딥러닝팀 3 주차 클린업

1. 자연어

자연어(natural language)란 우리가 일상생활에서 사용하는 언어를 인공적으로 만들어진 인공어와 구분하여 부르는 개념입니다. 이런 사람의 언어를 컴퓨터가 이해하고 여러가지 일을 처리할 수 있도록 하는 일을 자연어 처리(Natural Language Processing, NLP)라고 합니다. 이런 nlp작업의 종류에는 글에서 긍정과 부정을 판단하는 감성 분석, Siri, Bixby와 같은 챗봇, Papago와 같은 기계 번역, 자동요약 등이 있습니다.

1.1. 자연어의 특징

자연어가 가진 여러가지 특징들이 있는데 이를 하나씩 알아보도록 하겠습니다.

- 순차성

자연어의 가장 대표적인 특징으로, 자연어는 순차적(Sequential)인 데이터입니다. 순차적 데이터란 순서가 달라질 경우 의미가 손상되는 데이터를 의미합니다. 예를 들면 "마감시간이 끝나기 전에 과제를 끝냈다." 라는 문장과 "과제가 끝나기 전에 마감시간이 끝났다."은 순서가 바뀌지만 의미는 무시무시할 정도로 달라지게 됩니다. DNN은 자연어가 갖는 순차적인 특징을 반영하기 힘들기 때문에 자연어 처리에는 적합하지 않은 모델입니다. 또한, 이번 주차에서 가장 중점으로 배울 RNN은 이런 자연어의 순차적인 특징을 반영할 수 있어 자연어에서 가장 많이 사용됩니다.

- 모호성

자연어는 모호하다는 특징이 있습니다. 자연어가 모호한 이유는 크게 표현의 중의성과 문장 내 정보 부족의 문제가 있습니다.

1) 표현의 중의성

"차를 마시러 공원에 가는 차 안에서 나는 그녀에게 차였다."

이 문장에서 '차'라는 단어는 여러 번 나오지만, 각 '차'가 의미하는 의미는 다 다릅니다. 이렇게 같은 단어가 다른 의미를 가지는 경우가 존재합니다.

2) 문장 내 정보 부족

"나는 철수를 안 때렸다."

이 문장은 1) 철수는 맞았지만, 때린 사람이 나는 아니다. 2) 나는 누군가를 때렸지만, 그게 철수는 아니다. 3) 나는 누군가를 때린 적도 없고, 철수도 맞은 적이 없다. 이렇게 세가지의 경우로 해석이 가능합니다.

언어는 효율성을 극대화하기 위해서 커뮤니케이션 과정에서 많은 정보가 생략되기도 합니다. 이런 상황에서 앞 뒤 상황이 없다면 이 문장을 해석하기는 굉장히 어려워집니다.

- 불연속성

이미지와 음성, 영상 등은 연속적인 데이터이지만, 자연어는 그렇지 못합니다. 예를 들어 8-bit 이미지는 각 픽셀의 값이 [0, 255] 내의 값을 가지게 되는데, 이때 (255, 0, 0)과 (254, 0, 0)인 픽셀은 모두 붉은 색 픽셀이라는 의미에서 유사성을 판단할 수 있습니다. 하지만 자연어의 경우 한 단어가 문장에서 가지는 의미는 결코 하나로 단정지을 수 없으며, 형태가 유사한 두 단어라도 완전히 다른 의미를 가질 수 있습니다. 이를 자연어의 불연속성이라고 할 수 있습니다.

1.2. 자연어의 전처리

텍스트 데이터를 모델의 입력으로 사용하기 위해서는 이런 특성을 반영할 수 있는 데이터의 변환 방법을 먼저 이해해야 할 것입니다. 지금부터는 이러한 텍스트 데이터를 모델에 사용하기 이전 전처리하는 과정에 대해 알아보도록 하겠습니다.

1.2.1. 정제

정제(Cleaning) 단계는 텍스트 데이터의 의미를 분석하는 것에 필요하지 않은 변수들을 제거하는 것입니다. 예를 들면, 영어의 경우 대문자와 소문자가 존재하는데, 이는 의미 분석에 전혀 영향을 주지 않습니다. "There is an apple"과 "there is an apple"에는 의미 차이가 존재하지 않기 때문입니다. 따라서, 정제 단계에서는 필요한 부분에 한해 대문자를 남기고 소문자를 제거하는 과정을 진행합니다. 이때 정제의 대상이 되는 전체 문장 데이터 집합을 코퍼스(Corpus)라고 합니다. 하지만 기계적으로 모든 대문자를 소문자로 바꿀 수는 없습니다. 예를 들면, US와 us는 각각 '미국'과 '우리'라는 의미로, 서로 다른 의미를 가지고 있기 때문입니다.

1.2.2. 토큰화

다음으로 토큰화(Tokenize) 단계는 의미를 가지는 최소 단위로 코퍼스를 잘게 쪼개는 단계입니다. 이때 의미를 가지는 최소 단위를 토큰(Token)이라고 부릅니다. 의미를 가지는 최소 단위는 목적에 따라, 그리고 정의에 따라 달라질 수 있기 때문에 아래의 그림처럼 한 문장이라도 여러 기준에 따라 토큰화가 가능합니다. 하지만 반대로 이야기하면 그 기준이 명확히 정해져 있지 않은 만큼, 필요에 따라 그 방법을 잘 선택해야 함을 의미하기도 합니다.

실제로 Python 라이브러리에서는 어떤 기준으로 영어 토큰화를 하는지 알아보겠습니다.

- nltk - word_tokenize()

```
from nltk.tokenize import word_tokenize

print(word_tokenize("Don't be fooled by the dark sounding name, Mr. Jone's Orphanage is as cheery as cheery goes for a pastry shop."))

['Do', "n't", 'be', 'fooled', 'by', 'the', 'dark', 'sounding', 'name', ',', 'Mr.', 'Jone', "'s", 'Orphanage', 'is', 'as', 'cheery', 'as', 'cheery', 'goes', 'for', 'a', 'pastry', 'shop', '.']
```

nltk 라이브러리의 word_tokenize() 함수를 사용한 토큰화 예시입니다. 가장 기본적인 토큰화 함수라고 할 수 있으며, 이 함수는 띄어쓰기 단위(Space)와 구두점(Punctuation)을 기준으로 토큰화를 진행합니다.

- nltk - WordPunctTokenizer()

```
from nltk.tokenize import WordPunctTokenizer

print(WordPunctTokenizer().tokenize("Don't be fooled by the dark sounding name, Mr. Jone's Orphanage is as cheery as cheery goes for a pastry shop."))

['Don', "'", 't', 'be', 'fooled', 'by', 'the', 'dark', 'sounding', 'name', ',', 'Mr', '.', 'Jone', "'", 's', 'Orphanage', 'is', 'as', 'cheery', 'as', 'cheery', 'goes', 'for', 'a', 'pastry', 'shop', '.']
```

동일한 라이브러리의 WordPunctTokenizer() 함수를 사용한 토큰화 예시입니다. 이 함수는 모든 구두점 (Punctuation)을 기준으로 토큰화를 진행합니다. 첫번째 예시와 달리 " ' "가 따로 분리되어 있음을 확인할 수 있습니다.

- Keras – text_to_word_sequence()

```
from tensorflow.keras.preprocessing.text import text_to_word_sequence

print(text_to_word_sequence("Don't be fooled by the dark sounding name, Mr. Jone's Orphanage is as cheery as cheery goes for a pastry shop."))

["don't", 'be', 'fooled', 'by', 'the', 'dark', 'sounding', 'name', 'mr', "jone's", 'orphanage', 'is', 'as', 'cheery', 'as', 'cheery', 'goes', 'for', 'a', 'pastry', 'shop']
```

Keras 의 text_to_word_sequence() 함수를 활용한 토큰화 예시입니다. Keras 는 Pytorch 와 Tensorflow 와 더불어 많이 사용되는 딥러닝 라이브러리입니다. 이 경우 대문자가 모두 소문자로 일괄 변환된 것을 확인할 수 있고, 쉼표와 마침표가 모두 사라졌음을 확인할 수 있습니다.

