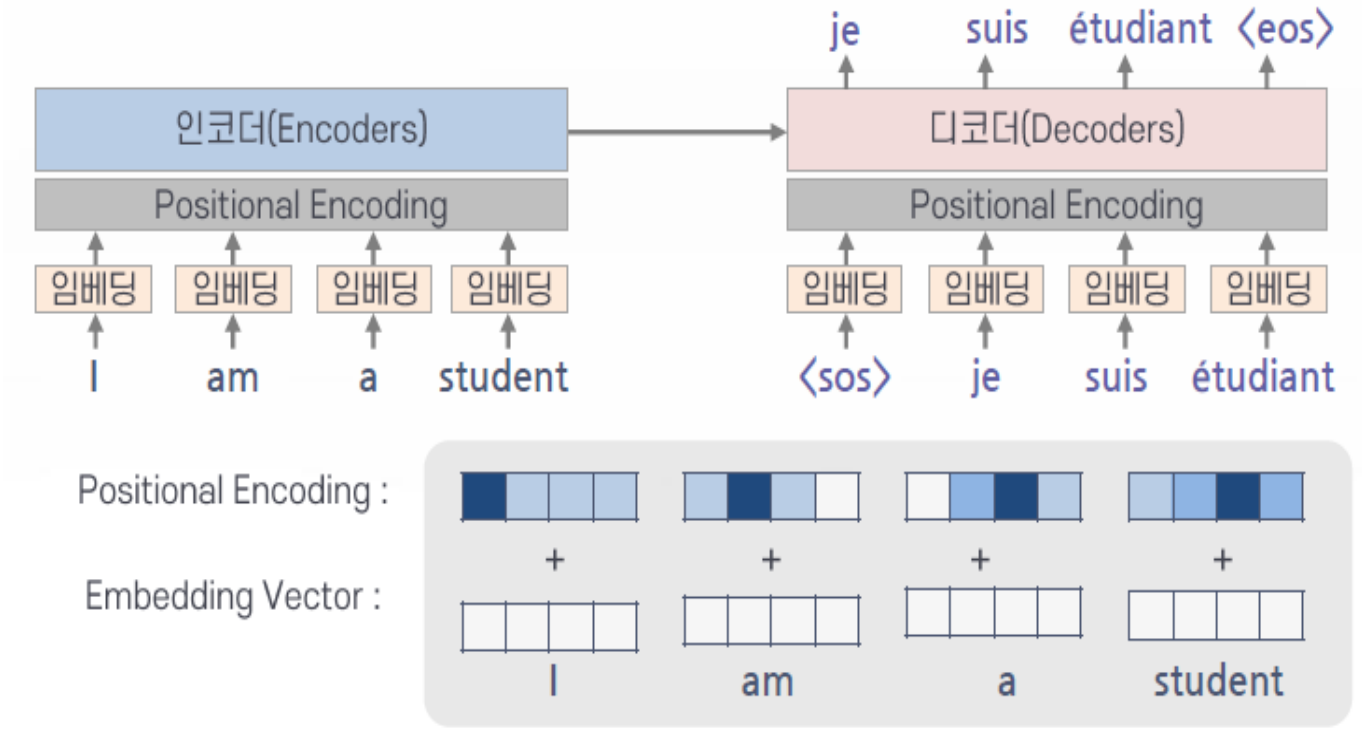
디코딩 컴포넌트는 인코딩 컴포넌트와 같은 개수의 디코더로 구성

- 논문에는 6개의 인코더와 디코더로 구성되어 있으나 임의의 개수로 변경 가능



트랜스포머의 임베딩과 Positional 인코딩

Positional 인코딩은 트랜스포머에서 각 입력 단어의 위치 정보 부여를 위해 각 단어의 임베딩 벡터에 위치 정보들을 추가하여 모델의 입력으로 사용하는 방법



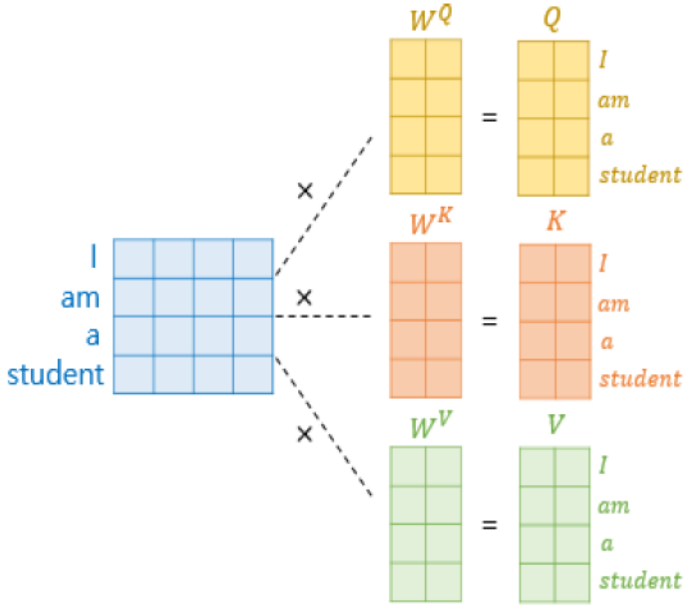
트랜스포머의 Self-Attention

트랜스포머의 Self-Attention은 현재 처리 중인 단어에 대해 다른 연관 있는 단어들과의 맥락을 파악하는 방법을 제공

'The animal didn't cross the street because it was too tired'

