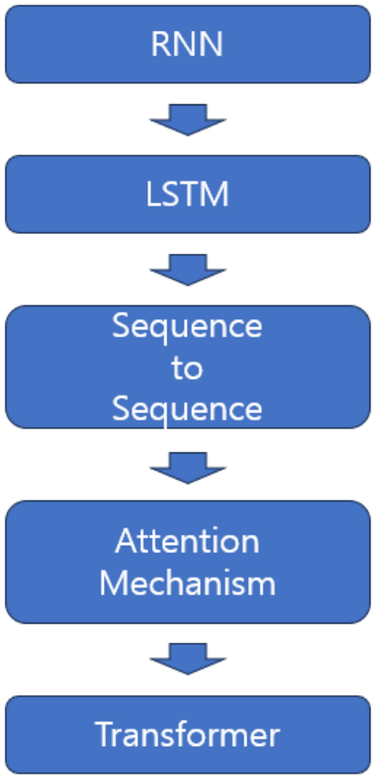
01

Sequence - To- Seqeunce

이번 장에서 Sequence-To-Sequence를 공부하기 전에 위에서 언급했던 RNN과 LSTM에 대해서 간략하게 알아보고 가도록 하겠습니다.

[그림 3-1] 3장의 학습 순서



RNN

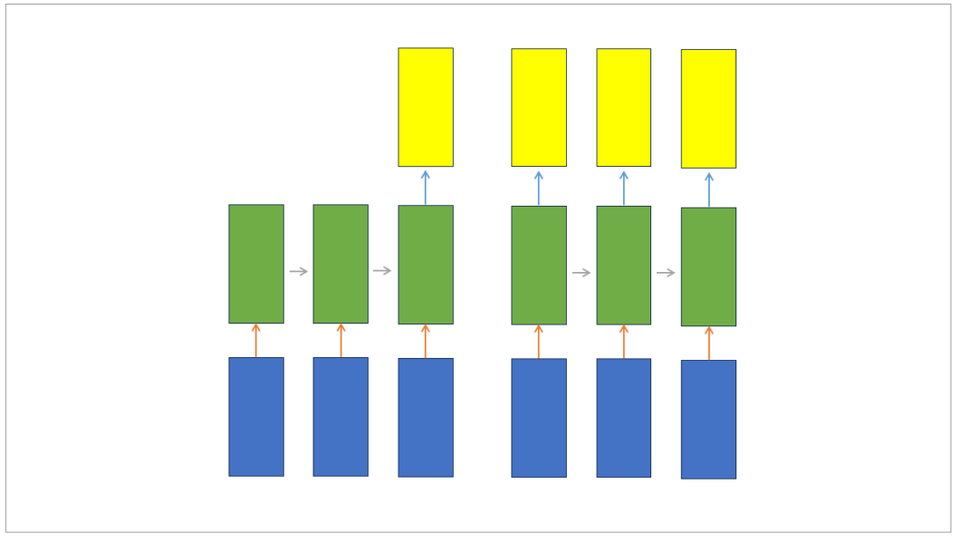
RNN(Recurrent Neural Network)은 순한 신경망이라는 이름으로 사용되는 딥러닝 알고리즘입니다. RNN의 개념은 1986년에 제시되었으며, 그 뒤에 RNN이 가지고 있는 문제점을 보완할 수 있는 LSTM 모델이 1997년에 발표되었습니다.

RNN은 시계열 데이터에 적합한 모델로 알려져 있으며 우리가 사용하는 말과 문장은 대표적인 시계열 데이터입니다. 우리가 하는 말과 문장들은 앞뒤 항목에 의존하기 때문에 시간의 흐름에 따라 말과 문장을 인식을 하고 대화 및 이해가 가능합니다. 말의 순서가 바뀌어서 기록된 문장은 우리가 제대로 이해하기 힘들 것입니다. 이런 데이터를 순차 데이터라고 부릅니다.

RNN을 사용한 대표적인 텍스트 분석은 언어 모델, 감성분석, 기계 번역 등이 있습니다.

RNN은 모델은 다대다(Many to Many) 방식으로도 사용할 수 있으며, 다대다 구조는 입력과 출력의 관계에서 여러 입력이 여러 출력으로 매핑 되는 구조이며 기계 번역, 음성 인식에서 활용됩니다. 참고로 다대일(Many to One) 구조는 감성분석등에 활용됩니다.

[그림 3-2] 다대일, 다대다 모델 구조



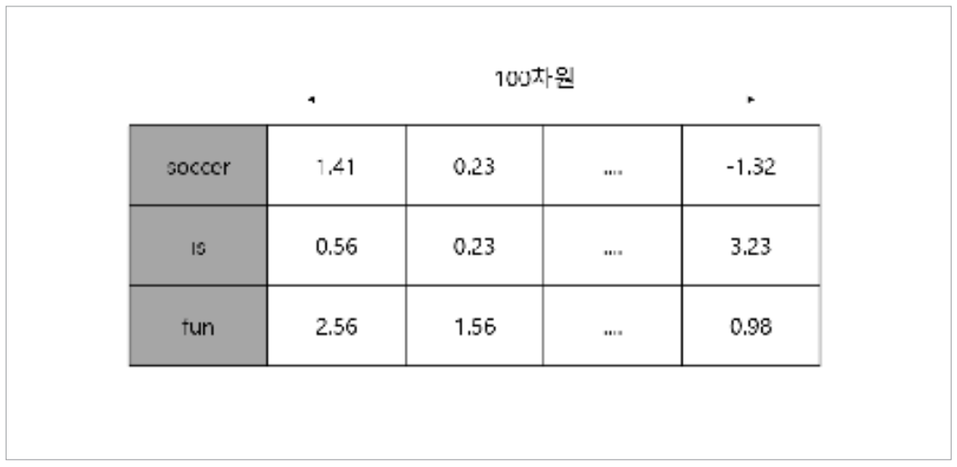
이번 장에서는 RNN 알고리즘 내부의 구체적인 내용보다 NLP 모델에서 어떻게 구현이 되는지에 대한 과정을 설명하겠습니다.

아래와 같은 문장이 있습니다.

"Soccer is fun"

RNN에 입력 값에 텍스트 데이터를 넣는다면 이전 챕터에서 배운 임베딩 된 벡터 데이터로 넣어야 합니다. 각 단어가 100차원의 벡터로 표현되었다고 가정하면 다음과 같은 표로 표현할 수 있습니다.

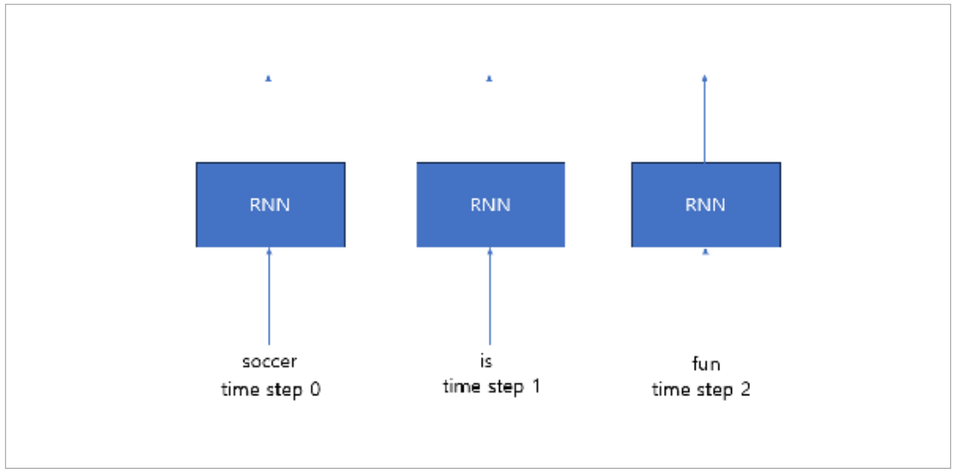
[그림 3-3] 워드 임베딩



각 단어들을 표현하고 있는 100차원의 벡터들은 초기에는 랜덤하게 설정되어 있으며, 학습을 통해서 업데이트가 되는 가중치 값입니다. 위의 표에서 soccer라는 값이 (1.41, 0.23.... -1.32)로 초기 표현되었지만 학습이 진행되면 해당 값들도 최적의 값으로 변경될 것입니다.

RNN 모델에 해당 단어가 입력되는 과정은 soccer의 100차원 벡터가 먼저 입력되고 is의 100차원 벡터, fun의 100차원 벡터가 순서를 가지고 입력이 됩니다. 그림으로 표현하면 아래와 같습니다.

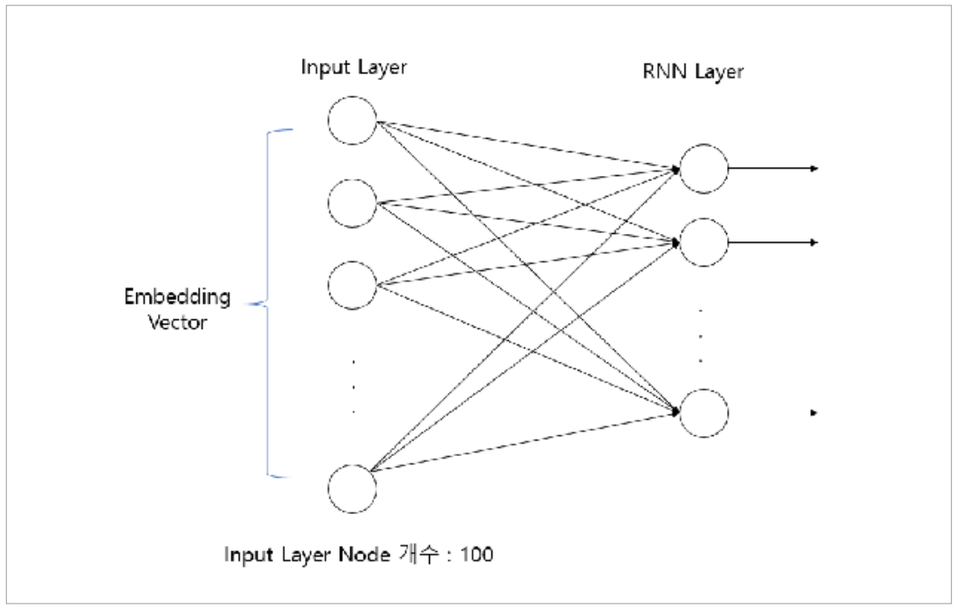
[그림 3-4] RNN 입력 구조



동일한 RNN 층이 여러 번 순차적으로 사용되고 있습니다. 입력되는 값의 순서를 표현하기 위해서 시계열 데이터에서는 time step이라는 표현을 사용합니다.

time step 0에서는 soccer의 임베딩 벡터(100차원) 값이 들어가고 time step 1에는 is의 임베딩 벡터가 time step 2에는 fun의 임베딩 벡터가 들어갑니다. RNN은 딥러닝 구조의 은닉층 역할을 하고 있습니다. 그렇기때문에 은닉층으로 전달할 층이 있어야 하며, 여기서는 입력층이 그 역할을 담당합니다.

[그림 3-5] Input, RNN Layer

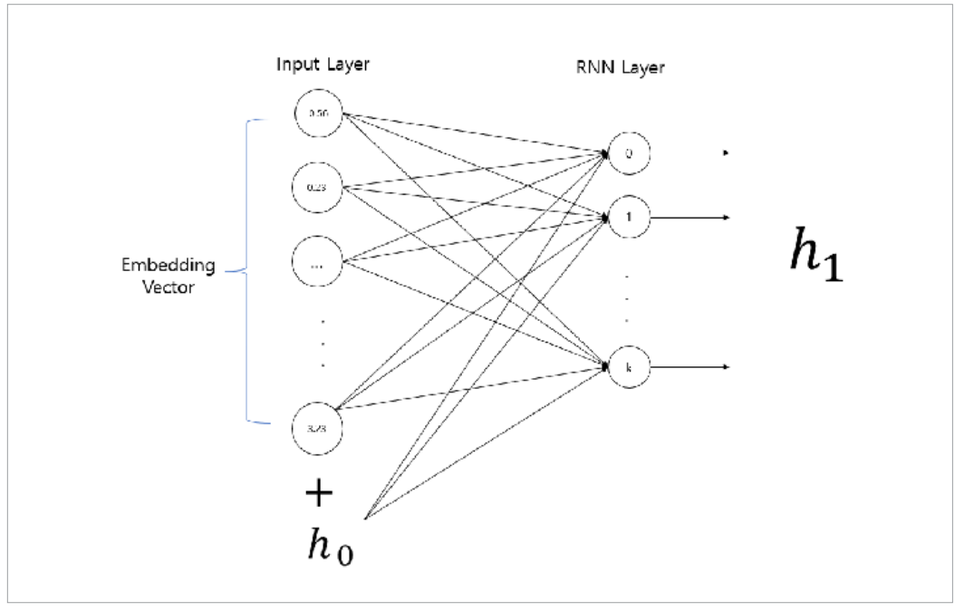


Input Layer의 값은 임베딩 벡터의 차원 수와 동일합니다. RNN Layer는 은닉층이므로 하이퍼파라미터로 사용자가 개수를 지정합니다. 여기서는 편의상 k를 지정했다고 하겠습니다. 그리고 은닉층에는 반드시 Bias의 값도 포함되어 있다는 것을 기억해 주세요. 설명을 위해서 Bias의 표현은 생략하였습니다.

은닉층을 통과하면 반드시 통과해야 하는 함수가 있습니다. 바로 활성화 함수입니다. 언어 모델에서 사용하는 RNN은 활성화 함수로 tanh를 많이 사용합니다.

time step의 순서대로 RNN 구동 원리를 살펴보겠습니다. 먼저 time step 0에서는 soccer의 임베딩 벡터의 값이 Input Layer에 입력됩니다. 그리고 RNN 층을 거쳐 tanh 활성화 함수를 통과하여 출력됩니다. 이 값을 h0라고 하겠습니다.

[그림 3-6] RNN의Time step 1



time step 1에서 is의 임베딩 벡터의 값이 Input Layer에 입력되면서 이전 time step 0의 출력값인 h0도 함께 은닉층인 RNN Layer에 전달됩니다. 즉 현재 time step에서의 은닉층의 입력값으로 이전 time step의 결과값이 사용됩니다.

[그림 3-7] RNN time step

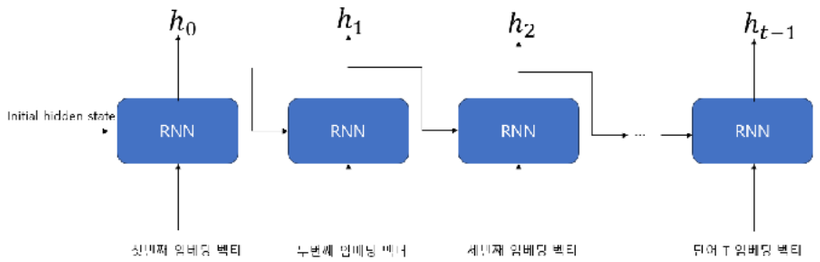


그림 3-7의 h_t-1이 RNN 층이 출력하는 입력된 텍스트 데이터에 대한 최종값이 됩니다. 이 값이 RNN 다음 계층으로 전달됩니다. h_t-1에는 이전에 입력된 단어들의 정보(hidden state)도 다음 계층으로 전달하게 됩니다.