총 3 가지의 토큰화 라이브러리 예시에서 살펴본 바와 같이, 라이브러리마다, 그리고 함수마다 그 기준이 조금씩 다릅니다. 따라서, 필요에 따라 적절히 골라 사용해야 합니다. 다음은 한국어의 특성과 함께 한국어 전처리 방법을 살펴보도록 하겠습니다.

한국어는 영어보다도 토큰화가 까다롭습니다. 그 이유는 한국어가 교착어이기 때문입니다. 교착어란, 어간에 접사가 붙어 단어를 이루고, 의미와 문법적 기능이 더해지는 언어의 유형을 의미합니다. 예를 들어, 어간 '가다'라는 단어의 어간은 '가-'입니다. 여기에 과거의 의미를 더하는 접사 '-ㅆ다'가 더해져 '갔다'라는 단어가 완성됩니다. 이러한 한국어의 형태 때문에 한국어는 영어처럼 단순히 칼로 자르듯이 구간을 나눌 수 없습니다. 따라서, 한국어는 일반적으로 형태소(Morpheme) 단위로 토큰화를 진행합니다. 한국어 토큰화 과정은 별도의 설명 대신 예시만 살펴보겠습니다.

```
from konlpy.tag import Hannanum

nanum = Hannanum()

nanum.morphs("드디어 교안 다 썼다!")

['드디어', '교안', '다', '쓰', '었다', '!']
```

```
from konlpy.tag import Kkma

kkma = Kkma()

kkma.morphs("드디어 교안 다 썼다!")

['드디어', '교안', '다', '쓰', '었', '다', '!']
```

1.2.3. Subword Segmentation

마지막 Subword Segmentation 단계는 의미 분석의 일반화를 위해 단어를 더 작은 단위로 분해하는 과정입니다. 이는 아무리 많은 데이터를 학습시켜도 세상에 존재하는 모든 단어를 학습하는 것은 불가능한 일이기 때문에, 학습된 단어의 일부분으로 유사한 단어의 의미를 예측할 수 있도록 하는 것을 목표로 합니다. 하지만, 모델이 학습하지 못한 단어(Out-of-Vocabulary)로 인해 문제가 발생하곤 하는데, 이를 OOV 문제라고 합니다. 이를 해결하기 위해 Subword Segmentation 이 필요한 것이며, 구글에서 공개한 Sentencepiece 라는 라이브러리로 이용이 가능합니다.

1.3. Word Vector

1.3.1. One-hot-vector

전통적인 NLP 에서는 단어의 의미를 이산적으로 생각했습니다. Book 과 Books 와 같은 단어의 변형 형태도 다른 단어로 간주합니다. 이산적인 단어의 의미를 표현하기 위해서 만들어진 것이 바로 단어 집합(vocabulary)입니다. 단어 집합은 서로 다른 단어들을 중복을 허용하지 않고 모아놓은 집합을 의미합니다. 이렇게 단어 집합을 만들게 된다면, 이제 우리는 One-hot-vector 의 형태로 단어를 표현할 수 있게 됩니다. One-hot-vector 란 N 개의 단어를 각각 N 개 차원의 벡터로 나타낸 것을 말하고, 이렇게 만드는 것을 One-hot-encoding 이라고 합니다. 여기서 표현하고 싶은 단어의 index 에는 1 을 넣고 나머지는 0 을 넣게 됩니다. 이렇게 벡터 또는 행렬의 값이 대부분 0 으로 표현되는 방법을 희소표현(sparse representation)이라고 하고, One-hot-vector 는 희소 벡터입니다.

예를 들어 hotel 과 motel 이라는 단어가 있다면, 이를 One-hot-vector 로 다음과 같이 표현할 수 있습니다.

hotel = [0 0 0 1 0 0 0 0 0 0] motel = [0 0 0 0 0 0 1 0 0 0]

하지만, 우리가 웹사이트에서 "Seoul hotel"을 검색하게 된다면, 우리는 "Seoul motel"에 대한 정보까지 찾기를 원할 것입니다. 하지만, 위의 벡터와 같이 표현했을 때 hotel 과 motel 은 서로 orthogonal 하기 때문에 유사성을 찾을 수가 없습니다. 단어의 의미를 다룰 수가 없다는 의미입니다. 또한, 위에서 설명했듯이 N 개의 단어가 있으면 N 개 차원을 가지게 되게 때문에 많은 단어를 표현하려고 하면 차원의 저주에 빠지게 되고, 유의미한 정보가 있는 공간은 아주 작기 때문에 공간적인 낭비가 심하다는 단점이 존재합니다.

1.3.2. Word Embedding

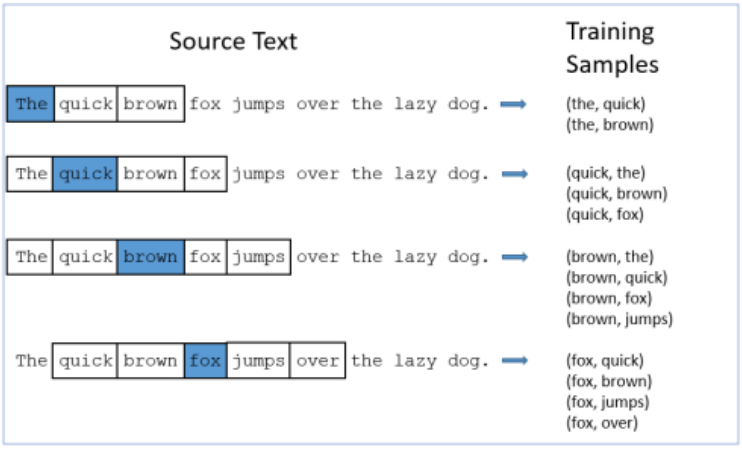
희소 표현과 반대되는 표현으로 밀집 표현이 있습니다. 밀집 표현은 벡터의 차원을 단어 집합의 크기로 상정하지 않습니다. 사용자가 설정한 값으로 모든 단어의 벡터 표현의 차원을 맞춥니다. 또한, 이 과정에서 더 이상 0 과 1 만 가진 값이 아니라 실수 값을 가지게 됩니다. 위의 hotel 과 motel 을 밀집 벡터로 표현하고, 차원을 4 로 설정한다면 다음과 같이 표현할 수 있습니다.

hotel = [0.7 0.3 0.1 0.5] motel = [0.5 0.1 0.3 0.6]

이렇게 단어를 밀집 벡터로 표현하는 방법을 Word Embedding 이라고 합니다. 그리고 이렇게 Word Embedding 과정을 거쳐 나온 밀집 벡터를 Embedding Vector 라고도 합니다. 밀집벡터의 각 요소는 단어의 실수로 표현되며 서로 다른 특징을 나타내게 됩니다. 물론, 각 요소가 어떠한 의미를 지니는지는 알 수 없지만, 텍스트를 단순히 구분하는 것이 아니라 의미적으로 정의한다고 볼 수 있습니다.

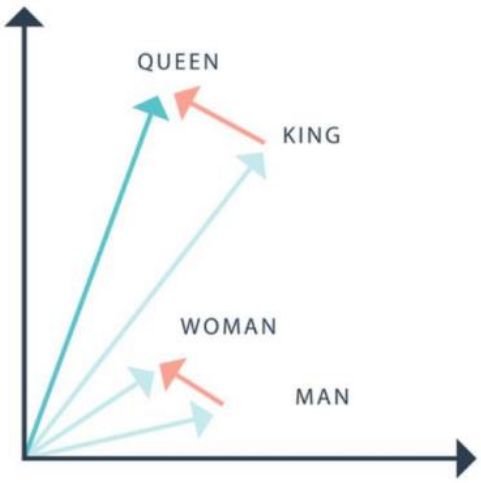
1.3.3. Word2Vec

이 Word Embedding 의 가장 대표적인 방법이 Word2Vec 입니다. Word2Vec 은 비슷한 위치에 등장하는 단어는 비슷한 의미를 가진다는 가정(분포 가설) 하에 단어를 벡터로 바꾸는 방법을 의미합니다. Word2Vec 에는 CBOW 와 Skip-gram 이라는 2 가지 방식이 존재합니다. CBOW 는 주변의 단어들을 입력으로 사용하여 중간에 있는 단어를 예측하는 방법이고, Skip-gram 은 중간에 있는 단어를 입력으로 사용하여 주변 단어들을 예측하는 방법입니다. 이중 Skip-gram 이 성능이 일반적으로 더 좋아 자주 사용되기에, Skip-gram 에 대해서만 설명하도록 하겠습니다.