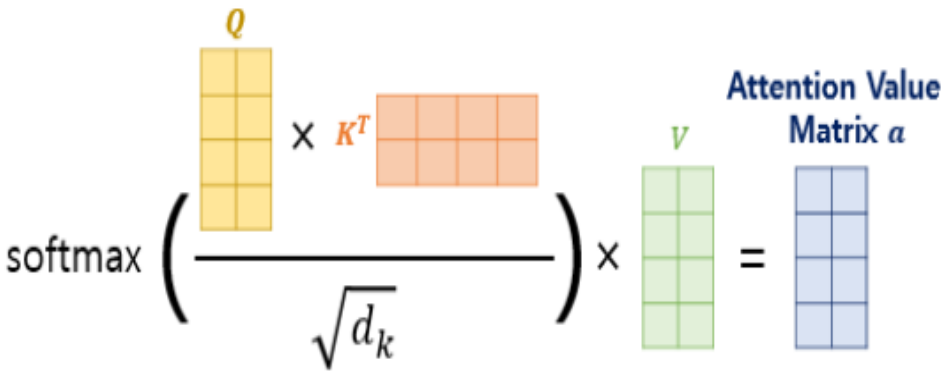
이 문장의 경우 "it"이 가리키는 것은 무엇일까요?
animal일까요? 아니면 street일까요?

- Self-Attention의 첫 단계는 입력 문장에 대해 Query, Key, Value를 계산하는 것
- 입력 문장의 512 크기의 벡터와 학습할 가중치(WQ, WK, WV)를 곱하여 64 크기의 Query, Key, Value 벡터를 생성



트랜스포머의 Self-Attention

- Query에 Key의 전치(Transpose) 행렬을 곱해서 내적을 계산
- 만일 Query와 Key가 특정 문장에서 중요한 역할을 하고 있다면 트랜스포머는 이들 사이의 내적(Dot Product)값을 크게 하는 방향으로 학습
- 내적 값이 커지면 해당 Query와 Key가 벡터 공간상 가까이 있을 확률이 높음
- Key의 벡터 크기인 64의 제곱근인 8로 나눈 후 소프트맥스 함수를 적용
- Value와 내적을 곱하여 어텐션 값인 Z를 계산



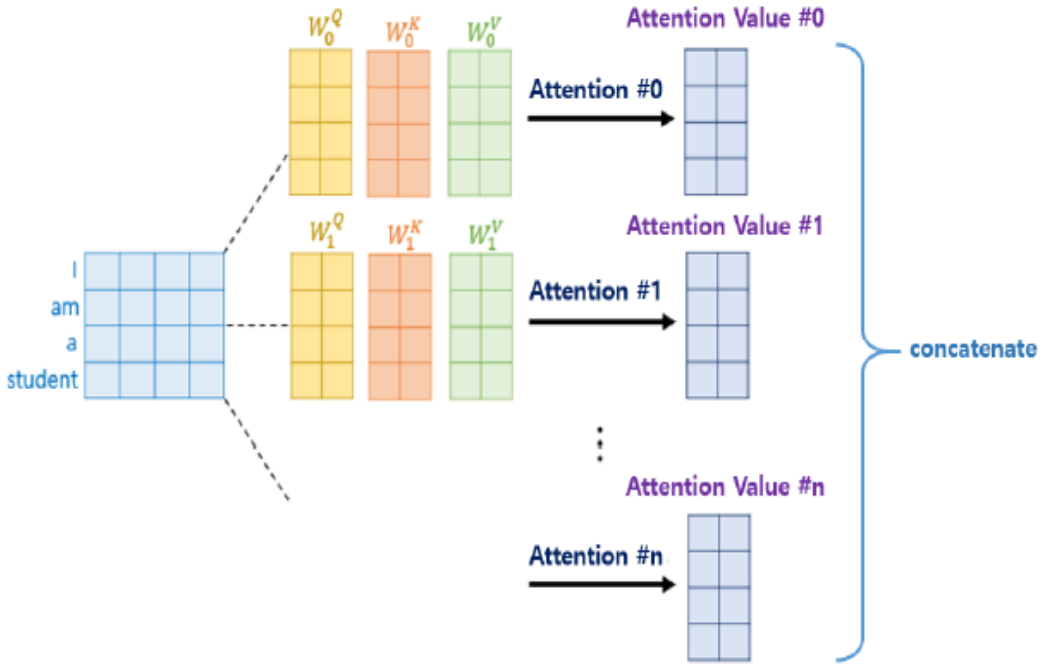
$$Attention(Q, K, V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

위 수식에서 $\sqrt{d_k}$로 나눈 이유는 Query와 Key의 내적 행렬의 분산(Variance)을 축소하고 경사 소실(Gradient Vanishing)을 발생을 방지하기 위해서입니다.