위의 그림을 행렬과 벡터로 표현해보겠습니다.

h_t : 특정 time step에서 RNN층을 통해 출력되는 값

- $h_t=\tanh^{f0}(x_t \cdot W_x+h_{t-1} \cdot W_h+bias)$
  $x_t \cdot W_x+h_{t-1} \cdot W_h+bias$의 차원수와 h_t의 차원수는 동일

- x_t : 해당 time step 에서 입력되는 단어의 임베딩 벡터

- h_{t-1} : 이전 time step 에서 전달되는 hidden state

- W_h : time step 과는 상관없이 가중치 동일
  (parameter sharing)

- W_x : time step 과는 상관없이 가중치 동일
  (parameter sharing)

[그림 3-8] RNN 의 행렬식

$$\mathbf{W_x} = \begin{bmatrix} W_{0,0}^{(x)} & W_{0,1}^{(x)} & \cdots & W_{0,j-1}^{(x)} \\ W_{1,0}^{(x)} & W_{1,1}^{(x)} & \cdots & W_{1,j-1}^{(x)} \\ W_{2,0}^{(x)} & W_{2,1}^{(x)} & \cdots & W_{2,j-1}^{(x)} \\ \vdots & \vdots & \ddots & \vdots \\ W_{k-1,0}^{(x)} & W_{k-1,1}^{(x)} & \cdots & W_{k-1,j-1}^{(x)} \end{bmatrix} \quad \mathbf{W_h} = \begin{bmatrix} W_{0,0}^{(h)} & W_{0,1}^{(h)} & \cdots & W_{0,j-1}^{(h)} \\ W_{1,0}^{(h)} & W_{1,1}^{(h)} & \cdots & W_{1,j-1}^{(h)} \\ W_{2,0}^{(h)} & W_{2,1}^{(h)} & \cdots & W_{2,j-1}^{(h)} \\ \vdots & \vdots & \ddots & \vdots \\ W_{j-1,0}^{(h)} & W_{j-1,1}^{(h)} & \cdots & W_{j-1,j-1}^{(h)} \end{bmatrix}$$

$$\mathbf{x_t} = (x_{t,0}, x_{t,1}, \ldots, x_{t,k-1})$$

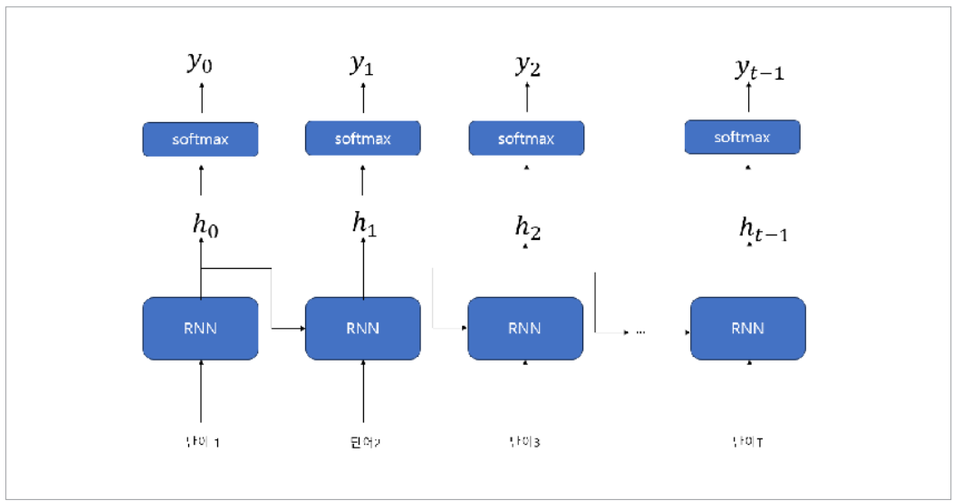
$$\mathbf{h_{t-1}} = (h_{t-1,0}, h_{t-1,1}, \ldots, h_{t-1,j-1})$$

$$\mathbf{b} = (b_0, b_1, \ldots, b_{j-1})$$

RNN은 아래와 같이 언어 모델에 사용됩니다. 언어 모델에 RNN이 적용되는 경우에는 각 단어에 대한 RNN의 출력값은 다음 단어를 예측하는 목적으로 사용됩니다.

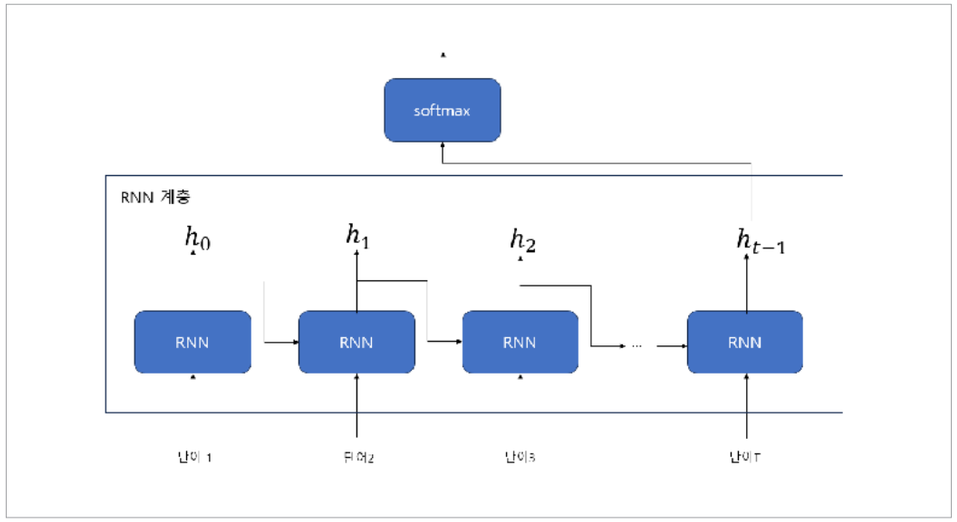
그림 3-9에서 Softmax 함수를 통과해서 출력되는 이유는 다음 단어을 맞추는 문제는 다중 분류 문제로 볼 수 있습니다. 그렇기 때문에 softmax에서 출력하는 값은 모델에 사용한 어휘의 수가 되겠습니다. 그림 3-9와 같은 구조는 기계 번역에서 사용되는 구조입니다.

[그림 3-9] RNN 기계 번역 모델



감성분석(Sentiment analysis)에 RNN이 활용됩니다. 아래 그림 3-10의 구조를 보면 출력은 마지막 h_{t-1}만 다음 출력층으로 전달되며 마지막 층에서는 softmax 함수를 통과하여 부정과 긍정을 구분하는 분류 모델이 됩니다.

[그림 3-10] RNN 감성분석 모델



LSTM

RNN 알고리즘에는 장기 의존 문제(Problem of long term dependency)가 존재합니다. 이 문제는 입력된 문서에서 상대적으로 오래전에 사용된 단어의 정보가 잘 전달되지 않습니다.

언어 모델에서 아래 문장의 마지막 단어를 예측하는 예제로 설명해보겠습니다.

I have lived in Japan for a long time, so I speak Japanese.

위 문장에서 마지막에 위치한 단어를 예측한다고 했을 때 RNN 알고리즘은 이전 단어들의 정보가 마지막까지 잘 전달되지 않기 때문에 마지막 단어인 Japanese 단어를 예측하는데 앞에 있는 Japan이라는 단어의 정보가 중요하지만 이 정보가 잘 전달 되지 않습니다.

분류 문제에서 사용했을 때 마지막 time step에 대한 RNN층에서 출력되는 hidden state만 다음 층으로 전달하는 경우, 입력된 문서에서 앞부분에서 사용 된 단어들의 정보가 잘 반영되지 못한다는 문제도 존재합니다.

장기 의존 문제의 주요 원인으로는 경사소실 문제를 들 수 있습니다. 이 문제를 해결하기 위해서 고안된 알고리즘이 LSTM(Long Short Term Memory)입니다.

LSTM은 RNN과 비교해서 가장 큰 차이는 기억셀(Memory cell)을 가지고 있어, 이를 활용하여 오래전 단어의 내용을 보다 잘 기억할 수 있다는 장점이 있습니다. LSTM도 RNN과 마찬가지로 time step t에서 단어 t에 대한 벡터 정보와 이전 은닉층(t-1)에서 전달하는 hidden state 정보를 담고 있는 벡터 h_{t-1}를 같이 입력 받는 구조로 되어 있습니다. 여기에 추가적으로 기억셀이 추가되어 장기기억이 가능합니다.

[그림 3-11] LSTM 구조

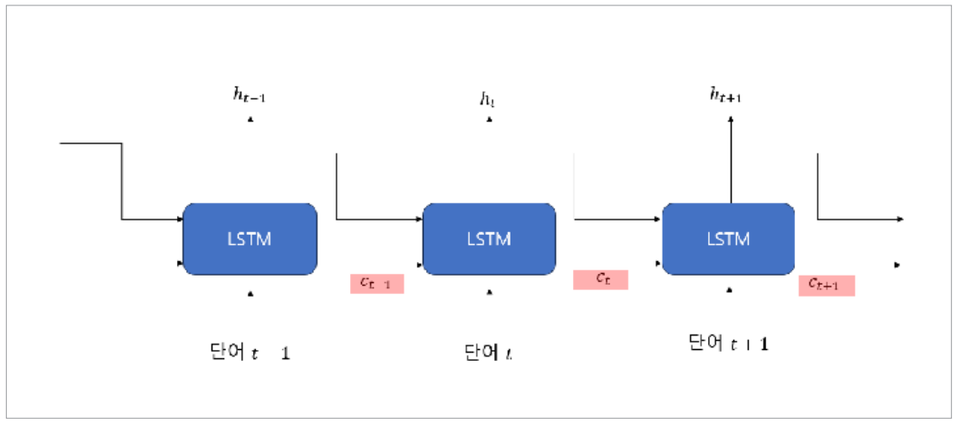
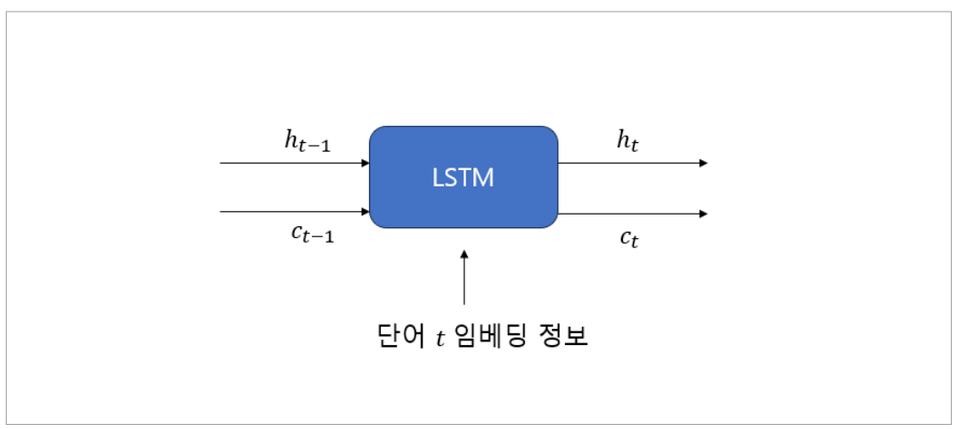


그림 3-11에서 빨간색으로 표시된 부분이 새롭게 추가된 기억셀입니다. RNN은 hidden state 정보만 전달 했지만, LSTM은 기억셀의 값도 다음 time step의 입력 값으로 들어갑니다.

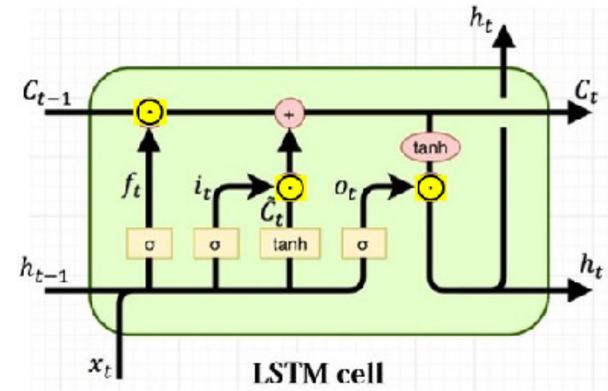
[그림 3-12] time step t에서 LSTM



LSTM의 작동원리에 대해서 알아보겠습니다. LSTM안에서는 2가지 특징이 있습니다. 첫번째로는 c_{t-1}(이전 기억셀)이 h_{t-1}(이전 단계에서 전달된 hidden state)와 단어 t의 임베딩 정보를 사용하여 c_t로 정보를 업데이트 합니다.

두번째로 단어 t의 임베딩 정보, h_{t-1}, c_t의 정보를 이용하여 h_t를 계산합니다.

[그림 3-13] LSTM 셀 다이어그램



위의 두가지 작업에서 중요한 역할을 하는 것이 게이트(Gate)입니다. 그림 3-13에서 볼 수 있듯이 Gate는 Input, Forget, Output 총 3개의 게이트가 존재합니다.

게이트는 기억셀이 가지고 있는 이전 정보를 기억/삭제하거나, 새로운 정보를 기억셀을 업데이트하면서 추가합니다. 다시 정리하면, 이전 정보 중에서 정답을 예측하는데 필요한 중요한 정보는 기억하고, 필요 없는 정보는 삭제하며, 새롭게 입력되는 정보들 중에서 중요한 정보는 추가하는 역할을 합니다.

게이트의 역할은 c_{t-1}를 업데이트하여 c_t를 계산하고, c_t와 단어 t의 정보, 그리고 h_{t-1}를 이용하여 h_t를 계산합니다.