Skip-gram 은 하나의 입력으로 여러 단어를 예측한다고 했던 만큼, 파란색을 입력으로 하여 주변 2n 개의 단어에 대한 예측을 수행합니다. 이때 학습에 사용되는 입력 단어 양 옆 단어의 개수 n 을 window size 라고 합니다. 예를 들면, brown 이라는 단어를 입력으로 할 때는 The, quick, fox, jumps 4 개의 단어를 개별적으로 예측 및 업데이트하게 되며, 이때의 window size 는 2 입니다. 이 방법을 통해 제한된 문장 안에서 여러 번 학습을 진행하여 효과적으로 임베딩 벡터를 만들 수 있게 됩니다. 궁극적으로 단어를 임베딩 벡터로 표현했을 때 얻을 수 있는 장점은 각 단어가 고차원 벡터공간의 한 점으로 표시됨으로써 벡터간 연산이 가능하다는 점입니다. 예를 들면 아래와 같은 단어간 연산이 가능해지게 됩니다.

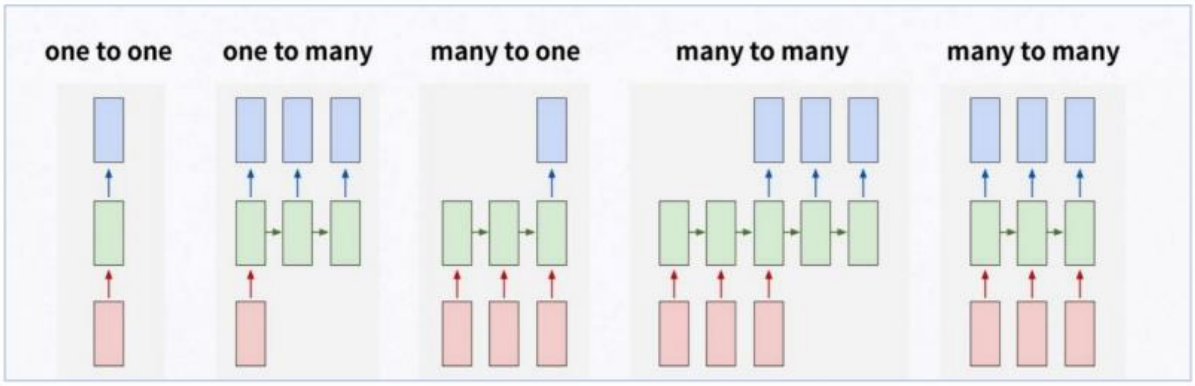
+ King	[0.3, 0.7, 0.1, 0.5]
- Man	[0.2, 0.2, -0.3, 0.5]
+ Woman	[0.6, 0.3, 0.2, 0.8]
≈ Queen	[0.7, 0.8, 0.6, 0.8]



2. RNN (Recurrent Neural Network)

2 주차에서는 CNN 에 대해서 알아보았다면, 3 주차에서는 RNN 에 대해 알아보도록 하겠습니다. RNN 을 주로 사용하는 영역은 텍스트, 음성 등이 있습니다. 텍스트, 음성에 CNN 을 사용하지 않는 이유는 텍스트와 음성은 입체적인 공간 정보보다는 시간정보나 순서정보가 훨씬 중요한 Sequential Data 이기 때문입니다. 그리고 RNN 은 이름에서 알 수 있듯이 이러한 sequential data 의 특성을 적절히 반영하여 학습할 수 있기 때문에 텍스트, 음성에서 사용하게 됩니다. 앞으로 RNN 의 가장 기초적인 형태의 Vanilla RNN, 그 단점을 보완한 LSTM 과 GRU 모델에 대해 알아보겠습니다.

구체적인 모델의 종류를 알아보기 전에, RNN 으로 어떤 종류의 문제를 해결할 수 있는지 알아보도록 하겠습니다.



먼저, One-to-One 모델은 바로 뒤에 공부할 가장 기본적인 RNN, Vanilla RNN 모델을 예시로 들 수 있습니다. 그리고 나머지 4 가지 구조에 해당하는 모델들은 모두 Vanilla RNN 을 목적에 맞게 변형한 형태의 모델들이라고 이야기할 수 있습니다.

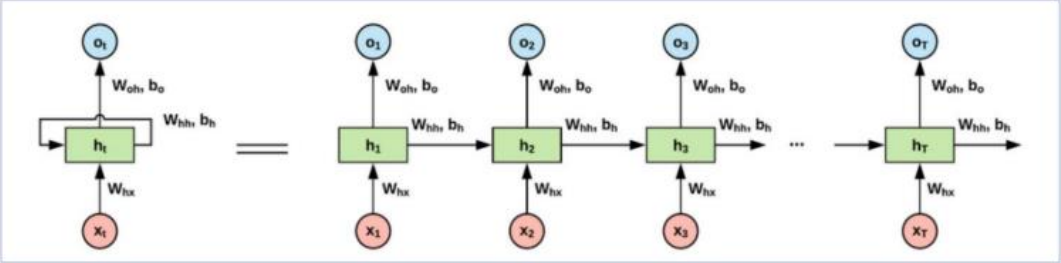
One-to-Many 의 구조를 가지는 RNN 모델은 하나의 입력에 대해 여러 시점의 출력을 반환합니다. 대표적인 예시로 이미지 설명(Image Captioning) 과제를 해결하기 위한 모델들이 이에 해당합니다. Many-to-One 은 반대로 다수의 입력에 대해 하나의 출력을 반환하는 모델로, 문장으로 이루어진 텍스트 데이터에 대한 감성 분석(Sentiment Classification)이 좋은 예시입니다.

Many-to-Many 는 두 가지 형태로 나눌 수 있습니다. 둘 중 왼쪽 도식에 해당하는 과제로는 기계번역(Machine Translation)이 있습니다. 단어의 연속이라고 볼 수 있는 하나의 문장 데이터를 입력하면, 그 전체 문장을 해석하여 다른 언어의 문장으로 반환하는 형태로 작동하기 때문입니다. 오른쪽의 형태는 한 시점의 입력에 대해 매번 출력을 반환하는 모델입니다. 영상 데이터와 관련한 과제, 그리고 실시간 오타 수정 등에 사용됩니다.

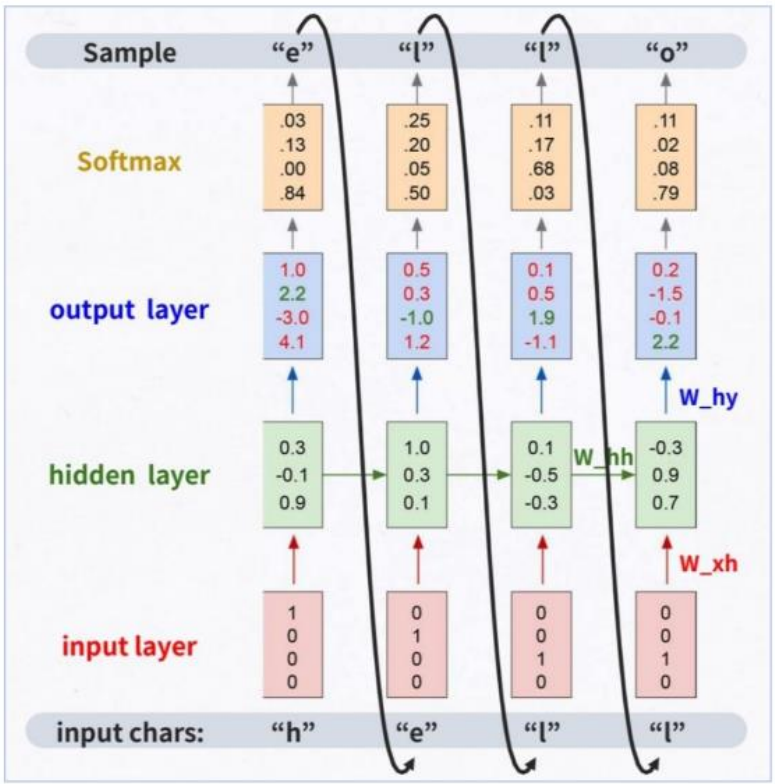
이렇듯 해결하고자 하는 과제에 따라 모델의 형태는 달라질 수 있지만, 사용되는 모델은 근본적으로 다르지 않습니다. 지금부터 대표적인 RNN 모델들을 알아보도록 하겠습니다.