Multi-head Self Attention

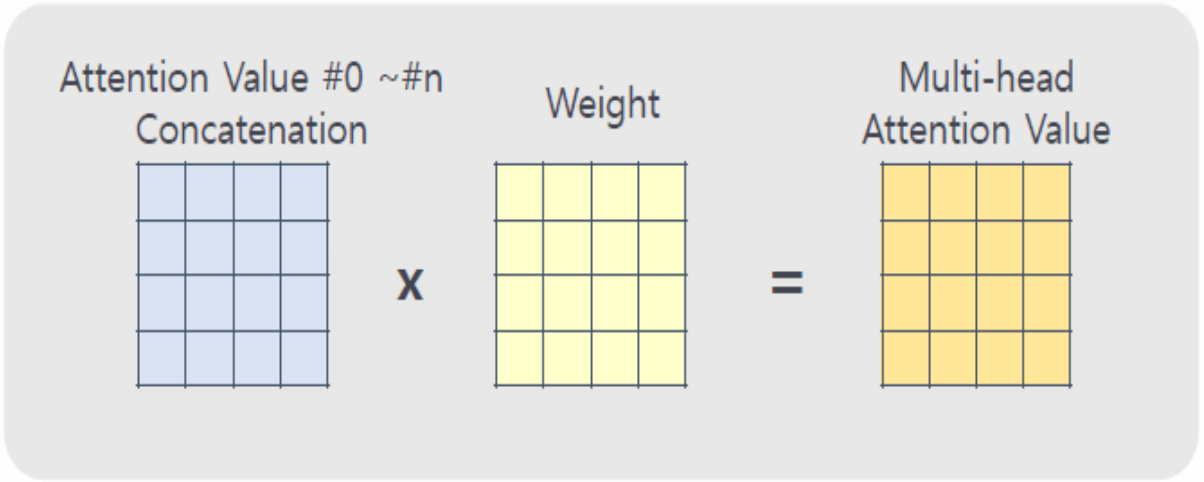
Attention을 병렬로 수행함으로써 다양한 관점에서 단어 간 관계 정보 파악이 가능

Input 문장에 대해 N개의 Query, Key, Value를 계산하기 위한 N개의 가중치(WQ, WK, WV)를 생성하여 병렬로 Self Attention 수행하고 N개의 Attention Value를 계산