Forget Gate

[그림 3-14] Forget Gate 다이어그램

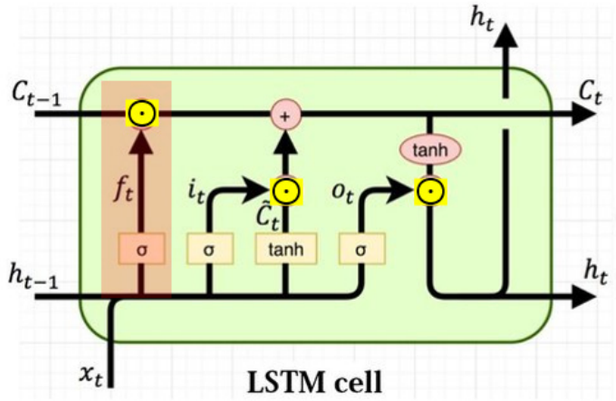


그림 3-14의 붉은 색 부분이 Forget Gate입니다. Forget Gate는 이전 기억 셀(c_{t-1})이 가지고 있는 정보 중에서 정답을 예측하는데 불필요한 정보는 잊어버리는(삭제)하는 역할을 합니다. c_{t-1}이 가지고 있는 각 원소의 정보 중에서 어느 정도의 정보를 잊어버릴 것인지 결정하기 위해서 0~1 사이의 값을 반환하는 sigmoid 함수를 사용합니다.

c_{t-1} 내용 중 얼마만큼의 정보를 잊을 것인지를 결정하기 위해 h_{t-1}와 단어 t의 임베딩 벡터 x_t를 이용하여 계산합니다. 계산 공식은 아래와 같습니다.

$$\sigma\left(x_t \bullet W_x^f + h_{t-1} \bullet W_h^f + b^f\right) \odot c_{t-1}$$

⊙ 기호는 아마다르 곱을 의미합니다. 아마다르 곱은 두 행렬을 곱을 같은 위치에 있는 원소들끼리 곱하는 연산을 의미합니다.
σ 기호는 시그모이드 함수를 의미합니다.
위 수식에서 시그모이드의 값이 0에 가까울수록 더 많이 잊는다는 것을 의미합니다. 시그모이드에 출력된 결과에 따라서 c_{t-1}가 가지고 있는 의미가 원소별로 다를 것입니다.

$$c'_{t-1} = \sigma\left(x_t \bullet W_x^f + h_{t-1} \bullet W_h^f + b^f\right) \odot c_{t-1}$$

[그림 3-14] Forget Gate 결과 예시

10		0.2		2
10		0.1		1
10		0.9		9
10		0.8		8
10	⊙	0.4	=	4
10		0.9		9
10		0.7		7
10		0.3		3
10		0.9		9
c_{t-1}		$\sigma\left(x_t \cdot W_x^f + h_{t-1} \cdot W_h^f + b^f\right)$		c'_{t-1}

그림 3-14는 설명을 위해서 만든 예시입니다. 이전 기억셀의 정보 벡터가 시그모이드가 출력한 벡터값과 아마다르 연산을 통해서 값들이 변화된 것을 확인할 수 있습니다.
c'_{t-1}은 Forget Gate를 통해 업데이트된 c_{t-1}를 의미합니다.

Input Gate

[그림 3-15] Input Gate 다이어그램

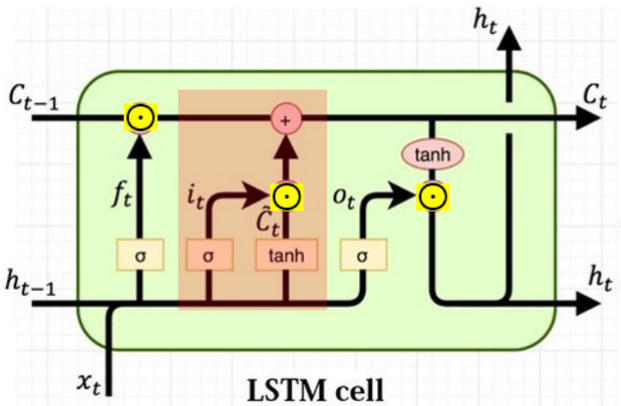


그림 3-15의 붉은 색 박스 부분이 Input Gate를 나타냅니다. Input Gate는 일부의 정보가 삭제된 c'_{t-1}에 새로운 정보를 추가하는 역할을 합니다. 추가하고자 하는 정보를 h_{t-1}와 단어 t의 정보를 사용합니다. 새롭게 추가되는 정보들의 긍부정의 역할을 하기 위해 -1 ~ 1 사이의 값을 출력하는 tanh() 함수를 사용합니다.

$$\tanh(x_t \cdot W_x^n + h_{t-1} \cdot W_h^n + b^n)$$

여기서 나온 결과를 그대로 반영되는 것이 아니라 정답을 예측하기 위한 기여도를 다르게 하기 위해 시그모이드 함수를 통과시킵니다.

$$\sigma(x_t \cdot W_x^i + h_{t-1} \cdot W_h^i + b^i)$$

위의 두 수식을 아마다르 곱을 하고 c'_{t-1}과 합산하여 c_t 를 출력합니다.

$$c_t = c'_{t-1} + \tanh(x_t \cdot W_x^n + h_{t-1} \cdot W_h^n + b^n) \odot \sigma(x_t \cdot W_x^i + h_{t-1} \cdot W_h^i + b^i)$$

Output Gate

[그림 3-16] Output Gate 다이어그램

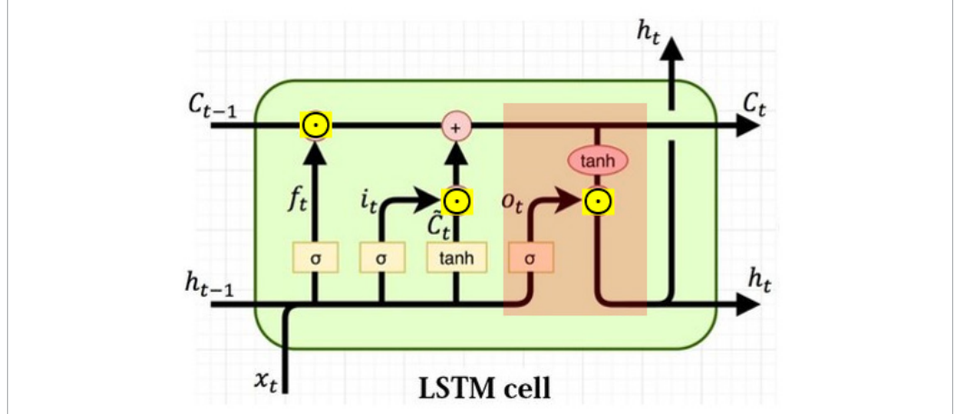


그림 3-16의 붉은 색 부분이 Output Gate입니다. Output Gate의 역할은 forget Gate와 Input Gate를 이용하여 업데이트된 기억셀의 정보(c_t)를 이용하여 현재 LSTM층에서 출력되는 Output 값인 h_t를 계산하는 것입니다.

Output Gate는 c_t가 가지고 있는 원소들 중에서 정답을 예측하는데 중요한 역할을 하는 원소의 비중은 크게 하고, 그렇지 않은 원소들의 비중은 작게 해서 h_t를 계산합니다.

정답을 예측하는데 기여하는 원소의 비중을 계산하기 위해서 현재 LSTM층에 입력되는 h_{t-1}과 단어t의 정보(x_t) 인자로 갖는 sigmoid 함수를 사용합니다.

$$\sigma(x_t \cdot W_x^o + h_{t-1} \cdot W_h^o + b^o)$$

c_t의 원소들의 긍부정 역할을 구분하기 위해서 tanh 함수를 적용합니다.

$$h_t = \sigma(x_t \cdot W_x^o + h_{t-1} \cdot W_h^o + b^o) \odot \tanh(c_t)$$

전체 작동 순서

1. 이전 기억셀(c_{t-1})의 일부 정보를 삭제한다. (Forget Gate)

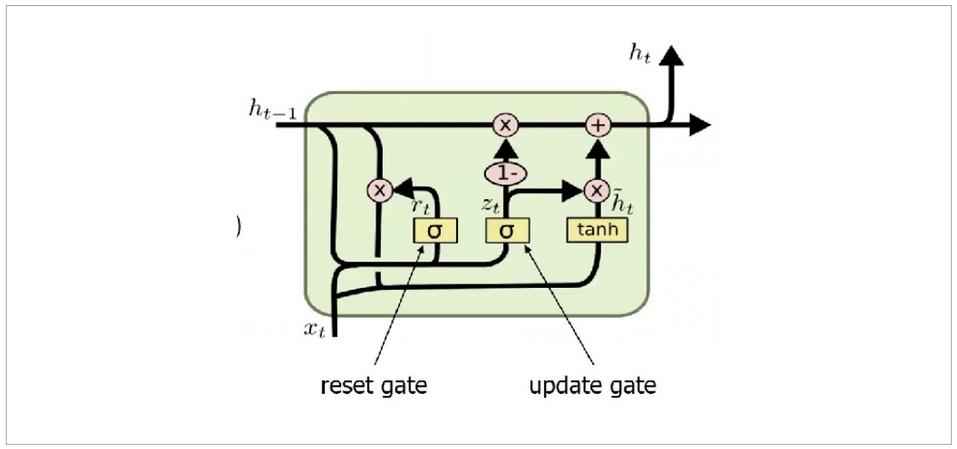
2. 기억셀에 새로운 정보를 추가하여 c_t를 생성한다.(Input Gate)

3. 업데이트된 기억셀 정보를 사용해서 h_t를 출력한다. (Output Gate)

LSTM에서 다음 계층으로 출력되는 값은 hidden state(h_t)입니다. 일반적으로 기억 셀의 특징은 다음 계층(출력층이나 또 다른 은닉층)으로 그 값이 전달되지 않습니다. 곧 기억셀은 LSTM 계층 내에서만 이전 단어들의 정보를 기억하는 목적으로 사용됩니다.

GRU(Gated Recurrent Unit)은 LSTM과 마찬가지로 Gate 개념을 사용하지만 기억셀은 사용하지 않습니다. Reset Gate와 Update Gate를 사용하여 hidden state 정보를 업데이트합니다. GRU는 LSTM 보다 간단한 구조이기 때문에 계산 속도는 빠르지만 정확도가 떨어지는 단점이 있습니다.

[그림 3-17] GRU 구조

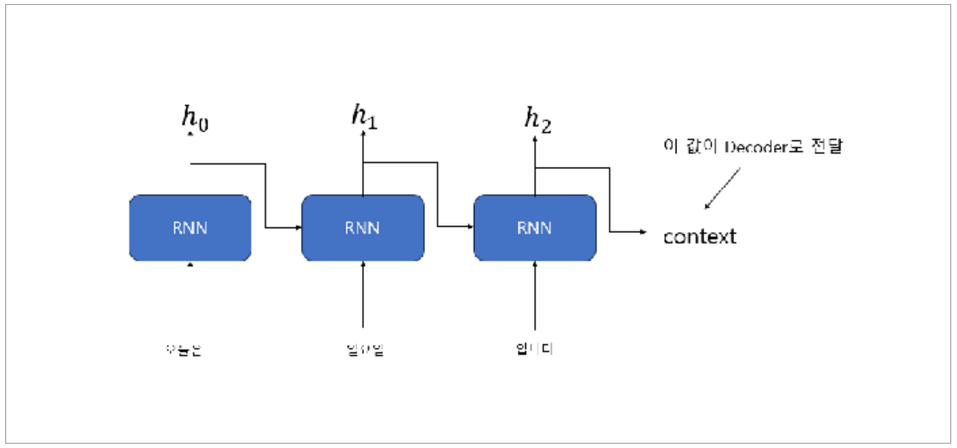


seq2seq

seq2seq(sequence to sequece) 기본 개념 및 단점을 이해를 해야 트랜스포머를 이해하는 도움이 됩니다. seq2seq에 대해서 알아보겠습니다. Seq2Seq의 구조는 Encoder와 Decoder로 구성되어 있습니다.

Seq2Seq의 주요 목적은 특정 도메인의 정보를 다른 도메인으로 변환시킬 수 있다는 것입니다. 대표적인 예로 번역이 있습니다. 여기서도 번역을 예로 설명하겠습니다. “오늘은 일요일 입니다”를 “Today is Sunday”라고 출력하는 예제로 설명하겠습니다.

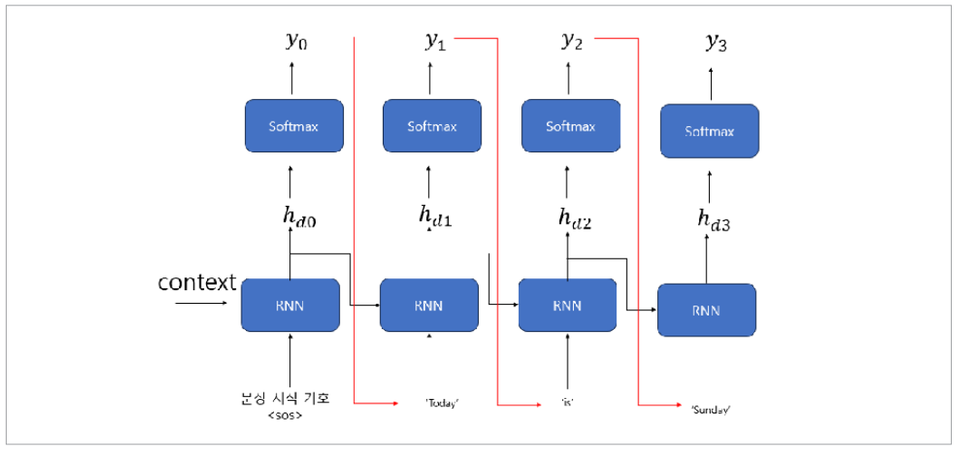
[그림 3-18] Seq2Seq Encoder 구조



Encoder의 역할은 입력된 텍스트 데이터를 숫자 형태로 혹은 벡터 형태로 변환합니다. Decoder의 역할은 Encoder에 의해 숫자로 변경된 정보를 다른 형태의 텍스트 데이터로 변환합니다. Decoder는 언어 모형의 역할을 수행합니다.

Encoder와 Decoder는 그림 3-18과처럼 순환신경망 기반 모형이 사용됩니다. (예 RNN, LSTM)

[그림 3-19] Seq2Seq Decoder 구조



Encoder는 입력 문장의 모든 단어를 순차적으로 입력 받은 뒤 마지막에 이 모든 단어 정보들을 압축해서 하나의 벡터를 생성(그림 3-18) 이를 컨텍스트 벡터(context vector)라고 합니다. 입력된 문장의 정보가 컨텍스트 벡터로 만들어지면 Decoder로 전달합니다.

Decoder에서 초기 입력으로 문장의 시작을 알리는 <sos> 토큰과 Encoder에서 전달된 컨텍스트 벡터가 입력됩니다. <sos> 입력 다음으로는 다음에 softmax 함수를 사용하여 등장할 확률이 높은 단어를 예측합니다. Decoder에서 예측을 하는데 사용되는 전체 단어의 수가 N이라고 했을 때 소프트맥스의 출력층은 동일하게 N개의 노드를 갖습니다.

N개의 단어의 인덱스 번호가 존재하며, 소프트맥스 함수가 N개의 노드 중에서 예측하는 단어의 인덱스 번호에 해당되는 노드의 확률 값을 가장 높게 할 것입니다.

Decoder의 첫번째 출력 값에서 Today가 출력되고, 해당 은닉층의 값이 두번째 입력에 사용됩니다. 두번째 출력은 Today 다음에 등장할 확률이 가장 높은 단어를 출력할 것입니다. 이런 방식으로 문장의 끝을 의미하는 토큰인 <eos>를 예측될 때까지 반복됩니다.

02 Attention mechanism

트랜스포머는 Google에서 2017년 제안한 attention 기반의 encode-decoder 알고리즘입니다. 순환 신경망(RNN) 기반의 방법이 아니라 attention을 사용하였습니다. 트랜스포머를 기반으로 하는 주요 Application에는 뒤에서 학습하게 될 BERT(Bidirectional Encoder Representations from Transformers), GPT(Generative Pre-trained Transformer), BART(Bidirectional and Auto-Regressive Transformers)등이 있습니다.