2.1. Vanilla RNN

Vanilla RNN 은 RNN 의 가장 기본적인 형태입니다. RNN 의 가장 중요한 특징은 시간 순서의 데이터를 학습함에 있어 하나의 layer 를 반복해서 사용한다는 점입니다. 즉, 데이터의 다양한 특징들을 추출하기 위해 재사용되지 않던 Convolution Layer 와 달리 RNN 의 Hidden Layer 는 처음부터 마지막 시점 t의 데이터까지 반복해서 사용됩니다. 이 때 RNN 의 Hidden Layer 는 이전 시점의 데이터들을 기억하는 역할을 하므로 메모리셀(Memory Cell)이라고 합니다.



RNN 의 입력은 x_t로 표현합니다. 전체 데이터를 t시점들로 나누어 모델에 넣기 때문입니다. 따라서, Vanilla RNN 은 간단하게 나타내면 왼쪽의 그림처럼 나타낼 수 있지만, 이를 펼쳐서 전체 데이터에 대해서 나타내면 오른쪽 그림으로도 나타낼 수 있습니다. 여기서 Memory Cell 의 값인 h_t는 Hidden State 라고 부르며, 각 시점의 입력에 따라 지속적으로 업데이트되며 어떤 데이터를 기억할지를 결정하게 됩니다. RNN 은 순서 정보를 다루는 모델이기 때문에 이전 시점의 정보들을 기억하는 h_t의 역할이 매우 중요하다고 할 수 있습니다. 간단한 예시를 통해 RNN 의 작동 원리를 살펴보도록 하겠습니다.



2.1.1. Hidden State

Hidden state 의 수식을 간단하게 표현하자면 다음과 같습니다.

$$h_t = f_W(h_{t-1}, x_t)$$

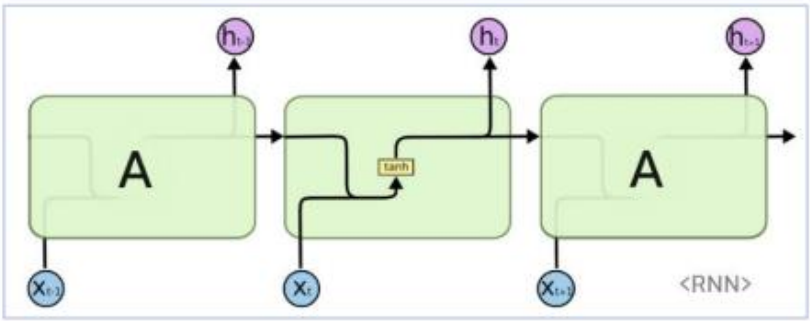
여기서 h_t는 new state, f_W는 파라미터 W를 가진 활성화 함수, h_{t-1}는 old state, x_t는 time step 의 input 을 말합니다. 즉, h_t라는 new state 는 전 단계들의 state 와 새로운 입력을 함수에 통과시켜서 얻은 값이라고 볼 수 있습니다. 이에 대해서 더 자세하게 알아보겠습니다.

Vanilla RNN 의 가중치는 크게 3 가지로, x_t에서 h_t사이의 W_x, h_t에서 h_{t+1}사이의 W_h, 그리고 h_t에서 o_t사이의 W_o입니다. 그리고 Vanilla RNN 을 비롯한 RNN 계열의 모델들은 대부분 tanh 함수를 활성화 함수로 이용합니다. tanh 함수는 절대값이 0 에서 1 사이인 output 을 가지기 때문에 다음 단계로 전달된 정보를 효과적으로 표현할 수 있습니다. 이에 반해, CNN 에서 자주 활용되는 ReLU 함수를 사용하게 될 경우, 순환 구조를 가지는 RNN 의 특성상 1 보다 큰 값이 나오게 되면 값이 발산할 수 있기 때문에 ReLU 함수를 사용하지 않습니다. 결과적으로 h_t의 업데이트와 각 시점의 output 반환은 다음처럼 이루어집니다.

$$h_t = \tanh\left(W_x \times x_t + W_h \times h_{t-1}\right)$$

$$o_t = f(W_o h_t)$$

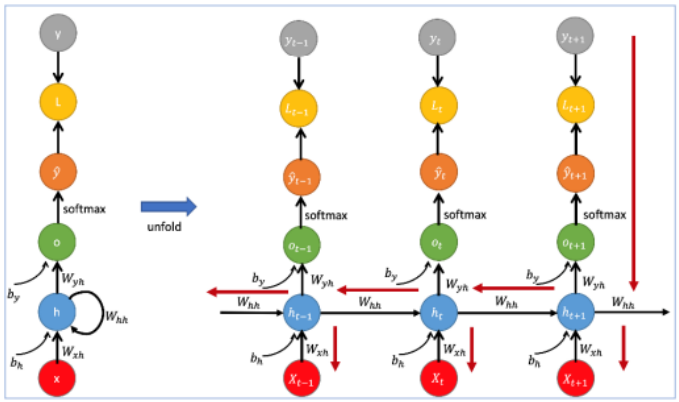
이 수식을 보면, h_t가 어떻게 이전 시점의 정보들을 기억할 수 있는지 의문이 들 수 있습니다. h_t의 수식을 통해 추상적으로 이해해보자면, h_t는 현 시점의 입력 x_t와 가중치의 곱, 그리고 이전 시점의 h_{t-1}과 가중치의 곱을 합한 후 tanh 함수에 통과시킵니다. 이는 이전 hidden state 를 일정 부분 반영하는 동시에 현 시점의 입력을 일정 부분 반영하는 것으로 이해할 수 있습니다. 이를 간단하게 도식화하면 아래와 같이 나타낼 수 있습니다.



2.1.2. 역전파 (Back Propagation)

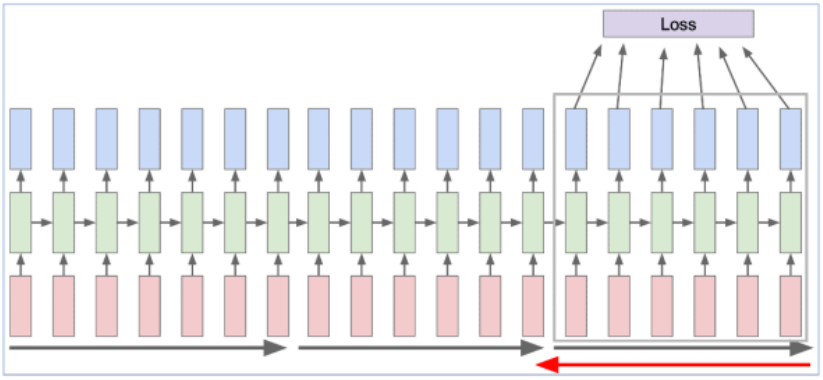
RNN 모델의 역전파는 여러가지 형태로 이루어집니다. 해당 모델로 해결하고자 하는 문제에 따라, 그리고 Sequential Data 의 크기 등에 따라 다르게 정의됩니다. 오늘은 크게 2 가지 방법에 대해서만 간단하게 알아보도록 하겠습니다.

- BPTT



RNN 의 역전파 방법은 BPTT 라고 부르는 형태로 이루어집니다. Many-to-One, One-to-Many 등 RNN 모델의 입출력 구조에 따라 형태가 다르겠지만, BPTT는 매 시점마다 출력이 반환되는 형태의 RNN 모델에서 활용됩니다. 매 시점마다 출력이 나온다는 뜻은 매 시점마다 loss 역시 계산할 수 있고, 이는 gradient 가 매 시점마다 존재할 수 있다는 뜻입니다. 따라서, BPTT 는 매 시점마다 gradient 를 계산하여 첫번째 시점의 입력까지 역전파를 진행합니다.