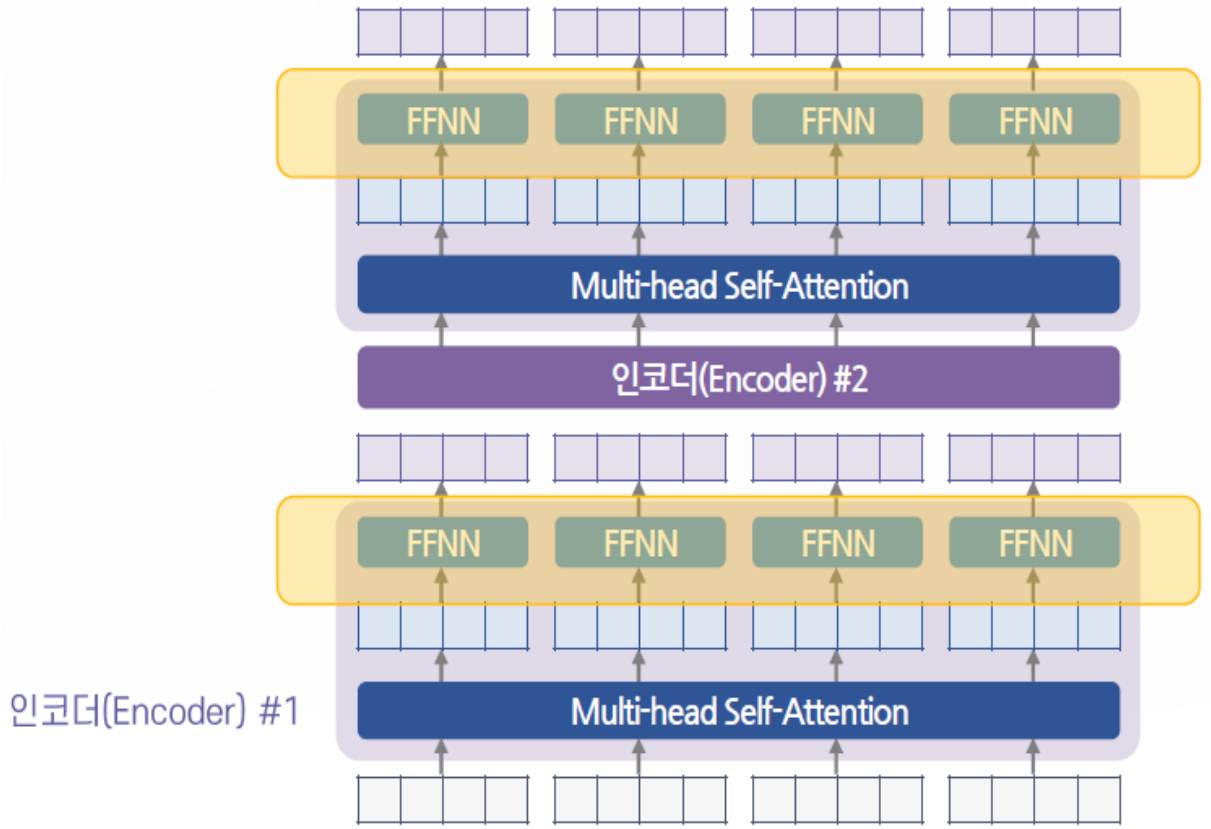
Multi-head Attention Value Matrix

Multi-head Attention 수행 결과를 연결하고 학습할 가중치를 곱하여 Multi-head Attention Value Matrix를 최종 결과로 도출



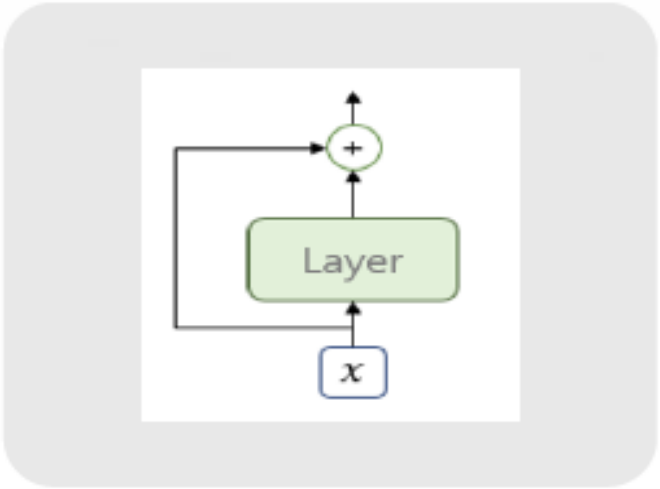
Position-wise FFN

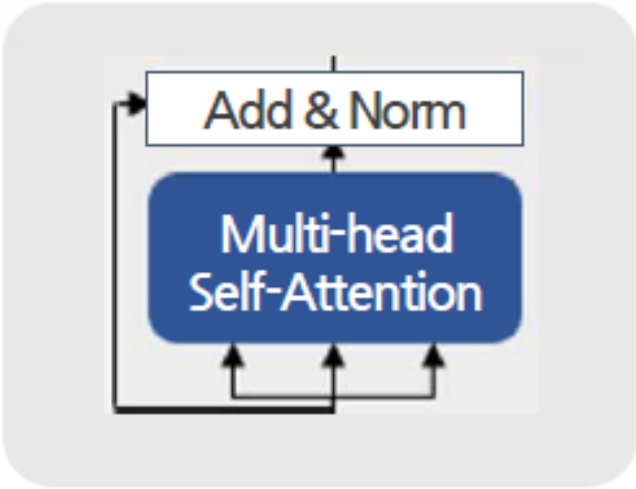
Position-wise FFN은 인코더와 디코더에서 공통으로 포함된 Sub-layer