순환신경망 기반의 seq2seq 모형이 가지고 있는 큰 문제점은 마지막의 컨텍스트 벡터만 Decoder에 전달하기 때문에 입력된 모든 단어들의 정보가 제대로 전달되지 못한다는 문제가 발생했습니다.

특히, 입력된 단어가 많은 경우 앞쪽에서 입력된 단어들의 정보는 거의 전달이 안되는 문제가 발생합니다. 번역 등의 작업을 할 때 마지막 단어만 중요한 게 아니라 이전에 입력된 단어들의 정보도 중요한데 Seq2Seq는 문장이 길면 앞의 단어들이 잘 전달되지 않습니다.

순환신경망 기반 seq2seq 모델에서 아래와 같은 문제점을 정리할 수 있습니다.

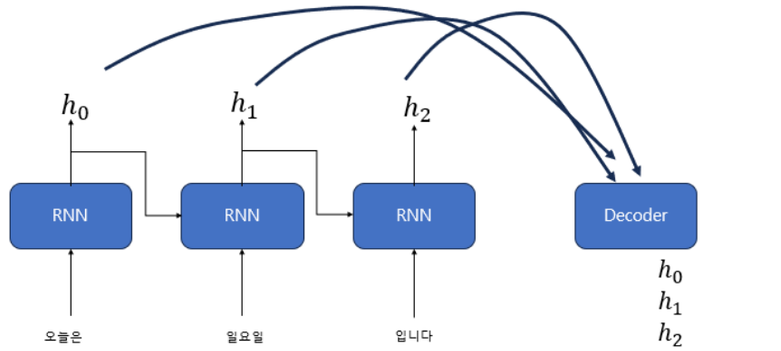
1. Encoder에서 Decoder로 정보를 전달할 때 손실이 발생

Encoder-Decoder 어텐션

이 문제를 해결하기 위해서 Encoder 부분에서 생성된 각 단어에 대한 hidden state 정보를 모두 Decoder로 전달하면 됩니다. Encoder부분에서 생성되는 각 단어에 대한 hidden state 정보를 모두 Decoder로 전달합니다. 그림 3-20은 “오늘은 일요일 입니다”라는 문장을 “Today is Sunday”로 번역하는 예시입니다.

Encoder의 각각의 time step에서 출력되는 hidden state의 값을 행렬로 Decoder에 전달합니다. hidden state의 값이 5차원이라면 Decoder에 전달되는 행렬의 형상은 (3, 5)으로 전달됩니다.

[그림 3-20] Encoder의 모든 hidden state의 값을 Decoder에 전달

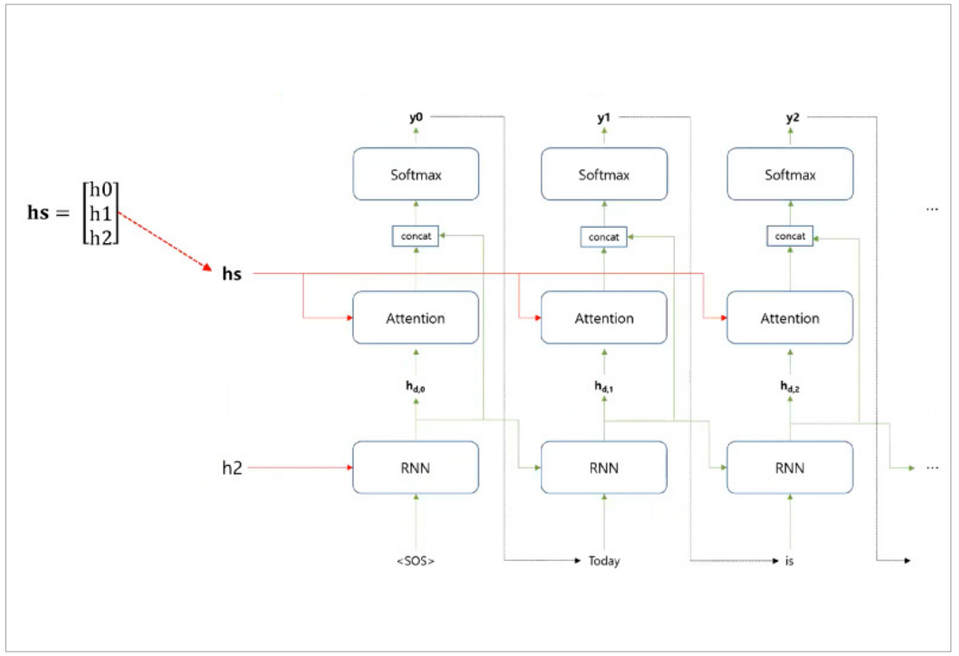


Decoder부분에서는 Encoder에서 전달받은 행렬 값을 어떻게 활용할까요? Decoder가 예측하는 단어가 Sunday라고 했을 때 전달 받은 행렬의 값을 모두 활용합니다. 하지만 행렬 값 중에서 일요일이라는 단어가 있는 h_1와 연관성이 높을 것입니다. 그렇기 때문에 h_1에 상대적으로 높은 가중치를 부여합니다. 가중치가 높다는 것은 다른 정보보다 더 많은 정보를 활용한다는 의미이고, 바로 이런 매커니즘이 어텐션 매커니즘의 기본 개념입니다.

어텐션은 Decoder에서만 사용됩니다. Encoder에서 전달받은 모든 hidden state값을 사용하지만 예측하려고 하는 단어와 연관성이 높은 hidden state의 값을 더 많은 가중치를 부여해서 사용하는 방식입니다.

Decoder에서는 정확한 번역을 위해서 Encoder에서 전달된 h_s의 각각의 hidden state에 예측하고자 하는 단어와 더 많은 관련이 있는 Encoder 단어에 더 많은 가중치를 부여합니다.

[그림 3-21] Attention을 포함한 Decoder 구조



Attention 층에서는 어떻게 가중치를 부여하는지 예시로 알아보겠습니다. 그림 3-21에서 y0에서 예측하는 단어는 Today가 될 것입니다. Encoder에서 전달받은 h_s의 값은 아래와 같습니다.

$$\begin{matrix} \text{오늘은} \rightarrow \\ \text{일요일} \rightarrow \\ \text{입니다} \rightarrow \end{matrix} \begin{bmatrix} h_0 \\ h_1 \\ h_2 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 1 & 2 \\ 1 & 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 & 1 \end{bmatrix}$$

가중치의 값은 0~1사이의 값을 갖습니다. 위의 예제에서 각각 h_0, h_1, h_2가 0.8, 0.1, 0.1의 가중치가 계산되었습니다. 계산 방법은 아래에 설명을 하겠습니다. 그럼 각각의 hidden state의 원소에 가중치를 곱하면 브로드캐스팅 연산을 통해서 아래와 같이 게산 됩니다.

$$\begin{bmatrix} 0.8 & 0 & 0 & 0.8 & 1.6 \\ 0.1 & 0 & 0 & 0.1 & 0.1 \\ 0.1 & 0.1 & 0 & 0 & 0.1 \end{bmatrix}$$

Attention 층에서 출력되는 값은 위의 결과 벡터들을 모두 더한 값인 (1.0, 0.1, 0, 0.9, 1.8) 이 됩니다. 이 값을 RNN층의 출력 값인 $h_{d,0}$와 concat 연산을 사용하여 Softmax층으로 전달됩니다.

그럼 이제는 가중치는 어떻게 계산되는지 알아보겠습니다. 그림 3-21에서 첫번째 RNN의 hidden state 출력값인 $h_{d,0}$과 Encoder에서 전달받은 h_s의 각각의 hidden state와의 유사도를 계산하게 됩니다. 유사도를 구하는 방식은 여러가지 존재하지만 어텐션은 계산 속도가 가장 빠른 내적을 사용합니다.

가중치는 h_s의 각각의 hidden state와 Decoder에서 예측하고자 하는 단어에 대한 hidden state와 내적 연산을 사용하여 각각의 hidden state와의 유사도를 계산합니다. Decoder 부분에서 첫번째 RNN층에서 출력되는 “Today”단어를 예측하는 데 사용되는 hidden state인 $h_{d,0}$의 값을 (1 0 0 0 2)라고 하겠습니다.

그럼 각각 h_0, h_1, h_2와 $h_{d,0}$와의 내적 연산(dot product)을 수행합니다.

$$\begin{bmatrix} 1 & 0 & 0 & 1 & 2 \\ 1 & 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 2 \end{bmatrix} = \begin{bmatrix} 5 \\ 3 \\ 3 \end{bmatrix}$$

이 값들을 어텐션 스코어(Attension Score)라고 합니다. 모든 어텐션 스코어의 값은 스칼라입니다. 어텐션 스코어의 값이 클수록 관련도가 크다는 것을 의미합니다. 위의 출력값을 그대로 사용하지 않고 위에서 언급했듯이 가중치의 값은 0 ~ 1 사이의 확률 값을 갖는다고 하였습니다.

그래서 위의 값에서 softmax 함수를 적용합니다. 그럼 어텐션 스코어의 값을 가지고 가중치 값으로 변환해보겠습니다.

$$y_k = \frac{\exp(a_k)}{\sum_{i=1}^{n} \exp(a_i)}$$

어텐션 스코어가 5인 경우에 대한 가중치 값은 위의 공식에 따라서 $e^5/(e^5+e^3+e^3)$ 약 0.80이 나옵니다. 같은 방식으로 어텐션 스코어 3에 대한 가중치 값은 약 0.10이 나오게 됩니다.

최종적으로 출력되는 값은 어텐션에서 출력되는 값과 RNN 층에서 출력되는 값과 concat 연산을 합니다. 즉, concat((1.0, 0.1, 0, 0.9, 1.8), (1 0 0 0 2))가 됩니다.

Decoder에서 사용된 어텐션을 Encoder-Decoder 어텐션(encoder-decoder attention)이라고 합니다. 이제는 셀프어텐션(self-attention)에 대해서 알아보겠습니다.

셀프-어텐션

먼저 Encoder에서 사용되는 어텐션에 대해서 설명하겠습니다. 셀프 어텐션과 어텐션과의 차이는 어텐션은 Encoder-Decoder 모형에서 보통 Decoder에서는 Encoder에서 넘어오는 정보에 가중치를 주는 식으로 작동합니다. 셀프 어텐션은 입력된 텍스트 데이터 내에 존재하는 단어들 간의 관계를 파악하기 위해 사용됩니다. 관련이 높은 단어에 더 많은 가중치를 주기 위해서 사용합니다.

지시대명사가 무엇을 의미하는지 등을 파악하는데 유용합니다. 예를 들어 The dog likes an apple. It has a long tail. 이러한 경우, 입력된 데이터 내에 존재하는 다른 단어들 하고 관계를 파악할 수 있다면, it이 dog라는 단어와 관련이 높다는 것을 알 수 있습니다. 그렇다면 it을 예측하는데 있어서 dog에 더 많은 가중치를 주면 더 정확도가 높아질 것입니다.

예를 들어 설명해보겠습니다 입력 문장이 “Today is Monday” 라는 문장이 있습니다. 이 문장의 각 단어에 대한 임베딩 벡터값이 아래와 같다고 가정해보겠습니다.

$$\begin{matrix}\text{Today} \rightarrow \\ \text{is} \rightarrow \\ \text{Monday} \rightarrow\end{matrix}\begin{bmatrix}e_0 \\ e_1 \\ e_2\end{bmatrix} = \begin{bmatrix}1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1\end{bmatrix}$$

Monday에 대한 셀프-어텐션 결과를 얻고자 하는 경우 Monday에 대한 임베딩 벡터와 다른 단어들의 벡터들 간의 내적 연산을 합니다. 이때 Monday 단어의 자기자신도 포함합니다.

$$\begin{bmatrix}1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1\end{bmatrix} \times \begin{bmatrix}1 \\ 1 \\ 0 \\ 0 \\ 1\end{bmatrix} = \begin{bmatrix}2 \\ 1 \\ 3\end{bmatrix}$$

이렇게 계산된 가중치의 값을 소프트맥스함수에 적용합니다.

$e^2/(e^2+e^1+e^3) \approx 0.25$
$e^1/(e^2+e^1+e^3) \approx 0.09$
$e^3/(e^2+e^1+e^3) \approx 0.66$

각 단어의 임베딩 벡터에 가중치를 곱셈 연산과 덧셈 연산을 하여 셀프-어텐션 결과값을 도출합니다.

(1 0 0 1 1) x 0.25 = (0.25 0 0 0.25 0.25)
(1 0 0 1 0) x 0.09 = (0.09 0 0 0.09 0)
(1 1 0 0 1) x 0.66 = (0.66 0.66 0 0 0.66)

트랜스포머에서의 셀프-어텐션은 입력 받은 단어들 중에서 어떠한 단어에 대한 더 많은 가중치를 부여해야 하는지 파악하기 위해서 각 단어들에 대한 Query, Key, Value라는 서로 다른 3개의 벡터들을 사용합니다.

Key, Value 벡터들은 사전 형태의 데이터를 의미합니다. key는 단어의 id와 같은 역할을 하고, value는 해당 단어에 대한 구체적인 정보를 저장하는 역할을 한다고 생각할 수 있고, Query벡터는 유사한 다른 단어를 찾을 때 사용되는 질의 벡터라고 생각할 수 있습니다.

동작 순서를 정리해보겠습니다.

1. 입력된 각 단어들에 대해서 Query, Key, Value 벡터를 계산합니다. 이때 각각의 가중치 행렬이 사용됩니다.

2. Query를 이용해서 각각의 key 값들 하고 유사한 정도를 내적을 사용해서 계산하여 어텐션 스코어를 생성합니다.

3. 어텐션 스코어에 softmax함수를 적용하여 가중치 값으로 변경합니다.

4. 가중치 값을 Value 벡터에 곱합니다.

5. 가중치가 곱해진 Value Vector들의 합을 구하여 최종 결과물을 만듭니다.

Query, Key, Value 벡터를 구해보겠습니다. 별도의 가중치 행렬(weights matrix)을 사용합니다. 입력된 각 단어의 임베딩 벡터와 가중치 행렬의 곱을 통해 Q, K, V 벡터를 구합니다. 임베딩 벡터의 차원이 5차원이고, Q, K, V의 차원이 3인 경우 아래와 같이 각 가중치 행렬의 크기를 구할 수 있습니다.