- Truncated BPTT



만약 Sequential Data 의 길이가 길 경우, 매 시점의 출력이 나올 때마다 첫번째 시점의 입력까지 모든 가중치를 업데이트하게 되면, 데이터의 후반부로 갈수록 연산에 소요되는 시간이 점점 길어질 것입니다. 예를 들면, 문장을 입력 데이터로 활용할 경우, 문장의 길이가 길다면 가중치 업데이트 시간이 오래 걸려 매우 비효율적일 것입니다. 이를 보완하기 위해 만들어진 truncated BPTT 는 Sequential Data 를 일정한 구간으로 끊어 구간마다 BPTT 를 진행하는 형태의 역전파 방법입니다.

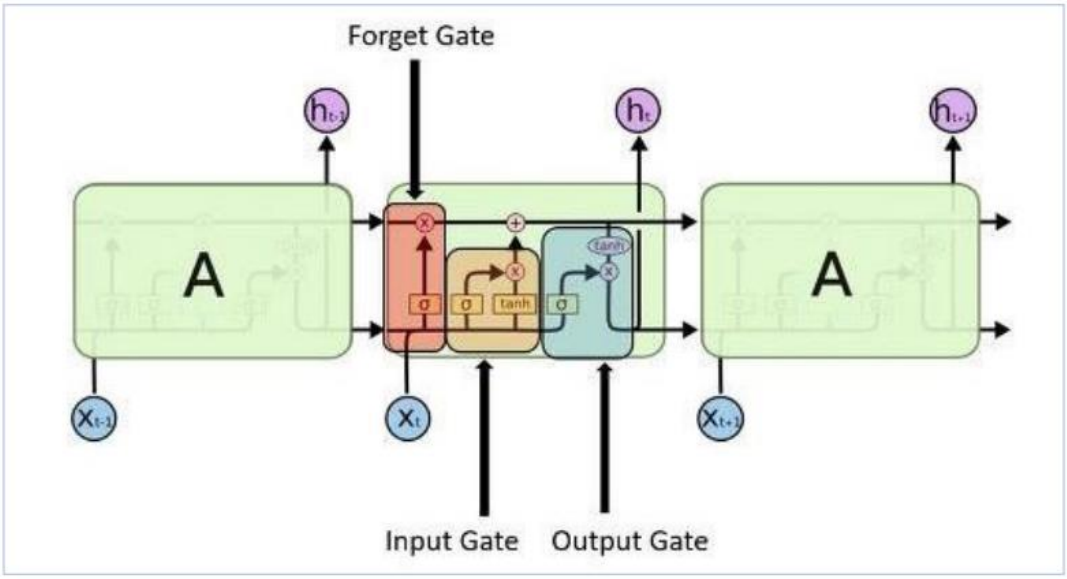
2.2. LSTM 과 GRU

Vanilla RNN 은 Sequential Data 에 대해 딥러닝을 적용할 수 있다는 큰 장점이 있지만, 데이터의 길이가 길어질수록 취약한 모습을 보인다는 단점이 있습니다. 이는 순전파시에 은닉층의 값에는 활성화함수인 tanh 가 계속해서 곱해지는데, 이 값이 (-1,1)이기 때문에 결과값이 점점 작아지는 문제가 발생하기 때문입니다. 이것을 장기기억을 제대로 처리하지 못하는 문제점, 즉 장기 의존성 문제(Long-Term Dependency Problem)이라고 합니다. 이러한 문제점을 해결하기 위해 제안된 모델이 LSTM 모델이며, LSTM 의 단점들을 일부 보완한 형태의 모델이 GRU 모델입니다.

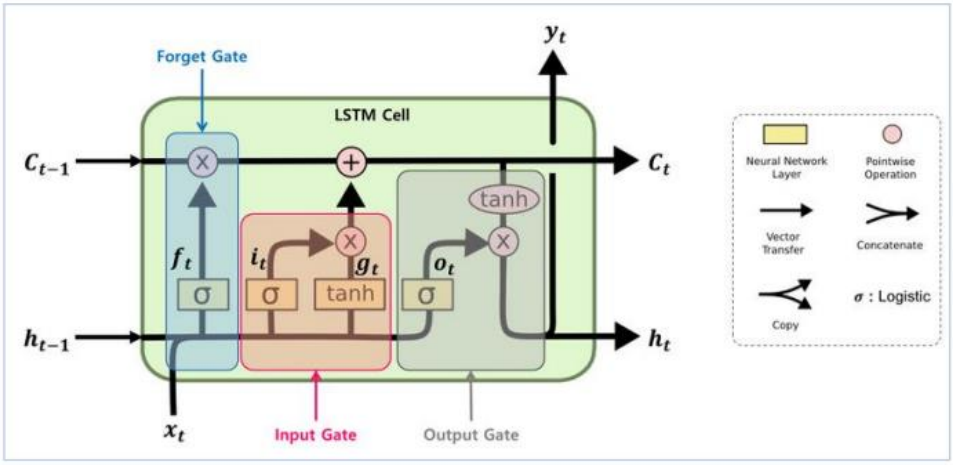
2.2.1. LSTM (Long Short-Term Memory)

LSTM 은 사람의 기억이 장기 기억과 단기 기억으로 나누어져 있다는 점에서 착안한 모델입니다. Vanilla RNN 의 경우 h_t가 데이터 전반의 정보들을 기억했다면, LSTM 에서는 h_t가 단기 기억을, c_t가 장기 기억을 담당하는 형태로 이루어져 있습니다. 여기서 h_t는 Hidden State, c_t는 Cell State 라고 부릅니다. c_t는 LSTM 에서 새로 등장하는 값으로 많은 연산을 거치지 않고 다음 time-step 으로 정보를 그대로 전달해줍니다.

그리고 각각의 State 에 어떤 값을 저장할지 정해주는 3 가지 Gate 가 존재합니다. 각 Gate 들에 대해서 알아보면서 Hidden state 와 Cell state 에 어떤 값들이 어떻게 저장되는지도 확인해보도록 하겠습니다.



2.2.2. Hidden State 와 Cell State



Forget Gate 는 과거의 정보를 얼마나 잊을지 결정하는 역할을 합니다. 화살표를 따라 이해해보면, 이전 시점의 Hidden State 인 h_{t-1}과 현재 시점의 입력인 x_t를 입력 받아 가중치 W_f와 곱하여 시그모이드를 통과시킨 후 이 값을 Cell State 로 넘겨주게 됩니다. Input Gate 는 현재의 정보를 얼마나 기억할지 결정하는 역할을 합니다. Input gate 역시 이전 시점의 Hidden State 인 h_{t-1}과 현재 시점의 입력인 x_t를 입력 받아 가중치 W_i와 곱하여 시그모이드를 통과시킨 후 이를 g_t로 보내게 됩니다. 마지막으로 Output Gate 는 현재 시점의 Cell State 와 이전 시점의 정보들을 바탕으로 현재 시점의 Hidden State 를 결정하는 역할을 합니다. 화살표로 보면, 현재까지의 정보들 중 어떤 정보들을 얼마나 활용할 것인지 정하게 됩니다.

이들을 각각 수식으로 나타내면 다음과 같습니다.

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$$

각 수식을 살펴보면 활성화 함수로 시그모이드 함수가 들어있는 것을 확인할 수 있습니다. 1 주차에 배웠던 시그모이드 함수를 다시 한번 생각해보면, 시그모이드 함수는 (0,1) 사이의 값을 가지고 있습니다. 이를 해석해보자면 시그모이드 함수를 '얼마나 반영할지?'로 생각할 수 있다는 뜻입니다. 시그모이드 함수의 값이 0 에 가까울수록 적게, 1 에 가까울수록 많이 반영할 수 있다고 생각할 수 있습니다. 다시 한번 각 게이트의 설명을 가져와 보겠습니다.

- Forget Gate: 과거의 정보를 **얼마나** 잊을지 결정하는 역할
- Input Gate: 현재의 정보를 **얼마나** 기억할지 결정하는 역할
- Output Gate: 현재까지의 정보들 중 어떤 정보들을 **얼마나** 활용할 것인지 정하는 역할

여기서 '얼마나'에 해당하는 함수를 시그모이드가 반영한다고 생각하시면 될 것 같습니다.