잔차 연결

경사가 줄어드는 문제를 해결하기 위해 구글은 ResNet을 통해 잔차연결을 적용



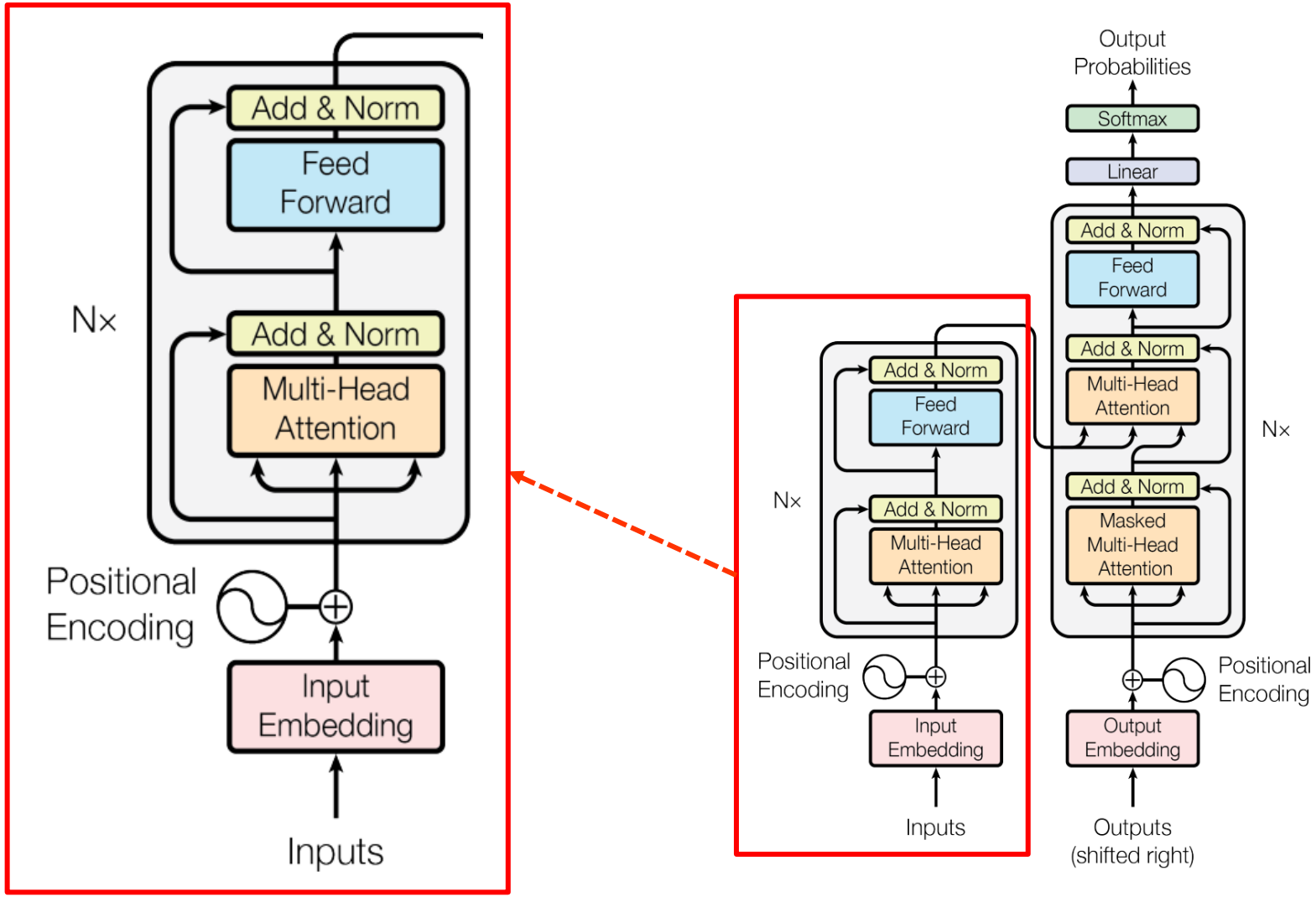


정규화

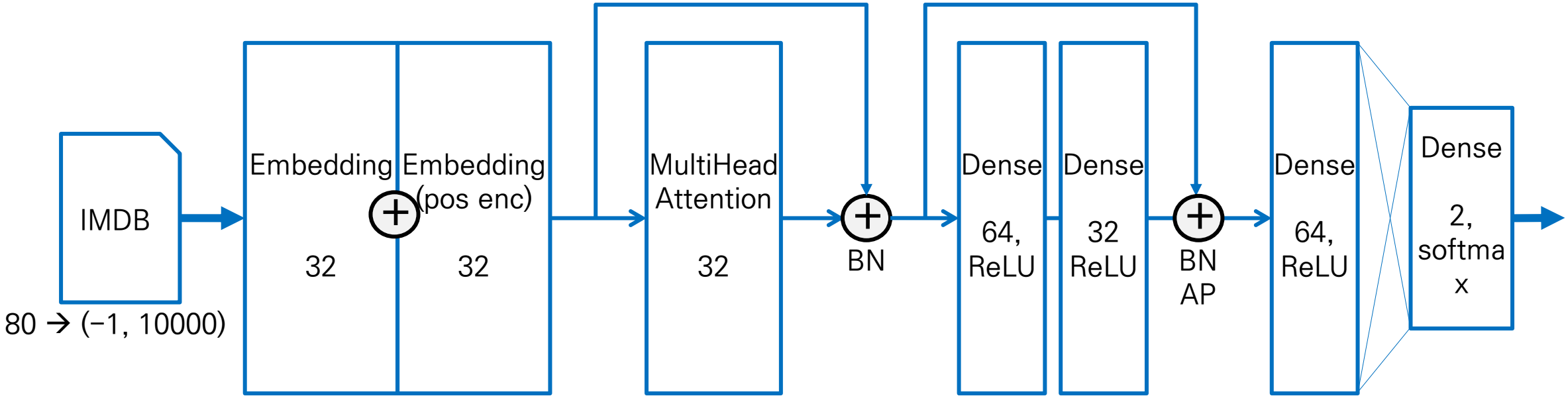
- Residual Connection으로 전달된 입력값과 Multi-head Attention Value를 더한 후 정규화(Normalization)를 수행