Query 가중치 행렬

$$\begin{bmatrix} W^Q_{0,0} & W^Q_{0,1} & W^Q_{0,2} \\ W^Q_{1,0} & W^Q_{1,1} & W^Q_{1,2} \\ W^Q_{2,0} & W^Q_{2,1} & W^Q_{2,2} \\ W^Q_{3,0} & W^Q_{3,1} & W^Q_{3,2} \\ W^Q_{4,0} & W^Q_{4,1} & W^Q_{4,2} \end{bmatrix}$$

Key 가중치 행렬

$$\begin{bmatrix} W^K_{0,0} & W^K_{0,1} & W^K_{0,2} \\ W^K_{1,0} & W^K_{1,1} & W^K_{1,2} \\ W^K_{2,0} & W^K_{2,1} & W^K_{2,2} \\ W^K_{3,0} & W^K_{3,1} & W^K_{3,2} \\ W^K_{4,0} & W^K_{4,1} & W^K_{4,2} \end{bmatrix}$$

Value 가중치 행렬

$$\begin{bmatrix} W^V_{0,0} & W^V_{0,1} & W^V_{0,2} \\ W^V_{1,0} & W^V_{1,1} & W^V_{1,2} \\ W^V_{2,0} & W^V_{2,1} & W^V_{2,2} \\ W^V_{3,0} & W^V_{3,1} & W^V_{3,2} \\ W^V_{4,0} & W^V_{4,1} & W^V_{4,2} \end{bmatrix}$$

각각 단어의 임베딩 벡터의 값이 Today = (1 0 0 1 1), is = (1 0 0 1 0), Monday = (1 1 0 0 1)이라고 했을 때 Today에 대해서 어텐션 작업을 수행하는 경우 Query 가중치 행렬이 아래와 같이 되어 있습니다.

$$\text{Query 가중치 행렬} = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 1 \end{bmatrix}$$

위의 Query 가중치 행렬의 각각의 값은 학습을 통해서 계산이 됩니다. 위의 값은 랜덤하게 초기화된 가중치 값입니다. 단어 1인 Today에 대한 Query 벡터는 아래와 같이 계산할 수 있습니다.

$$[1\ 0\ 0\ 1\ 1] \times \begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 1 \end{bmatrix} = [2\ 0\ 2]$$

여기서 나온 값은 Today에 대한 Query 벡터가 됩니다. 어텐션 스코어를 계산하기 위해서는 여기서 계산한 Query벡터의 값을 각각의 단어의 key 벡터의 값들과 내적 연산을 진행합니다.

Key 가중치 행렬

$$\begin{bmatrix} 1 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}$$

위의 같이 key 가중치 행렬이 주어졌을 때 각각의 단어에 대한 key 벡터는 아래처럼 계산할 수 있습니다.

$$[1\ 0\ 0\ 1\ 1]\ \text{x}\ \begin{bmatrix} 1 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix} = [3\ 2\ 1]$$

$$[1\ 0\ 0\ 1\ 0]\ \text{x}\ \begin{bmatrix} 1 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix} = [2\ 2\ 0]$$

$$[1\ 1\ 0\ 0\ 1]\ \text{x}\ \begin{bmatrix} 1 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix} = [2\ 1\ 2]$$

단어 1에 대한 Query 벡터 : [2 0 2]
단어 1에 대한 Key 벡터 : [3 2 1]
단어 2에 대한 key 벡터 : [2 2 0]
단어 3에 대한 key 벡터 : [2 1 2]

이 값을 사용해서 Query와 key 값을 내적 계산합니다.

[2 0 2]와 [3 2 1] 내적의 값은 8
[2 0 2]와 [2 2 0] 내적의 값은 4
[2 0 2]와 [2 1 2] 내적의 값은 8

이렇게 계산된 [8 4 8]의 값을 softmax함수를 적용하면 [0.5 0 0.5]의 값을 구할 수 있습니다.

이제 value 벡터의 값을 구해보겠습니다. 아래와 같이 Value 가중치 행렬의 값이 초기화 되었다고 가정하겠습니다.

Value 가중치 행렬

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix}$$

$$[1\ 0\ 0\ 1\ 1] \times \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix} = [1\ 2\ 2]$$

$$[1\ 0\ 0\ 1\ 0] \times \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix} = [1\ 1\ 1]$$

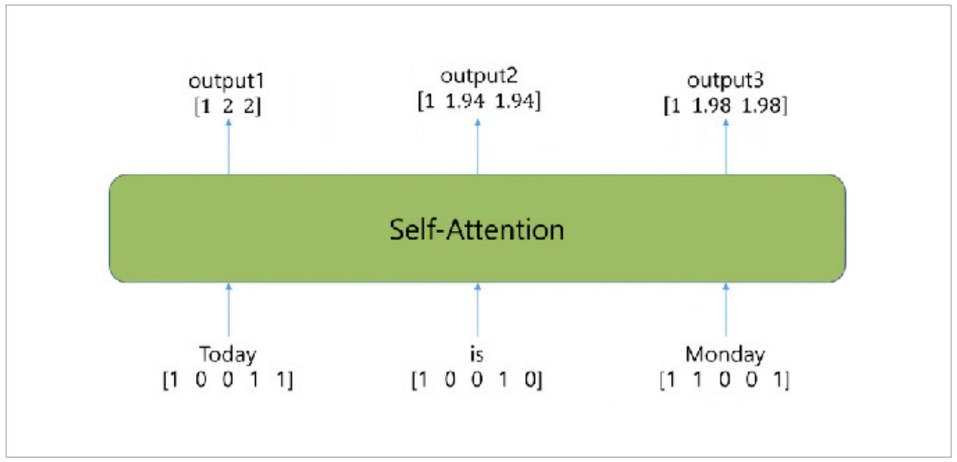
$$[1\ 1\ 0\ 0\ 1] \times \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix} = [1\ 2\ 2]$$

단어1의 value 벡터의 값은 [1 2 2]
단어2의 value 벡터의 값은 [1 1 1]
단어3의 value 벡터의 값은 [1 2 2]

value 벡터의 값과 query벡터의 값을 곱하여 아래와 같은 값을 생성합니다.

0.5 x [1 2 2] = [0.5 1 1]
0.0 x [1 1 1] = [0 0 0]
0.5 x [1 2 2] = [0.5 1 1]

최종 결과물은 각 벡터의 합인 [1 2 2]가 됩니다. 이 값이 Today라는 단어에 대한 셀프-어텐션의 값이 됩니다. 동일한 작업을 단어 2, 3에 대해서 수행합니다. 다음과 같은 결과가 출력됩니다.



위의 방식이 트랜스포머에 사용되는 셀프-어텐션의 방법입니다. 이 방식을 수식으로 표현해보겠습니다.

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

Q는 Query 벡터들에 대한 행렬
K는 key 벡터들에 대한 행렬
V는 Value 벡터들에 대한 행렬
K^T는 K 행렬의 전치행렬
QK^T는 Query 벡터들과 Key 벡터들의 내적연산

위의 공식을 정리하면 아래와 같이 표현할 수 있습니다.

$$Q = \begin{bmatrix}2 & 0 & 2\\ 1 & 0 & 1\\ 1 & 0 & 2\end{bmatrix}, K = \begin{bmatrix}3 & 2 & 1\\ 2 & 2 & 0\\ 2 & 1 & 2\end{bmatrix}, V = \begin{bmatrix}1 & 2 & 2\\ 1 & 1 & 1\\ 1 & 2 & 2\end{bmatrix}$$

$$, K^T = \begin{bmatrix}3 & 2 & 2\\ 2 & 2 & 1\\ 1 & 0 & 2\end{bmatrix}$$

첫번째 단어에 대한 attention score

$$QK^T = \begin{bmatrix}2 & 0 & 2\\ 1 & 0 & 1\\ 1 & 0 & 2\end{bmatrix} \times \begin{bmatrix}3 & 2 & 2\\ 2 & 2 & 1\\ 1 & 0 & 2\end{bmatrix} = \begin{bmatrix}8 & 4 & 8\\ 4 & 2 & 4\\ 5 & 2 & 6\end{bmatrix}$$

의 값을 softmax 함수를 적용하여 아래와 같이 표현합니다.

$$\text{Softmax}(QK^T) = \text{Softmax}\left(\begin{bmatrix}8 & 4 & 8\\ 4 & 2 & 4\\ 5 & 2 & 6\end{bmatrix}\right) \approx \begin{bmatrix}0.5 & 0 & 0.5\\ 0.47 & 0.06 & 0.47\\ 0.27 & 0.01 & 0.72\end{bmatrix}$$

softmax에서 출력된 가중치의 값을 value 벡터와 내적연산을 사용하여 다음과 같은 결과를 출력합니다.

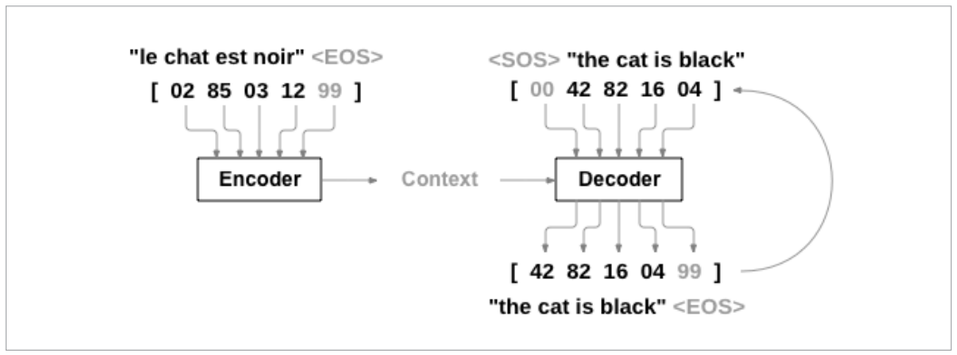
$$\text{Attention}(Q, K, V) = \text{Softmax}(QK^T) \times V$$
$$\approx \begin{bmatrix}0.5 & 0 & 0.5\\ 0.47 & 0.06 & 0.47\\ 0.27 & 0.01 & 0.72\end{bmatrix} \times \begin{bmatrix}1 & 2 & 2\\ 1 & 1 & 1\\ 1 & 2 & 2\end{bmatrix} = \begin{bmatrix}1 & 2 & 2\\ 1 & 1.94 & 1.94\\ 1 & 1.99 & 1.99\end{bmatrix}$$

각각의 행의 값은 해당 단어의 셀프-어텐션의 값의 결과값입니다.

실습

pytorch를 사용하여 실습을 진행하겠습니다. 아래와 같이 실습에 필요한 라이브러리를 import를 진행합니다. 해당 예제는 불어를 영어로 번역하는 예제입니다.

신경망을 사용하여 불어를 영어로 번역하도록 학습시킬 것입니다. seq2seq를 사용해서 아래와 같이 구상할 수 있습니다. Encoder는 입력 시퀀스 데이터를 최종 hidden state인 context 벡터를 생성하고, Decoder는 해당 벡터를 새로운 시퀀스로 펼치게 됩니다.



이 모델은 context 벡터가 가지는 이전 단어의 대한 정보량의 손실로 인하여 성능이 좋지 못하다고 언급한 바 있습니다. 여기에 어텐션을 적용하여 Decoder가 입력 시퀀스의 특정 범위에 집중할 수 있도록 합니다.

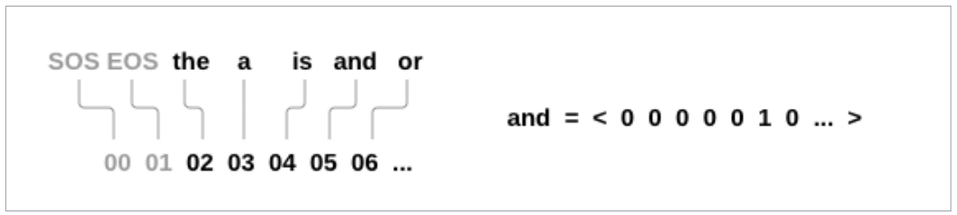
제공된 파일의 구조는 아래와 같이 영어 - 프랑스어로 구성되어 있습니다.

```
Run!    Cours !
Run!    Courez !
Wow!    Ça alors !
Fire!   Au feu !
Help!   À l'aide !
Jump.   Saute.
```


실습에 필요한 라이브러리를 import 합니다. 해당 예제는 pytorch로 되어 있습니다.

```
from __future__ import unicode_literals, print_function, division
from io import open
import unicodedata
import string
import re
import random

import torch
import torch.nn as nn
from torch import optim
import torch.nn.functional as F

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

각 단어들은 One-Hot 벡터로 표현합니다. One-hot은 단어의 수가 많아지면 벡터의 크기가 매우 커지는 단점이 있습니다. 예제에서는 약간의 다른 방식을 사용합니다. 단어마다 index 값을 부여하여 이 값을 사용할 것입니다.



입력 및 목표로 사용하려면 단어 당 고유 index 번호가 필요합니다. 이 모든 것을 추적하기 위해 단어→색인(word2index)과 색인→단어(index2word) 사전, 그리고 나중에 희귀 단어를 대체하는데 사용할 각 단어의 빈도 word2count 를 가진 Lang 이라는 헬퍼 클래스를 사용합니다

```
SOS_token = 0
EOS_token = 1

class Lang:
    def __init__(self, name):
        self.name = name
        self.word2index = {}
        self.word2count = {}
        self.index2word = {0: “SOS”, 1: “EOS”}
        self.n_words = 2  # SOS 와 EOS 포함

    def addSentence(self, sentence):
        for word in sentence.split(‘ ‘):
            self.addWord(word)

    def addWord(self, word):
        if word not in self.word2index:
            self.word2index[word] = self.n_words
            self.word2count[word] = 1
            self.index2word[self.n_words] = word
            self.n_words += 1
        else:
            self.word2count[word] += 1
```


제공된 파일은 모두 유니코드로 되어있어 간단하게 하기 위해 유니코드 문자를 ASCII로 변환하고, 모든 문자를 소문자로 만들고, 대부분의 구두점을 지워주는 전처리 작업을 진행합니다.

```
def unicodeToAscii(s):
    return ‘’.join(
        c for c in unicodedata.normalize(‘NFD’, s)
        if unicodedata.category(c) != ‘Mn’
    )

# 소문자, 다듬기, 그리고 문자가 아닌 문자 제거

def normalizeString(s):
    s = unicodeToAscii(s.lower().strip())
    s = re.sub(r”([.!?])”, r” ₩1”, s)
    s = re.sub(r”[^a-zA-Z.!?]+”, r” “, s)
    return s
```

파일을 읽기 위해 파일을 행 단위로 읽은 위에서 선언한 normalizeString 함수를 사용하여 정규화를 진행합니다. 해당 작업에서 필요한 문자는 삭제될 것입니다. 제공된 파일의 데이터가 영어-불어로 되어 있기 때문에 불어-영어로 변환하기 위한 reverse라는 조건문도 삽입하였습니다.

```
def readLangs(lang1, lang2, reverse=False):
    print(“Reading lines...”)

    # 파일을 읽고 줄로 분리
    lines = open(‘data/%s-%s.txt’ % (lang1, lang2), encoding=’utf-8’).₩
        read().strip().split(‘₩n’)

    # 모든 줄을 쌍으로 분리하고 정규화
    pairs = [[normalizeString(s) for s in l.split(‘₩t’)] for l in lines]

    # 쌍을 뒤집고, Lang 인스턴스 생성
    if reverse:
        pairs = [list(reversed(p)) for p in pairs]
        input_lang = Lang(lang2)
        output_lang = Lang(lang1)
    else:
        input_lang = Lang(lang1)
        output_lang = Lang(lang2)

    return input_lang, output_lang, pairs
```

파일에는 많은 예제 문장이 있고 학습의 속도를 위해 비교적 짧고 간단한 문장으로만 데이터 셋을 정리합니다. 여기서 최대 길이는 10 단어 (종료 문장 부호 포함)이며 《I am》 또는 《He is》 등의 형태로 번역되는 문장으로 필터링 됩니다.