Input Gate 에서 i_t는 현재의 정보를 얼마나 기억할지 결정하는 역할이었다면, g_t는 현재의 정보 중 어떤 정보를 Cell State 에 전달할 것인지 결정하는 역할을 맡습니다. 어떤 정보를 표현하기 위해서는 시그모이드 함수가 아닌, (-1,1) 범위를 가진 tanh 함수를 사용합니다.

$$g_t = \tanh\left(W_g \cdot [h_{t-1}, x_t] + b_i\right)$$

지금까지 본 Forget Gate 와 Input Gate 의 출력을 바탕으로 현 시점의 Cell State 값이 정해지게 됩니다. 추상적으로 이야기하자면, Forget Gate 로부터 얼마나 이전 시점의 정보들을 잊을 것인지 결정하고, Input Gate 에서 현재 시점의 정보들 중 어떤 정보들을 얼마나 기억할지 결정하여 이를 이전 Cell State 의 값에 대해 반영하는 것입니다. 이를 수식으로 나타내면 아래와 같습니다.

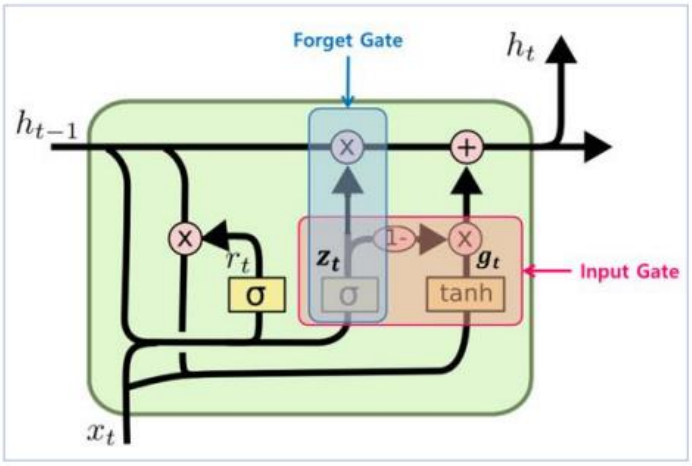
$$C_t = f_t \times C_{t-1} + i_t \times g_t$$

그리고 최종적으로 Output Gate 의 값과 Cell State 의 값을 사용하여 현재 시점의 Hidden State 값을 결정하게 됩니다.

$$h_t = o_t \times \tanh\left(C_t\right)$$

정리하자면 Forget Gate 와 Input Gate 에서 이전 시점의 Hidden State 와 현 시점의 입력을 바탕으로 현 시점의 Cell State 를 정의하고, 이 정의된 Cell State 와 이전 시점의 정보들을 활용하여 현 시점의 Output 이자 현 시점의 Hidden State 를 결정하게 되는 것입니다.

2.2.3. GRU (Gated Recurrent Unit)

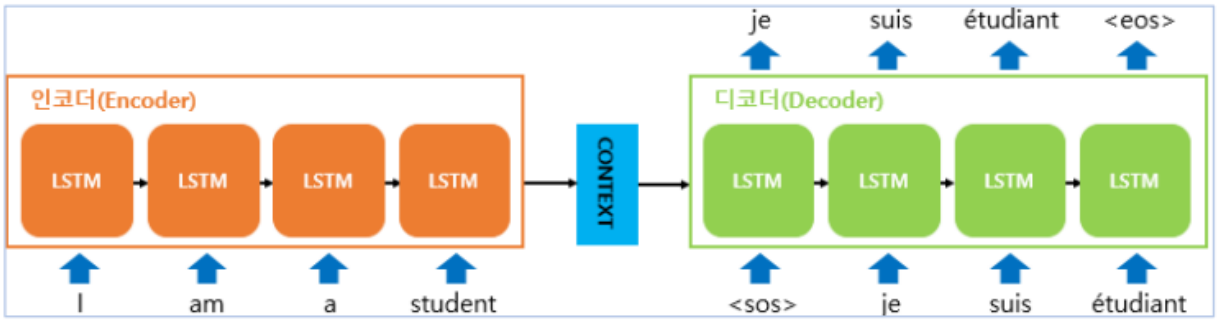


GRU 는 기본적으로 LSTM 의 원리와 유사한 과정으로 작동합니다. 하지만, Cell State 가 없어졌다는 점, 그리고 Output Gate 가 따로 없다는 점에서 차이가 있습니다. 간단하게 말하자면, 위 그림의 z_t는 h_{t-1}, x_t, 그리고 시그모이드 함수를 활용하여 (0, 1) 범위의 값을 반환하는 Gate Controller 로써 역할을 수행합니다. z_t가 1 에 가까우면 Forget Gate 가 Input Gate 보다 활성화되어 새로운 Hidden State 의 값이 이전 Hidden State 와 유사한 값으로 정의되도록 하며, 0 에 가까우면 Input Gate 가 Forget Gate 보다 활성화되어 새로운 Hidden State 의 값이 현재 시점의 입력을 많이 반영하도록 합니다. GRU 는 LSTM 의 Cell State 를 제거했다는 점에서 파라미터 수가 LSTM 보다 적다는 장점이 있습니다. 이로 인해 연산 속도가 더 빠르다는 장점 또한 존재합니다. 하지만, 실제로 사용해보면 성능에 있어서 매우 큰 차이가 존재하지는 않습니다.

3. RNN 모델의 응용

앞서 보았듯이 RNN 모델은 One-to-One 부터 Many-to-Many 까지 다양한 구조로 사용이 가능합니다. 지금부터는 Vanilla RNN, LSTM, 그리고 GRU 등과 같은 기본적인 형태의 RNN 모델들이 어떻게 다양한 분야에서 활용되고 있는지 알아보도록 하겠습니다.

3.1. RNN 을 이용한 Encoder-Decoder



Encoder 와 Decoder 의 개념은 원래 논리회로에서 등장하는 개념입니다. 예시를 활용하여 Encoder 와 Decoder 를 설명하자면, Encoder 는 영상 신호를 이진수인 디지털 신호로 변환하여 압축하는 역할을, Decoder 는 디지털 신호를 다시 영상으로 변환하는 역할을 합니다. Encoder 는 변환, Decoder 는 역변환을 담당하는 것으로 볼 수 있는 것입니다.

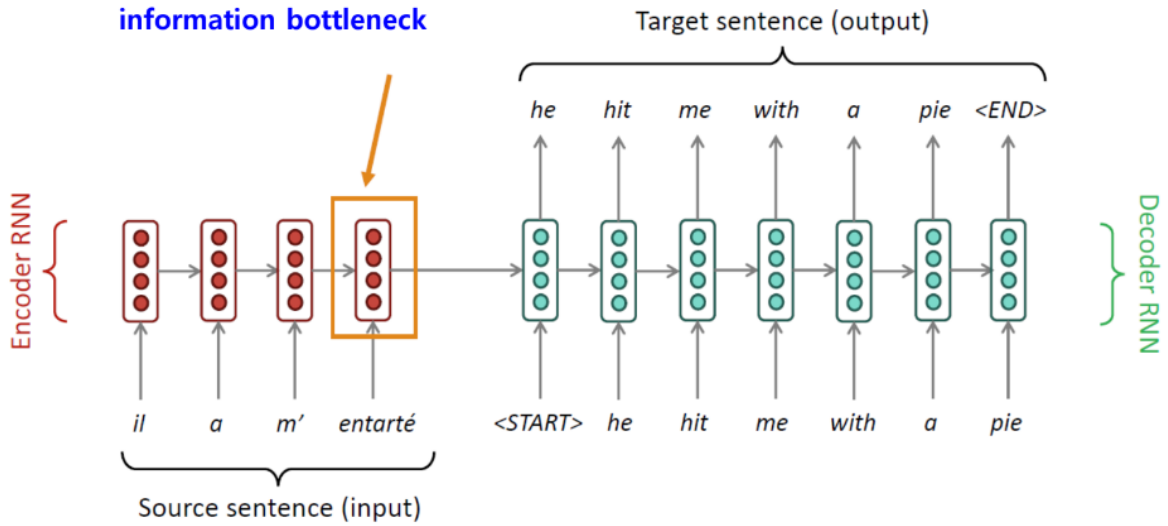
이런 Encoder 와 Decoder 각각에 서로 다른 RNN 계열 모델을 넣어 만든 모델이 바로 RNN 을 이용한 Encoder-Decoder 입니다. 이러한 구조는 입력과 출력의 길이가 달라도 된다는 점 때문에 번역이나 텍스트 요약 과제에 사용되는 형태의 모델입니다.