- Position-wise FFNN의 입력값으로 전달하기 전 정규화를 수행함으로써 과적합(Over-fitting)을 방지할 수 있음

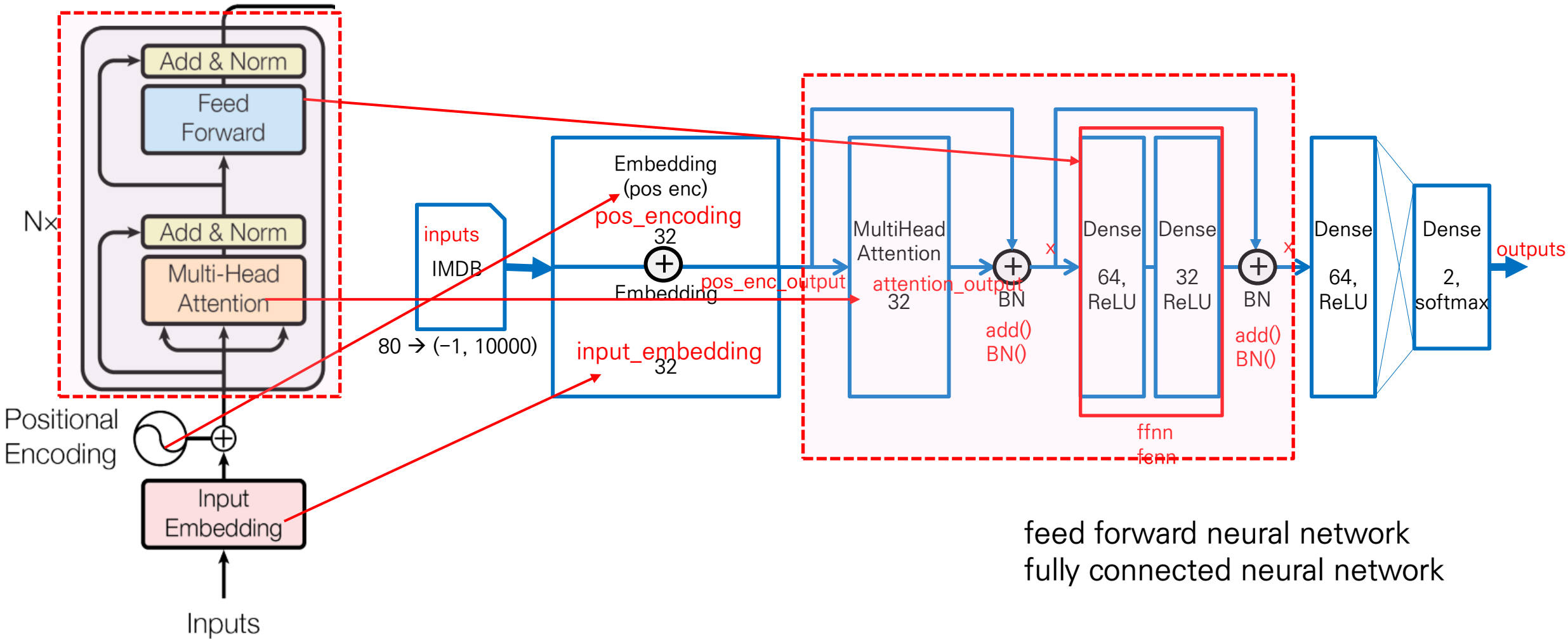
실습 – 트랜스포머의 인코더 블록을 이용한 영화평 분류



실습 – 트랜스포머의 인코더 블록을 이용한 영화평 분류 (모델 구조)



실습 – 트랜스포머의 인코더 블록을 이용한 영화평 분류 (모델 구조)



feed forward neural network
fully connected neural network

실습 – 트랜스포머의 인코더 블록을 이용한 영화평 분류 (모델 구조)

2장. 자연어처리 인공신경망 / 5절. 트랜스포머

```python
import tensorflow as tf
from tensorflow.keras import layers
from tensorflow.keras.models import Sequential, Model

inputs = layers.Input(shape=(80,))

input_embedding = layers.Embedding(input_dim=10000, output_dim=32)(inputs)
positions = tf.range(start=0, limit=80)
pos_encoding = layers.Embedding(input_dim=80, output_dim=32)(positions)
pos_enc_output = pos_encoding + input_embedding

attention_output = layers.MultiHeadAttention(num_heads=3, key_dim=32)(pos_enc_output, pos_enc_output)
x = layers.add([pos_enc_output, attention_output])
x = layers.BatchNormalization()(x)

ffnn = Sequential([layers.Dense(64, activation="relu"),
                   layers.Dense(32, activation="relu")])(x)
x = layers.add([ffnn, x])
x = layers.BatchNormalization()(x)
x = layers.GlobalAveragePooling1D()(x)
x = layers.Dropout(0.1)(x)

x = layers.Dense(64, activation="relu")(x)
x = layers.Dropout(0.1)(x)

outputs = layers.Dense(2, activation="softmax")(x)
model = Model(inputs=inputs, outputs=outputs)

model.summary()
```

`RNN 실습 – 영화평 분류` 예제 코드에서 모델만 수정하세요.