```
MAX_LENGTH = 10

eng_prefixes = (
    "i am ", "i m ",
    "he is", "he s ",
    "she is", "she s ",
    "you are", "you re ",
    "we are", "we re ",
    "they are", "they re "
)

def filterPair(p):
    return len(p[0].split(' ')) < MAX_LENGTH and ₩
        len(p[1].split(' ')) < MAX_LENGTH and ₩
        p[1].startswith(eng_prefixes)

def filterPairs(pairs):
    return [pair for pair in pairs if filterPair(pair)]
```


이제 생성한 함수를 이용하여 prepareData라는 함수를 만들어서 위 과정을 포함시켜 최종 데이터를 생성합니다.

```
def prepareData(lang1, lang2, reverse=False):
    input_lang, output_lang, pairs = readLangs(lang1, lang2, reverse)
    print("Read %s sentence pairs" % len(pairs))
    pairs = filterPairs(pairs)
    print("Trimmed to %s sentence pairs" % len(pairs))
    print("Counting words...")
    for pair in pairs:
        input_lang.addSentence(pair[0])
        output_lang.addSentence(pair[1])
    print("Counted words:")
    print(input_lang.name, input_lang.n_words)
    print(output_lang.name, output_lang.n_words)
    return input_lang, output_lang, pairs

input_lang, output_lang, pairs = prepareData('eng', 'fra', True)
print(random.choice(pairs))
```

```
Reading lines...
Read 135842 sentence pairs
Trimmed to 10599 sentence pairs
Counting words...
Counted words:
fra 4345
eng 2803
['nous sommes en train de dejeuner .', 'we re having dinner .']
```

seq2seq 모델을 설계하겠습니다. 먼저 Encoder 부분을 설계하겠습니다. Seq2Seq의 Encoder는 입력 문장의 모든 단어에 대해 어떤 값을 출력하는 순환신경망(RNN)입니다. 모든 입력 단어에 대해 Encoder는 벡터와 은닉 상태를 출력하고 다음 입력 단어를 위해 그 은닉 상태를 사용합니다.

여기서 순환신경망의 GRU를 사용하여 Encoder를 설계하였습니다.

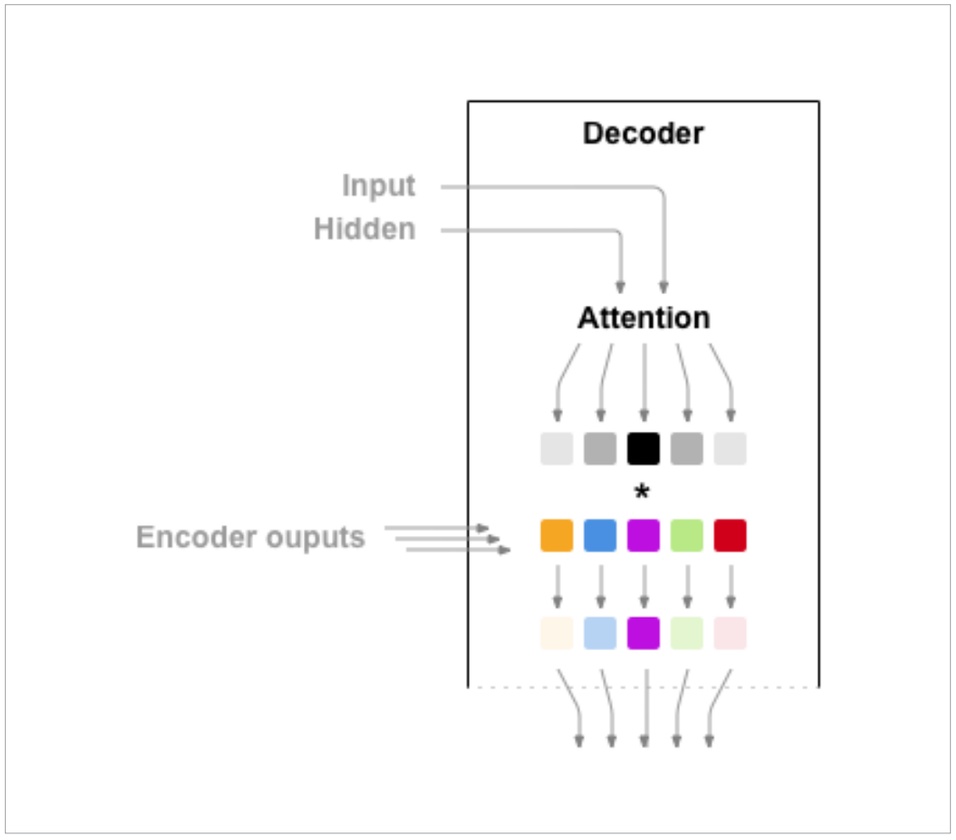
```
class EncoderRNN(nn.Module):
    def __init__(self, input_size, hidden_size):
        super(EncoderRNN, self).__init__()
        self.hidden_size = hidden_size

        self.embedding = nn.Embedding(input_size, hidden_size)
        self.gru = nn.GRU(hidden_size, hidden_size)

    def forward(self, input, hidden):
        embedded = self.embedding(input).view(1, 1, -1)
        output = embedded
        output, hidden = self.gru(output, hidden)
        return output, hidden

    def initHidden(self):
        return torch.zeros(1, 1, self.hidden_size, device=device)
```

Decoder는 Encoder에서 출력된 context 벡터 값을 전달받아 번역을 생성하기 위한 단어 시퀀스를 출력합니다.

```
class DecoderRNN(nn.Module):
    def __init__(self, hidden_size, output_size):
        super(DecoderRNN, self).__init__()
        self.hidden_size = hidden_size

        self.embedding = nn.Embedding(output_size, hidden_size)
        self.gru = nn.GRU(hidden_size, hidden_size)
        self.out = nn.Linear(hidden_size, output_size)
        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self, input, hidden):
        output = self.embedding(input).view(1, 1, -1)
        output = F.relu(output)
        output, hidden = self.gru(output, hidden)
        output = self.softmax(self.out(output[0]))
        return output, hidden

    def initHidden(self):
        return torch.zeros(1, 1, self.hidden_size, device=device)
```

seq2seq는 context에는 문장이 길어질 수록 처음 등장하는 단어에 대한 정보량이 없는 문제가 있다고 이야기했습니다. 그래서 이번 예제에서는 어텐션 방식을 적용하여 진행합니다.

어텐션은 Decoder가 출력의 모든 단계에서 Encoder 출력의 다른 부분에 《집중》 할 수 있게 합니다. 첫째 어텐션 가중치의 세트를 계산합니다. 가중치 조합을 만들기 위해서 Encoder 출력 벡터와 곱해집니다. 그 결과(코드에서 attn_applied)는 입력 시퀀스의 특정 부분에 관한 정보를 포함해야 하고 따라서 Decoder가 알맞은 출력 단어를 선택하는 것을 도와줍니다.



어텐션 가중치 계산은 Decoder의 입력 및 은닉 상태를 입력으로 사용하는 다른 feed-forwad 계층인 attn으로 수행됩니다. 학습 데이터에는 모든 크기의 문장이 있기 때문에 이 계층을 실제로 만들고 학습시키려면 적용할 수 있는 최대 문장 길이(Encoder 출력을 위한 입력 길이)를 선택해야 합니다. 최대 길이의 문장은 모든 Attention 가중치를 사용하지만 더 짧은 문장은 처음 몇 개만 사용합니다.

```
class AttnDecoderRNN(nn.Module):
    def __init__(self, hidden_size, output_size, dropout_p=0.1, max_length=MAX_LENGTH):
        super(AttnDecoderRNN, self).__init__()
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.dropout_p = dropout_p
        self.max_length = max_length

        self.embedding = nn.Embedding(self.output_size, self.hidden_size)
        self.attn = nn.Linear(self.hidden_size * 2, self.max_length)
        self.attn_combine = nn.Linear(self.hidden_size * 2, self.hidden_size)
        self.dropout = nn.Dropout(self.dropout_p)
        self.gru = nn.GRU(self.hidden_size, self.hidden_size)
        self.out = nn.Linear(self.hidden_size, self.output_size)

    def forward(self, input, hidden, encoder_outputs):
        embedded = self.embedding(input).view(1, 1, -1)
        embedded = self.dropout(embedded)

        attn_weights = F.softmax(
            self.attn(torch.cat((embedded[0], hidden[0]), 1)), dim=1)
        attn_applied = torch.bmm(attn_weights.unsqueeze(0),
                                 encoder_outputs.unsqueeze(0))

        output = torch.cat((embedded[0], attn_applied[0]), 1)
        output = self.attn_combine(output).unsqueeze(0)

        output = F.relu(output)
        output, hidden = self.gru(output, hidden)

        output = F.log_softmax(self.out(output[0]), dim=1)
        return output, hidden, attn_weights

    def initHidden(self):
        return torch.zeros(1, 1, self.hidden_size, device=device)
```

모델 학습을 위해서 학습 데이터를 torch의 텐서 유형으로 변환합니다. 학습을 위해서, 각 쌍마다 입력 Tensor(입력 문장의 단어 주소)와 목표 Tensor(목표 문장의 단어 주소)가 필요합니다. 이 벡터들을 생성하는 동안 두 시퀀스에 EOS 토큰을 추가 합니다.

```
def indexesFromSentence(lang, sentence):
    return [lang.word2index[word] for word in sentence.split(' ')]

def tensorFromSentence(lang, sentence):
    indexes = indexesFromSentence(lang, sentence)
    indexes.append(EOS_token)
    return torch.tensor(indexes, dtype=torch.long, device=device).view(-1, 1)

def tensorsFromPair(pair):
    input_tensor = tensorFromSentence(input_lang, pair[0])
    target_tensor = tensorFromSentence(output_lang, pair[1])
    return (input_tensor, target_tensor)
```

모델 학습을 위해서 Encoder에 입력 문장을 넣고 모든 출력과 최신 은닉 상태를 추적합니다. 그런 다음 Decoder에 첫 번째 입력으로 <SOS> 토큰과 Encoder의 마지막 은닉 상태가 첫 번째 은닉 상태로 제공됩니다.

《Teacher forcing》은 다음 입력으로 Decoder의 예측을 사용하는 대신 실제 목표 출력을 다음 입력으로 사용하는 컨셉입니다. 《Teacher forcing》을 사용하면 수렴이 빨리 되지만 학습된 네트워크가 잘못 사용될 때 불안정성을 보입니다.

Teacher-forced 네트워크의 출력이 일관된 문법으로 읽지만 정확한 번역과는 거리가 멀다는 것을 볼 수 있습니다. 직관적으로 출력 문법을 표현하는 법을 배우고 교사가 처음 몇 단어를 말하면 의미를 《선택》 할 수 있지만, 번역에서 처음으로 문장을 만드는 법은 잘 배우지 못합니다.

Teacher-forced에 대한 내용은 뒤에서 설명할 Transformer의 Decoder 부분에 설명되어 있으니 해당 내용을 참고하시기 바랍니다.

PyTorch의 autograd 가 제공하는 자유 덕분에 간단한 If 문으로 Teacher Forcing을 사용할지 아니면 사용하지 않을지를 선택할 수 있습니다.

```
teacher_forcing_ratio = 0.5
def train(input_tensor, target_tensor, encoder, decoder, encoder_optimizer,
          decoder_optimizer,  criterion, max_length=MAX_LENGTH):
    encoder_hidden = encoder.initHidden()

    encoder_optimizer.zero_grad()
    decoder_optimizer.zero_grad()
    input_length = input_tensor.size(0)
    target_length = target_tensor.size(0)

    encoder_outputs = torch.zeros(max_length, encoder.hidden_size, device=device)

    loss = 0

    for ei in range(input_length):
        encoder_output, encoder_hidden = encoder(
            input_tensor[ei], encoder_hidden)
        encoder_outputs[ei] = encoder_output[0, 0]

    decoder_input = torch.tensor([[SOS_token]], device=device)
    decoder_hidden = encoder_hidden
    use_teacher_forcing = True if random.random() < teacher_forcing_ratio else False

    if use_teacher_forcing:
        # Teacher forcing 포함: 목표를 다음 입력으로 전달
        for di in range(target_length):
            decoder_output, decoder_hidden, decoder_attention = decoder(
                decoder_input, decoder_hidden, encoder_outputs)
            loss += criterion(decoder_output, target_tensor[di])
            decoder_input = target_tensor[di]  # Teacher forcing

    else:
        # Teacher forcing 미포함: 자신의 예측을 다음 입력으로 사용
        for di in range(target_length):
            decoder_output, decoder_hidden, decoder_attention = decoder(
                decoder_input, decoder_hidden, encoder_outputs)
            topv, topi = decoder_output.topk(1)
            # 입력으로 사용할 부분을 히스토리에서 분리
            decoder_input = topi.squeeze().detach()
```

```
        loss += criterion(decoder_output, target_tensor[di])
        if decoder_input.item() == EOS_token:
            break

    loss.backward()

    encoder_optimizer.step()
    decoder_optimizer.step()

    return loss.item() / target_length
```

아래 함수들은 학습의 진행 상태를 확인할 수 있게 도와주는 헬퍼 함수입니다.

```
import time
import math

def asMinutes(s):
    m = math.floor(s / 60)
    s -= m * 60
    return '%dm %ds' % (m, s)

def timeSince(since, percent):
    now = time.time()
    s = now - since
    es = s / (percent)
    rs = es - s
    return '%s (- %s)' % (asMinutes(s), asMinutes(rs))
```

전체 학습 과정은 다음과 같습니다:

- 타이머 시작

- optimizers와 criterion 초기화

- 학습 쌍의 세트 생성

- 도식화를 위한 빈 손실 배열 시작

그런 다음 우리는 여러 번 train을 호출하며 때로는 진행률 (예제의 %, 현재까지의 예상 시간)과 평균 손실을 출력합니다.

```
def trainIters(encoder, decoder, n_iters, print_every=1000, plot_every=100,
               learning_rate=0.01):
    start = time.time()
    plot_losses = []
    print_loss_total = 0  # print_every 마다 초기화
    plot_loss_total = 0  # plot_every 마다 초기화

    encoder_optimizer = optim.SGD(encoder.parameters(), lr=learning_rate)
    decoder_optimizer = optim.SGD(decoder.parameters(), lr=learning_rate)
    training_pairs = [tensorsFromPair(random.choice(pairs))
                      for i in range(n_iters)]
    criterion = nn.NLLLoss()

    for iter in range(1, n_iters + 1):
        training_pair = training_pairs[iter - 1]
        input_tensor = training_pair[0]
        target_tensor = training_pair[1]

        loss = train(input_tensor, target_tensor, encoder,
                     decoder, encoder_optimizer, decoder_optimizer, criterion)
        print_loss_total += loss
        plot_loss_total += loss

        if iter % print_every == 0:
            print_loss_avg = print_loss_total / print_every
            print_loss_total = 0
            print('%s (%d %d%%) %.4f' % (timeSince(start, iter / n_iters),
                                         iter, iter / n_iters * 100, print_loss_avg))

        if iter % plot_every == 0:
            plot_loss_avg = plot_loss_total / plot_every
            plot_losses.append(plot_loss_avg)
            plot_loss_total = 0
```

이제 학습에 필요한 구성이 되었으니 실제로 아래 코드를 사용해서 학습을 진행합니다.