한번 구체적으로 알아보도록 하겠습니다. Encoder 는 Sequential Data 를 입력 받아 압축된 하나의 벡터로 만드는 역할을 합니다. 위 그림에서는 Encoder 속 RNN 의 최종 Hidden State 값, 즉 h_n 을 압축된 벡터라고 볼 수 있으며, 이 벡터를 컨텍스트 벡터(Context Vector)라고 합니다. 이는 입력 문장의 의미를 하나의 벡터로 압축시킨 형태이며, Encoder 의 출력이자 동시에 Decoder 의 첫 Hidden State 로 사용됩니다.

Decoder 는 Encoder 의 마지막 Hidden State 와 입력을 바탕으로 Encoder 와 달리 매 시점마다 출력을 내보내게 됩니다. 이 때 사용되는 매 시점의 입력은 이전 시점의 출력인데, 첫번째 시점에는 이전 시점의 출력이 존재하지 않으므로 자연어 처리의 경우 <sos>라는 것을 사용하게 되며, 이를 sos(start of sentence) 토큰이라고 합니다. Decoder 를 학습하는 과정에서 Decoder 가 초반부에 예측을 잘못할 경우 Decoder 전체의 학습 과정을 저해하기 때문에 이를 방지하기 위해 Decoder 의 매 시점 출력과는 별개로 다음 시점의 입력은 실제 정답을 넣어주게 되는데, 이를 교사 강요(Teacher Forcing)이라고 부릅니다. 이 과정을 통해 Encoder 의 Hidden State 와 토큰을 활용하여 문장의 끝까지 도달하면 Decoder 는 토큰을 반환하여 문장이 끝났음을 알립니다.

3.2. Attention

~~팬심 아닙니다. 아무튼 아닙니다.~~



다시 한번 Encoder-Decoder 의 구조를 생각해보자면, Encoder 는 문장의 의미를 하나의 압축된 벡터, 즉 Context Vector 로 만들었습니다. 하지만, 이에는 크게 두가지 문제가 있습니다.

1. Context Vector, 즉 전체 Input 을 하나의 벡터 형태로 압축시킨 것이기 때문에, 이 과정에서 손실되는 정보가 생깁니다. (병목현상, Bottleneck problem)
2. Neural Network 기반 딥러닝 모델에서 층이 깊어질수록 Gradient Vanishing 문제가 발생할 수 있었습니다. 마찬가지로 RNN 에서는 Cell 이 늘어날수록 Gradient Vanishing 문제가 발생할 수 있습니다. 이를 최대한 방지하기 위해 RNN 에서는 LSTM 형태의 Cell 을 채택하지만, 그럼에도 장기의존성 문제는 발생할 수 있습니다.

Encoder-Decoder 의 이러한 문제들을 해결하는 방법으로 제시된 것이 바로 Attention 입니다.

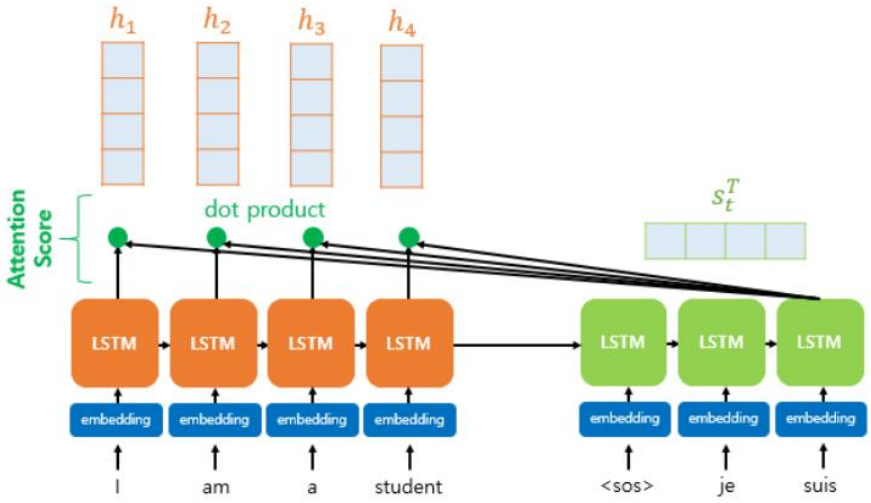
Attention 의 개념을 간단하게 설명하기 위해서 한 예시를 들어보도록 하겠습니다. 우리가 검색엔진에 어떤 내용을 검색을 했을 때, 정확하게 그 내용에 해당하는 값들이 있는 경우에는 그 내용이 출력이 될 것입니다. 하지만, 정확하게 그 내용에 해당하는 값이 없을 때에도 검색엔진은 그와 비슷한 내용으로 내용을 출력을 합니다. 다시 말해서 우리가 찾고자 하는 값과의 유사도를 바탕으로 얼마나 그 내용에 집중해야 하는지를 정하는 것을 Attention 이라고 합니다. 여러가지 Attention 기법이 존재하지만, 그 중에서 내적을 이용한 Attention 의 원리와 방법에 대해 알아보도록 하겠습니다.

우리는 두 벡터 x와 y에 대해서 내적을 다음과 같이 정의할 수 있습니다.

$$x \cdot y = |x||y| \cos\theta$$

즉, x와 y의 내적은 두 벡터의 크기와 사잇값의 곱으로 나타낼 수 있고, 여기서 우리는 코사인 유사도를 사용할 수 있습니다. 여기서 코사인 유사도란 내적공간의 두 벡터간 각도의 코사인값을 이용하여 측정된 벡터간의 유사한 정도를 의미합니다. 따라서 내적을 사용하여 우리는 두 벡터의 유사도를 파악할 수 있는 것입니다. 이 내적 방법을 이용하여 Attention Score 를 계산해보도록 하겠습니다.

① Attention Score 계산

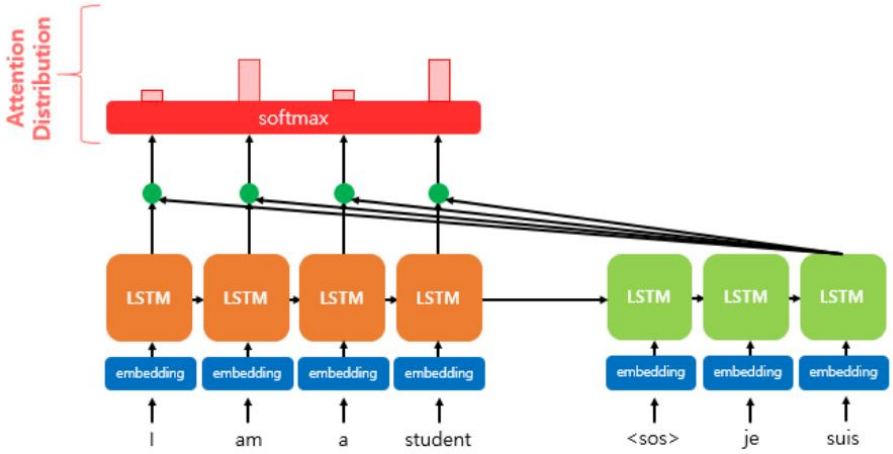


Encoder 의 각 시점 Hidden State 를 h_t, Decoder 의 각 시점 Hidden State 를 s_t로 표기하도록 하겠습니다. 이 때 Attention 기법이 의도하는 것은, Decoder 의 매 시점에 출력을 반환하기 전에 Encoder 의 모든 시점 Hidden State 들을 참고함으로써 현 시점의 s_t와 유사한 Encoder 의 Hidden State 값들을 출력에 반영하는 것입니다. 이를 위해 s_t와 Encoder 의 모든 시점 Hidden State 의 유사도를 구하는 과정이 필요한 것입니다. 이 유사도를 Attention Score 라고 부릅니다. 수식은 다음과 같습니다.