Model: "functional_1"

Layer (type)	Output Shape	Param #	Connected to
input_layer (InputLayer)	(None, 80)	0	-
embedding (Embedding)	(None, 80, 32)	320,000	input_layer[0][0]
add (Add)	(None, 80, 32)	0	embedding[0][0]
multi_head_attention (MultiHeadAttention)	(None, 80, 32)	12,608	add[0][0], add[0][0]
add_1 (Add)	(None, 80, 32)	0	add[0][0], multi_head_attention[…
batch_normalization (BatchNormalization)	(None, 80, 32)	128	add_1[0][0]
sequential (Sequential)	(None, 80, 32)	4,192	batch_normalization[0…
add_2 (Add)	(None, 80, 32)	0	sequential[0][0], batch_normalization[0…
batch_normalization_1 (BatchNormalization)	(None, 80, 32)	128	add_2[0][0]
global_average_pooling1d (GlobalAveragePooling1D)	(None, 32)	0	batch_normalization_1…
dropout_1 (Dropout)	(None, 32)	0	global_average_poolin…
dense_2 (Dense)	(None, 64)	2,112	dropout_1[0][0]
dropout_2 (Dropout)	(None, 64)	0	dense_2[0][0]
dense_3 (Dense)	(None, 2)	130	dropout_2[0][0]

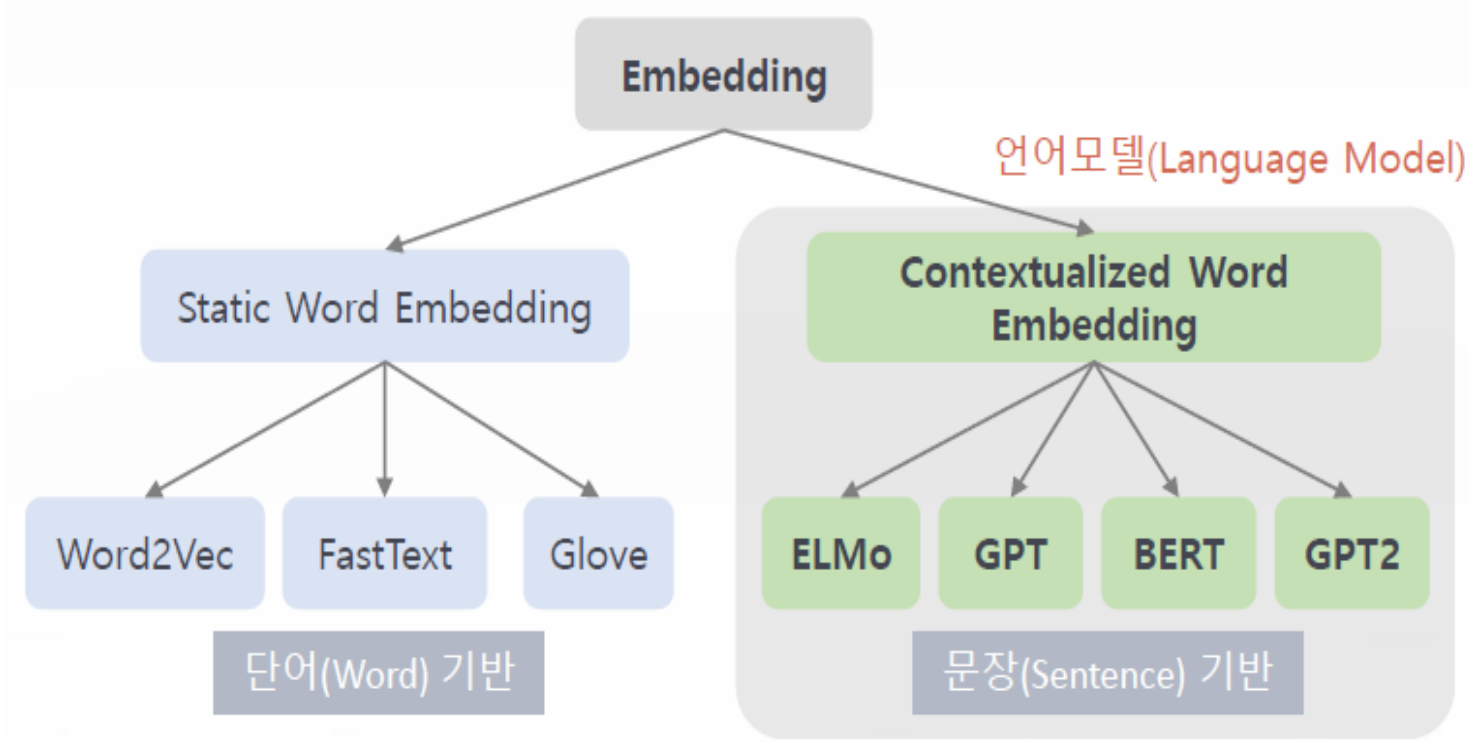
Total params: 339,298 (1.29 MB)
Trainable params: 339,170 (1.29 MB)
Non-trainable params: 128 (512.00 B)