```
hidden_size = 256
encoder1 = EncoderRNN(input_lang.n_words, hidden_size).to(device)
attn_decoder1 = ₩
  AttnDecoderRNN(hidden_size, output_lang.n_words, dropout_p=0.1).to(device)

trainIters(encoder1, attn_decoder1, 75000, print_every=5000)
```

```
1m 14s (- 17m 29s) (5000 6%) 2.8785
2m 30s (- 16m 18s) (10000 13%) 2.2923
3m 46s (- 15m 4s) (15000 20%) 1.9435
5m 1s (- 13m 50s) (20000 26%) 1.6941
6m 18s (- 12m 36s) (25000 33%) 1.4950
7m 35s (- 11m 23s) (30000 40%) 1.3658
8m 55s (- 10m 11s) (35000 46%) 1.2174
10m 13s (- 8m 57s) (40000 53%) 1.0299
11m 32s (- 7m 41s) (45000 60%) 0.9583
12m 51s (- 6m 25s) (50000 66%) 0.8735
14m 9s (- 5m 9s) (55000 73%) 0.7904
15m 28s (- 3m 52s) (60000 80%) 0.7395
16m 46s (- 2m 34s) (65000 86%) 0.6778
18m 4s (- 1m 17s) (70000 93%) 0.6129
19m 23s (- 0m 0s) (75000 100%) 0.5428
```

학습 데이터 세트에 있는 임의의 문장을 평가하기 위해서 실제 데이터 쌍과 함께 모델이 생성한 값을 같이 출력하는 함수를 작성해서 값을 확인하겠습니다.

```
def evaluateRandomly(encoder, decoder, n=10):
    for i in range(n):
        pair = random.choice(pairs)
        print('>', pair[0])
        print('=', pair[1])
        output_words, attentions = evaluate(encoder, decoder, pair[0])
        output_sentence = ' '.join(output_words)
        print('<', output_sentence)
        print('')


evaluateRandomly(encoder1, attn_decoder1)
```



```
> vous travaillez dur .
= you re working hard .
< you re working hard hard . <EOS>

> je suis fourbu .
= i am exhausted .
< i am exhausted . <EOS>

> nous n allons pas le jeter .
= we re not throwing it away .
< we re not going . . <EOS>

> on ne m autorise pas a partir .
= i m not allowed to leave .
< i m not going to go . <EOS>

> nous nous ennuyons toutes .
= we re all bored .
< we re all bored . <EOS>

> je suis mediocre au tennis .
= i am poor at tennis .
< i m allergic at playing . <EOS>