$$score(s_t, h_i) = s_t^T h_i$$

$$e^t = [s_t^T h_1, s_t^T h_2, \cdots, s_t^T h_n]$$

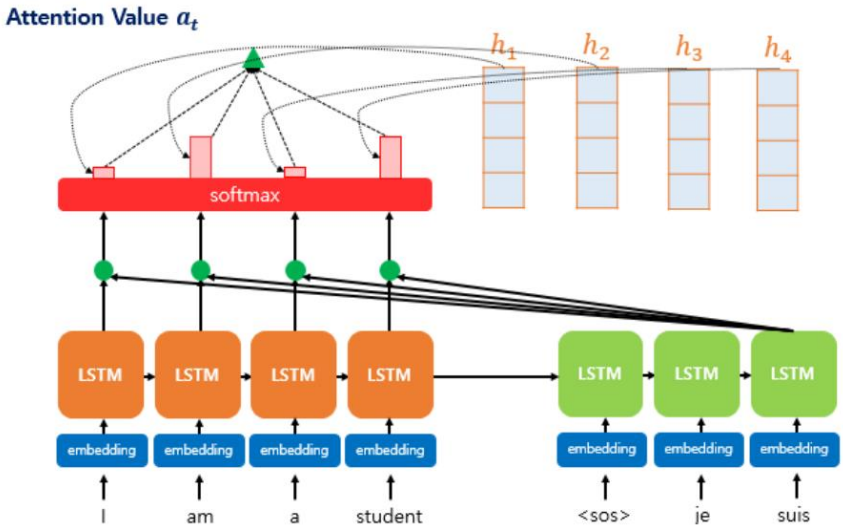
② Attention Distribution 구하기



위 수식을 통한 Decoder 의 t시점에 대한 Attention Score 의 벡터를 Softmax 에 통과시키게 되면, 우리는 0 과 1 사이의 값으로 정규화된 일종의 확률들을 얻을 수 있습니다. 즉, 이 값은 Encoder 의 각 시점 Hidden State 들이 Decoder 의 t시점 Hidden State 에 대해 가지는 중요도로 받아들일 수 있습니다. 이를 수식으로 나타내면 다음과 같습니다.

$$a^t = softmax(e^t)$$

③ Attention Value 구하기



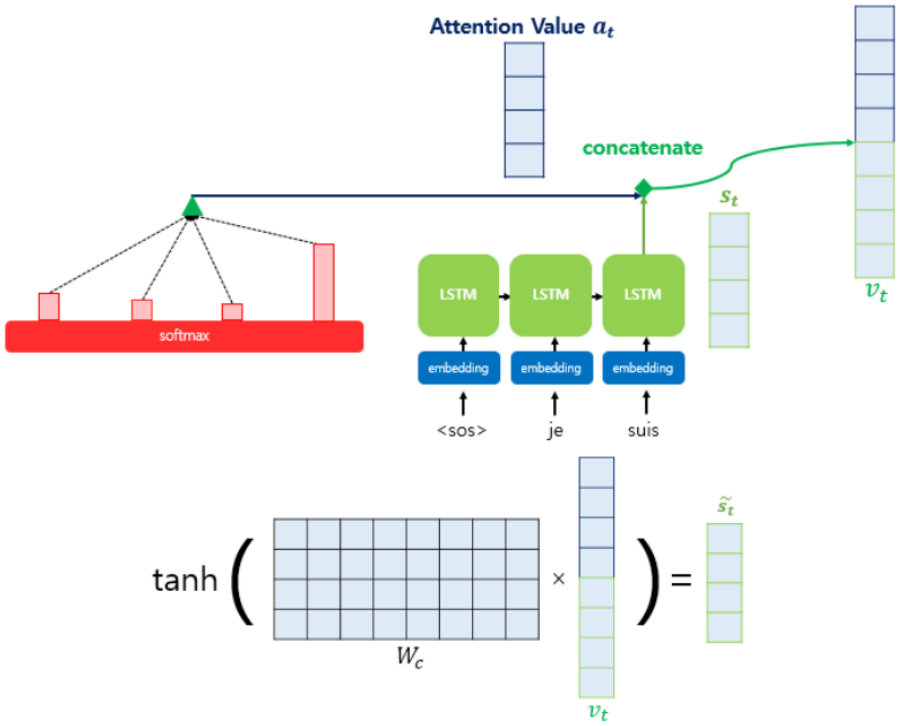
Encoder 의 모든 Hidden State 들이 가지는 중요도를 Attention Distribution 형태로 구했으므로, Attention Distribution 을 Encoder 의 모든 Hidden State 와 곱해주면 가중합과 같은 효과를 얻을 수 있게 됩니다. Attention value 를 a^t라고 한다면, 이를 바탕으로 다음과같이 표현할 수 있습니다.

$$a^t = \sum_{i=1}^{n} a_i^t h_i = a^t \cdot h$$

$$where\ h = [h_1, \cdots, h_n]$$

이런 Attention Value 가 Attention 모델에서의 Context Vector 가 됩니다. Seq2Seq 에서는 h_n만을 Context Vector 로 사용했지만, Attention 모델에서는 Attention Value a_t를 사용함으로써 Encoder 의 모든 hidden state $[h_1, \cdots, h_n]$을 모두 활용하여 기존 Seq2Seq 의 병목현상을 해결할 수 있었습니다.

④ Decoder 의 Hidden State 와 연결하기 (Concatenation)



위 과정을 통해 구한 Attention Value 와 Decoder 의 현 시점 Hidden State 인 s_t를 연결한 벡터 v_t를 구성합니다. 그리고 이 벡터를 가중치와 연산해줌으로써 출력층의 입력을 만들어주게 됩니다. 기존 Encoder-Decoder 구조에서는 h_n이 바로 출력층의 입력이 됐다면, 여기서는 Hidden State 와 Attention Value 를 연결한 후 tanh 함수를 통과한 벡터가 출력층의 입력이 되는 것입니다.

$$\tilde{s}_t = \tanh\left(W_c \cdot v_t + b_c\right)$$

$$o_t = softmax(W_o \cdot \tilde{s}_t + b_o)$$

4. 마무리

3 주차에서는 자연어에 대한 특성을 알아보고 그 특성을 고려하여 전처리를 하는 방법에 대해서 알아보았습니다. 또한 RNN 의 개념과 그 구조에 대해서 알아보았습니다. RNN 모델은 Sequential 한 데이터의 특성을 잘 살릴 수 있다는 장점이 있지만, 층이 깊어지면 깊어질수록 앞의 정보를 제대로 기억하지 못하는 장기의존성 문제가 존재했습니다. 따라서 이를 보완할 수 있는 LSTM 과 GRU 에 대해서 알아본 이후, RNN 을 응용한 모델인 Seq2Seq 모델과 Attention 에 대해서 알아보았습니다. 이번주차 코딩 실습에서는 Seq2Seq 모델을 활용한 기계번역을 할 에정입니다.

지금까지 3 주동안 딥러닝의 전체적인 구조를 공부해보았습니다. 1 주차에서는 딥러닝의 정의와 원리, 2 주차에서는 이미지 데이터의 특징과 CNN 에 대해서 알아보는 시간을 보냈고, 3 주차에서는 자연어의 특징과 RNN 에 대해서 알아보는 시간을 보냈습니다. 추가적으로 매주 코딩 실습에서는 그 주차에서 배운 모델을 직접 실행해볼 수 있었습니다. 이렇게 3 주동안의 스터디를 마치게 되었습니다! ~~이제 진짜 끝...이 아닌 이제 진짜 시작...~~

분명 딥러닝과 통계학에는 교집합이 존재합니다. 하지만, 그 성격이 많이 다르기 때문에 통계학도로써 딥러닝을 경험해보는 것은 쉽지 않습니다. 실제로 딥러닝 모델을 구현하다보면 내가 통계학과인지, 아니면 소프트웨어학과인지 헷갈리게 됩니다. 그래도 이번 기회를 통해 딥러닝을 간단하게나마 소개해드릴 수 있었어서 참 영광입니다. 여러분들도 이번 기회를 통해 딥러닝에 관심을 가져보시고, 같이 공부해 나갈 수 있었으면 좋겠습니다. 그동안 딥러닝 클린업을 들어주신 여러분들께 감사드립니다.