6절. 언어 모델

2장. 자연어처리 인공신경망

언어 모델

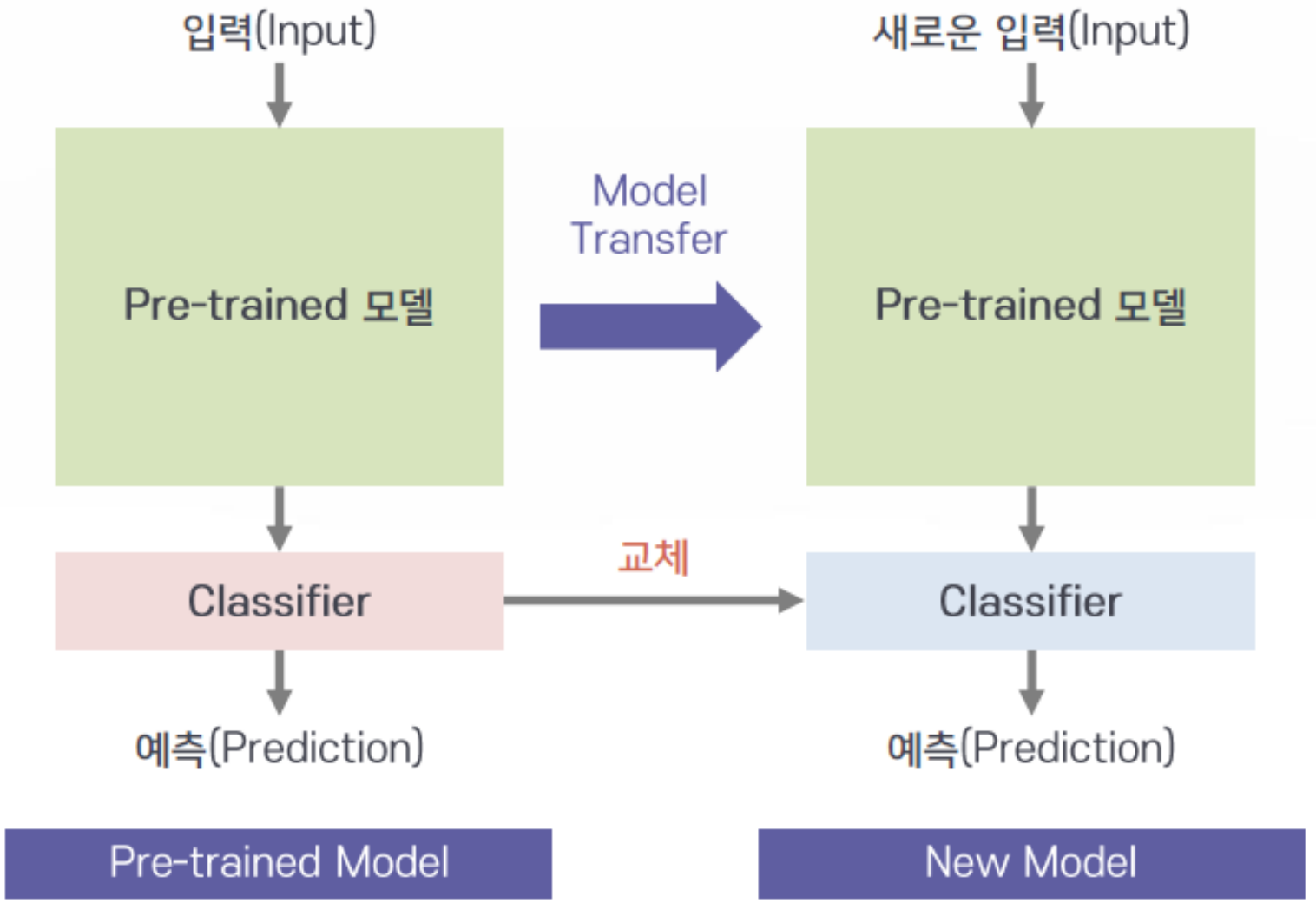
문맥을 고려하여 문장(Sentence)을 기반으로 임베딩을 수행하는 것



언어 모델

언어모델	발표 시기	모델 발표 기관
ELMo	2018년 2월	Allen AI와 Washington University
GPT-1	2018년 5월	Open AI
BERT	2018년 10월	Google
GPT-2	2019년 2월	Open AI
KoBERT	2019년 6월	ETRI
XLNet	2019년 7월	Carnegie Mellon University와 Google Brain
RoBERTa	2019년 7월	Facebook AI Research
ALBERT	2019년 9월	Google과 TTIC(Toyota Technological at Chicago)
T5	2019년 10월	Google
KoGPT-2	2020년 4월	SKT와 AWS
GPT-3	2020년 6월	Open AI
GPT-4	2023년	Open AI

전이학습


입력(Input)
Pre-trained 모델
Classifier
예측(Prediction)
Pre-trained Model
Model
Transfer
교체
새로운 입력(Input)
Pre-trained 모델
Classifier
예측(Prediction)
New Model

딥러닝을 활용한 자연어 처리