> tu es une sacree menteuse .
...
```

>, =은 학습 데이터세트에 존재하는 데이터이고, <은 모델이 생성한 데이터입니다.

03
Transformer

트랜스포머는 2017년에 “Attention Is All You Need”라는 논문을 통해서 발표되었습니다.

[그림 3-22] 트랜스포머 모델 아키텍쳐

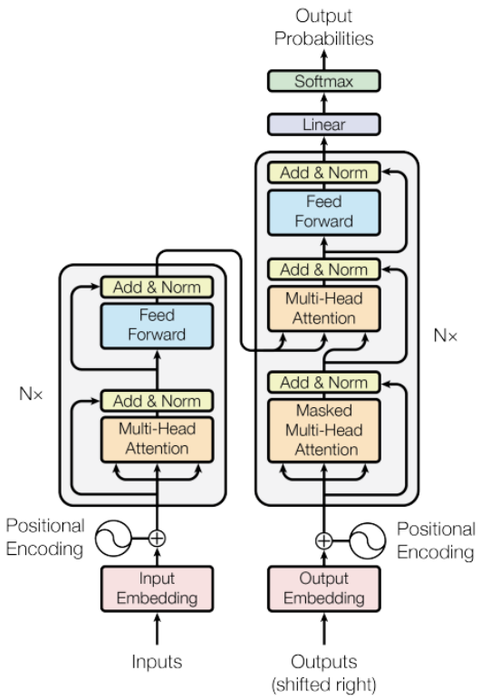
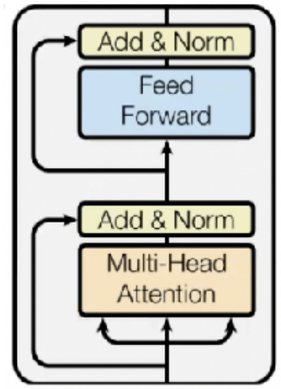


그림 3-22에서 Encoder block과 Decoder block을 여러 개 쌓아서 encoder 부분과 decoder 부분을 생성한 구조입니다. 논문에서는 6개의 block를 사용했습니다. Decoder 부분은 언어 모형으로 동작합니다.

Encoder block에 사용된 어텐션은 셀프-어센텬이 사용되었습니다. Decoder block에서는 2가지 어텐션이 사용되었습니다. 셀프-어텐션과 Encoder-Decoder 어텐션이 사용되었습니다.

Encoder 부분

[그림 3-23] Encoder block



Encoder block을 먼저 알아보겠습니다. 논문에서는 Encoder block를 6개를 사용했다고 합니다. Encoder block안에는 각각의 레이어로 구성되어 있는데 첫 번째로 Multi-Head Attention이 나옵니다. 여기서 Attention은 셀프-어텐션(Self-Attention)을 의미하며, 입력된 Sequence data에 대한 셀프-어텐션을 적용합니다. 앞에 Multi-head는 셀프-어텐션을 여러 개 적용했다는 것을 의미합니다.

[그림 3-24] Details of Multi-Head Attention

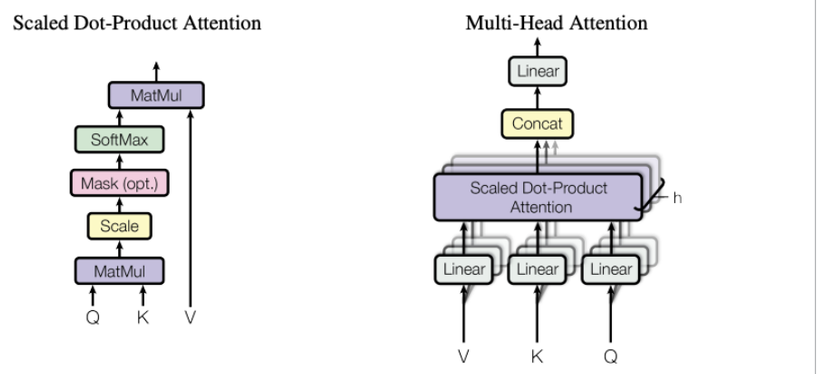


그림 3-24의 오른쪽 그림은 Scaled Dot-Product Attention의 h값을 논문에서는 8개를 사용했습니다. 왼쪽 그림은 해당 구조를 자세히 표현한 그림입니다. Encoder에서는 Mask는 사용되지 않고 Decoder에서만 사용되었습니다. 앞에서 이야기한 셀프 어텐션과 다른 점은 Scale 연산이 들어갔다는 점입니다. Scale 연산이 들어간 공식은 아래와 같습니다.

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

d_k는 key 벡터의 크기를 의미합니다. 논문에서는 64라는 값을 사용했습니다.

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_\text{h})W^O$$
$$\text{where head}_\text{i} = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

입력으로 들어가는 V, K, Q 값을 바로 사용하지 않고, 또 다른 가중치 행렬을 곱하여 사용합니다.(Linear Projection) Q, K, V에 속한 각각의 벡터의 차원수는 512 차원을 가지고 있습니다. 해당 차원을 Linear 부분에서 가중치 행렬($W_i^Q W_i^k W_i^v$)을 이용하여 특정한 차원의 크기로 변환합니다. 논문에서는 64차원으로 변환합니다.

[그림 3-25] 가중치 행렬을 사용한 벡터 변환

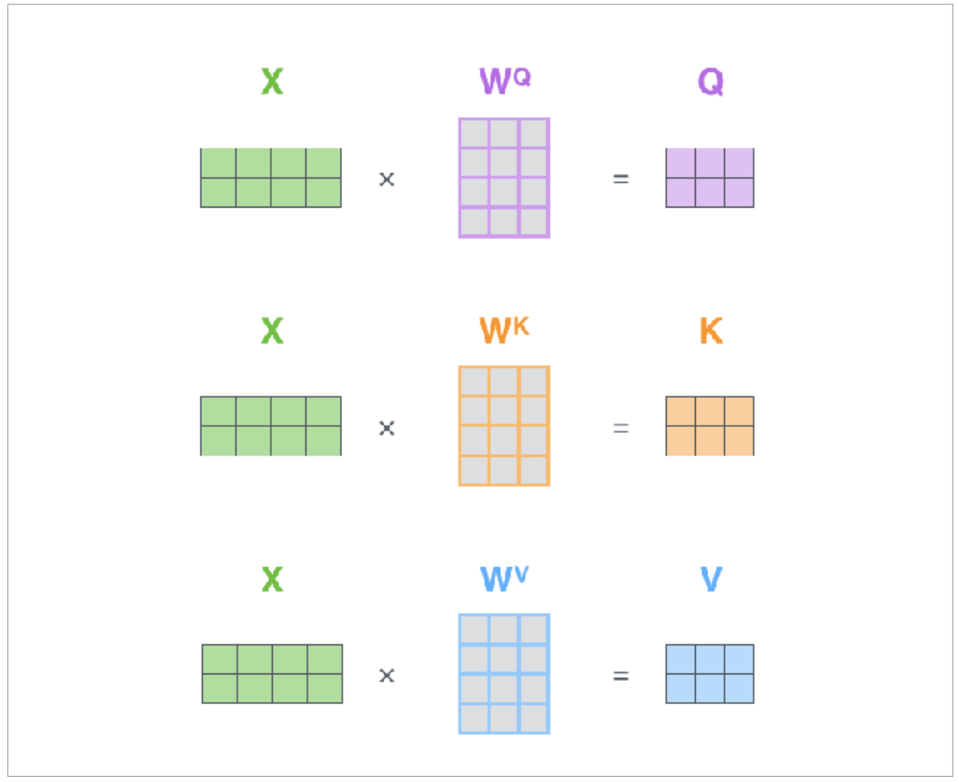


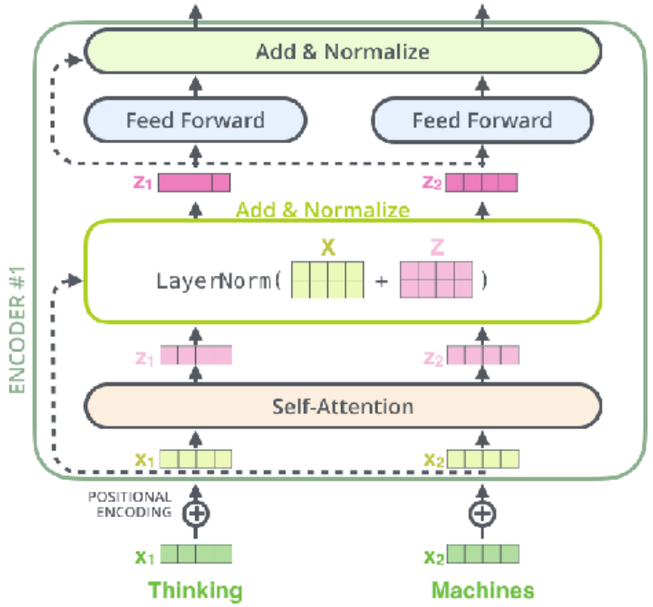
그림 3-23에서 Scaled Dot-Product Attention의 h값(논문에서는 8)만큼 나온 결과물을 Concat를 통해서 결과물을 만듭니다.

concat를 통과한 값의 차원수는 64차원이 8번 사용되었기 때문에 64x8=512 크기의 차원이 concat의 결과물의 차원수가 될 것입니다.

셀프-어텐션을 여러 번 사용한 이유는 주목해야 하는 다른 단어가 무엇인지를 더 잘 파악할 수 있고 각 단어가 가지고 있는 문맥적 특성을 더 잘 표현을 할 수 있습니다.

Multi-Head Attention의 결과 값과 단어의 임베딩 값이 Add & Norm 부분에 입력값으로 들어갑니다. 이 부분에서 셀프-어텐션 층이 출력한 값과 셀프-어텐션에 입력된 값을 더합니다. 이 값을 각 관측치에 대해 하나의 레이어에 존재하는 노드들의 입력값들의 정보를 이용해서 표준화하는 LayerNorm를 사용합니다.

[그림 3-26] Add & Norm의 구조



출처 : http://jalammar.github.io/illustrated-transformer/

Feed Forward에서는 Position-wise feed-forward network를 의미합니다. Position-wise는 각각의 단어에 대한 서로 다른 FCL(fully connected layer)를 2개를 사용합니다.

$$\mathrm{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

첫번째 Layer에서는 Relu 활성화 함수를 사용하며, 두번째는 활성화 함수를 사용하지 않습니다.

x는 첫번째 FCL에 입력되는 입력 벡터를 의미 함
W_1는 입력벡터와 첫번째 FCL 사이의 가중치 행렬
W_2는 첫번째 FCL과 두번째 FCL 사이의 가중치 행렬

Encoder block으로 Input Embedding 벡터값이 들어가기 전에 Positional Embedding 연산을 진행합니다. Transformer 모형에서는 단어들의 embedding 정보만을 사용하는 것이 아니라 단어들이 갖는 입력된 sequence data 내에서의 위치(position) 정보도 같이 사용합니다.

위치 정보를 사용하게 되면, 단어들 간의 위치를 파악함으로써 단어들 간의 상대적인 거리를 파악할수 있습니다. 이를 사용하는 주된 이유는 트랜스포머는 RNN이나 LSTM같은 순환신경망 구조를 사용하지 않고 어텐션 방법만을 사용하기 때문입니다.

단어들이 갖는 상대적인 위치 정보를 반영하기 위해서 위치 정보를 반영하는 벡터를 사용하는데 이를 위치정보 임베딩 벡터(positional embedding vector)라고 합니다. 위치정보 임베딩은 아래와 같이 계산할 수 있습니다.

최종 embedding	(e0,0, e0,1, ..., e0,n)	(e1,0, e1,1, ..., e1,n)	(e2,0, e2,1, ..., e2,n)
	=	=	=
Positional embedding	(p0,0, p0,1, ..., p0,n)	(p1,0, p1,1, ..., p1,n)	(p2,0, p2,1, ..., p2,n)
	+	+	+
원래의 embedding	(x0,0, x0,1, ..., x0,n)	(x1,0, x1,1, ..., x1,n)	(x2,0, x2,1, ..., x2,n)
입력단어	단어0	단어1	단어2

이렇게 만들어진 최종 embedding값이 Encoder block으로 입력값으로 들어갑니다. 그럼 Positional embedding 값은 어떻게 계산되는지 알아보겠습니다.

Positional embedding은 sin, cos 함수를 이용하여 계산할 수 있습니다.

$$PE_{(i,j)} = \sin\left(\frac{i}{10000^{j/emb\_dim}}\right), \quad \text{j가 짝수인 경우}$$

$$PE_{(i,j)} = \cos\left(\frac{i}{10000^{j-1/emb\_dim}}\right), \quad \text{j가 홀수인경우}$$

PE의 값은 단어 i의 positional embedding vecotr의 위치 j의 원소값을 의미합니다.

Today is Monday라는 문장이 입력된다고 하면 각각의 단어에 대한 positional embedding 벡터를 계산을 아래와 같이 할 수 있습니다. 여기서 embedding 벡터의 차원은 512라고 하겠습니다.

$$\begin{matrix} & j=0 & j=1 & \\ Today\ (i=0) & sin\left(\frac{0}{10000^{\frac{0}{512}}}\right) & cos\left(\frac{0}{10000^{\frac{0}{512}}}\right) & \cdots \\ is\ \ (i=1) & sin\left(\frac{1}{10000^{\frac{0}{512}}}\right) & cos\left(\frac{1}{10000^{\frac{0}{512}}}\right) & \cdots \\ Monday\ (i=2) & sin\left(\frac{2}{10000^{\frac{0}{512}}}\right) & cos\left(\frac{2}{10000^{\frac{0}{512}}}\right) & \cdots \end{matrix}$$

Positional embedding 값들은 학습이 되는 것이 아니라, 위의 공식으로 미리 계산되는 값입니다.

Decoder 부분

Decoder 부분은 6개의 decoder block이 사용되었습니다. Decoder block에서는 2개의 어텐션이 사용되었는데 첫번째는 셀프-어텐션, 두 번째는 Encoder-Decoder 어텐션입니다.

Decoder는 언어 모형의 역할을 하지만, 학습 단계에서는 모형이 현재 단계까지 예측한 단어들의 정보를 사용하여 다음 단어를 예측하는 것이 아니라, 정답 데이터 정보를 이용해서 각 단계의 단어들을 예측하는 방식으로 동작합니다. 이런 방식을 Teacher forcing이라고 합니다.

이런 방식을 사용하는 이유는 모형이 예측한 단어들의 정보를 이용해서 다음 단어를 예측하는 경우에는 이전 단어들에 대한 예측이 잘못되면 그 다음 단어에 대한 예측이 제대로 될 수 없기 때문입니다.

예를 들어 “오늘은 금요일 입니다”라는 단어를 “Today is Friday”라고 번역하는 경우 첫 단어인 “Today”를 정확히 예측하지 못하면 그 다음 단어인 “is”를 제대로 예측할 수 없습니다. 이를 위해 학습에서는 실제 정답 데이터를 사용하여 학습하게 됩니다.

Encoder에서 사용된 셀프-어텐션과는 약간 다르게 작동하는데 Masked 부분이 추가됩니다.

문장 앞에 <sos> 토큰을 추가하여 사용하는데 이를 right shifted output이라고 표현합니다.

Masked self-attention의 원리에 대해서 예를 들어 설명하겠습니다. “오늘은 금요일 입니다.”를 “Today is Friday”라고 번역하는 경우 아래처럼 <sos>, “Today”, “is”, “Friday”을 학습 데이터로 사용하여 각 단계에서의 단어를 예측합니다. 각 토큰의 임베딩 정보와 위치정보 임베딩 정보를 사용해서 Query, Key, Value 벡터를 계산하고 어텐션 스코어를 계산하게 됩니다. 이렇게 계산된 결과가 아래와 같다고 가정하겠습니다.

	<sos>	Today	is	Friday
<sos>	7	2	2	2
Today	1	6	2	4
is	1	2	8	1
Friday	1	4	2	6

Today에 대한
어텐션 스코어

Decoder는 언어 모형으로 작동하기 때문에 특정 단어의 어텐션 스코어 중에서 자기 자신 다음에 나오는 단어에 대한 정보를 사용할 수 없습니다. 예를 들어 두번째 단어(is)까지 정보를 사용해서 세 번째 단어를 예측하는 경우에 is 단어에 대한 셀프-어텐션 결과까지 사용해야 합니다.

is까지의 토큰인 <sos>, Today, is에 대한 정보만을 사용해서 셀프-어텐션을 계산하고 그 정보를 사용해서 계산된 가중치를 value 벡터에 곱해서 어텐션 결과물을 얻어야 합니다. 이를 위해서 기준이 되는 단어 이후에 나오는 단어들에 대한 어텐션-스코어를 아래처럼 -∞로 대체합니다.

	<sos>	Today	is	Friday
<sos>	7	-∞	-∞	-∞
Today	1	6	-∞	-∞
is	1	2	8	-∞
Friday	1	4	2	6

-∞로 되어 있는 부분은 softmax함수를 사용하면 공식에 따라 분자 부분에서 e-∞의 값이 0이 되어 해당되는 부분의 값들은 어텐션 결과물에서는 0으로 변환됩니다.

Decoder에서 사용하는 또 다른 어텐션인 Encoder-Decoder 어텐션(Encoder-Decoder attention)에 대해서 알아보겠습니다. 셀프-어텐션과 마찬가지로 Query, Key, Value 벡터들을 사용합니다.

Query 벡터는 Decoder 부분에 입력된 단어에 대한 Query 벡터이며(Add & Norm층의 결과) key와 value 벡터는 Encoder 부분의 마지막 Encoder block에서 출력하는 각 단어들에 대한 값을 사용하여 계산합니다. 그 외 작동 방식은 앞서 설명한 셀프-어텐션과 동일합니다.

4.

전이학습
소개 및 기본

1. 전이학습(Transfer Learning) 기법 소개
2. 사전훈련(Pre-training)과 미세조정(Fine-tuning)
3. BERT 모델 구조의 이해
4. GPT 모델 구조의 이해

01
전이학습 (Transfer Learning) 기법 소개

이 절에서는 사전훈련(Pre-training), 미세조정(Fine-tuning) 등 전이학습(Transfer Learning)과 관련된 개념을 설명합니다.

위키 백과에서는 전이학습을 다음과 같이 설명하고 있습니다.

"전이학습(Transfer Learning)은 하나의 문제를 해결하고 이와 다르면서 관련된 문제에 적용하는 동안 얻은 지식을 저장하는 데 집중하는 기계 학습의 연구 문제이다. 예를 들어 자동차를 인식하기 위해 학습하는 동안 얻은 지식은 트럭 인식을 시도할 때 적용할 수 있다."

다시 말하자면, 전이학습(Transfer Learning)이란 이란 특정 태스크를 학습한 모델을 다른 태스크 수행에 재사용하는 기법을 가리킵니다. 비유하자면 사람이 새로운 지식을 배울 때 그가 평생 쌓아왔던 지식을 요긴하게 다시 써먹는 것과 같습니다.

[그림 4-1] 전이학습 개념도

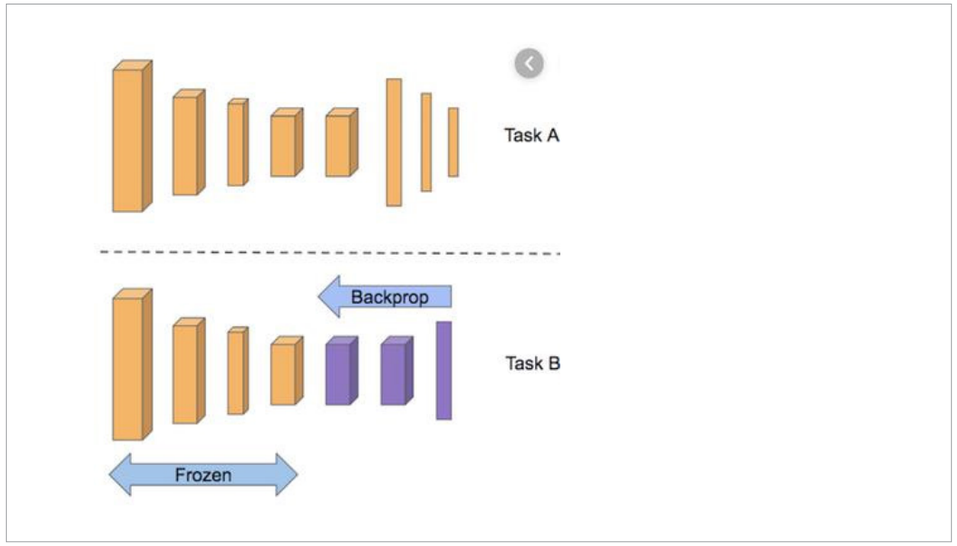


그림 4-1처럼 태스크 B를 수행하는 모델을 만든다고 가정해 보면 전이 학습이 도움이 될 수 있습니다. 모델이 태스크 B를 배울 때 태스크 A를 수행해 봤던 경험을 재활용하기 때문입니다.

전이학습을 적용하면 기존보다 모델의 학습 속도가 빨라지고 새로운 태스크를 더 잘 수행하는 경향이 있습니다. 이 때문에 전이학습은 최근 널리 쓰이고 있습니다. 본 교재에서 소개하는 자연어 처리 모델인 BERT나 GPT 등도 전이학습이 적용됐습니다.

위에서 설명한 장점으로 인하여, 전이학습은 해당 교재에서 소개하는 자연어 처리 모델에 사용되는 것뿐만 아니라 컴퓨터 비전, 음성 인식 등 다양한 분야의 문제를 우수한 성능으로 쉽게 해결하기 위해 많이 사용되는 개념입니다.

그림 4-1에서 태스크 A는 업스트림(upstream) 태스크라고 부르고 태스크 B는 이와 대비된 개념으로 다운스트림(downstream) 태스크라고 부릅니다. 태스크 A는 다음 단어 맞히기, 빈칸 채우기 등 대규모 말뭉치의 문맥을 이해하는 과제이며, 태스크 B는 문서 분류, 개체명 인식 등 우리가 풀고자 하는 자연어 처리의 구체적인 문제들입니다.

업스트림 태스크를 학습하는 과정을 사전훈련(pre-training)이라고 부릅니다. 다운스트림 태스크를 본격적으로 수행하기에 앞서(pre) 학습(train)한다는 의미입니다.

02
사전훈련 (Pre-training)과 미세조정 (Fine-tuning)

사전훈련(Pre-training)

전이학습이 주목받게 된 것은 업스트림 태스크와 사전훈련(Pre-training) 덕분입니다. 자연어의 풍부한 문맥(context)을 모델에 내재화하고 이 모델을 다양한 다운스트림 태스크에 적용해 성능을 대폭 끌어올리게 된 것입니다.

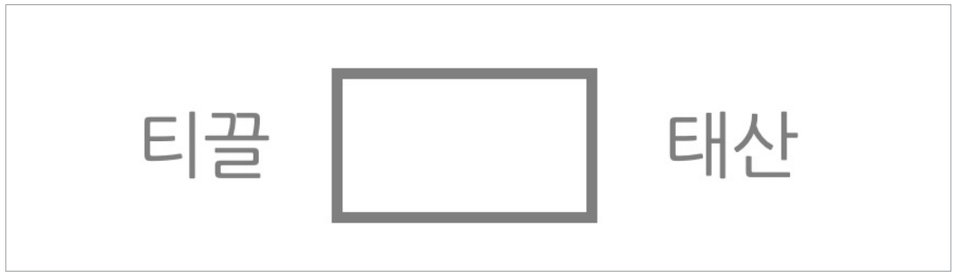
[그림 4-2] 다음 단어 맞히기

대표적인 업스트림 태스크 가운데 하나가 다음 단어 맞히기입니다. GPT 계열 모델이 바로 이 태스크로 사전훈련을 수행합니다. 예를 들어 그림 4-2처럼 티끌 모아라는 문맥이 주어졌고 학습 데이터 말뭉치에 티끌 모아 태산이라는 구(phrase)가 많다고 하면 모델은 이를 바탕으로 다음에 올 단어를 태산으로 분류하도록 학습됩니다.

모델이 대규모 말뭉치를 가지고 이런 과정을 반복 수행하면 이전 문맥을 고려했을 때 어떤 단어가 그 다음에 오는 것이 자연스러운지 알 수 있게 됩니다. 다시 말해 해당 언어의 풍부한 문맥을 이해할 수 잇게 되는 것입니다. 이처럼 다음 단어 맞히기로 업스트림 태스크를 수행한 모델을 언어 모델(language model)이라고 합니다.

언어 모델을 학습하는 것은 학습 대상 언어의 어휘 수 (보통 수만 개 이상)만큼의 범주를 분류하는 학습 과정과 같습니다. 예를 들어 티끌 모아 다음 단어의 정답이 태산이라면 태산이라는 단어에 해당하는 확률은 높이고, 나머지 단어들의 확률은 낮추는 방향으로 모델 전체를 업데이트합니다.

[그림 4-3] 빈칸 채우기



또 다른 업스트림 태스크로는 빈칸 채우기가 있습니다. BERT 계열 모델이 바로 이 태스크로 사전훈련을 수행합니다. 그림 4-3처럼 문장에서 빈칸을 만들고 해당 위치에 들어갈 단어가 무엇일지 맞히는 과정에서 학습됩니다.

모델이 많은 양의 데이터를 가지고 빈칸 채우기를 반복 학습하면 앞뒤 문맥을 보고 빈칸에 적합한 단어를 알 수 있습니다. 이 태스크를 수행한 모델 역시 언어 모델과 마찬가지로 해당 언어의 풍부한 문맥을 내재화 할 수 있습니다. 이처럼 빈칸 채우기로 업스트림 태스크를 수행한 모델을 마스크 언어 모델(masked language model)이라고 합니다.

마스크 언어 모델의 학습 역시 언어 모델과 비슷합니다. 그림 4-3에서 빈칸의 정답이 모아라면 모아라는 단어에 해당하는 확률을 높이고 나머지 단어와 관계된 확률은 낮추는 방향으로 모델 전체를 업데이트합니다.

다음 단어 맞히기, 빈칸 채우기 같은 업스트림 태스크는 강력한 힘을 지닙니다. 뉴스, 웹 문서, 백과사전 등 글만 있으면 수작업 없이도 다량의 학습 데이터를 아주 싼값에 만들어 낼 수 있습니다. 덕분에 업스트림 태스크를 수행한 모델은 성능이 기존보다 월등히 좋아졌습니다. 이처럼 데이터 내에서 정답을 만들

고 이를 바탕으로 모델을 학습하는 방법을 자기지도 학습(self-supervised learning)이라고 합니다.

이전에 학습 데이터는 사람이 일일이 정답(레이블)을 만들어 줘야 했습니다. 이처럼 사람이 만든 정답 데이터로 모델을 학습하는 방법을 지도 학습(supervised learning)이라고 합니다. 이 방식은 데이터를 만드는 데 비용이 많이 발생하고 사람이 실수로 잘못된 레이블을 줄 수도 있습니다.

결론적으로, 자연어 처리 분야에서 전이학습을 하기 위해 업스트림 태스크의 대표 과제인 다음 단어 맞추기와 빈칸 단어 채우기를 활용하여 사전훈련을 진행합니다. 사전훈련을 통해서 대규모 말뭉치를 미리 학습(pre-train)한 임베딩을 다운스트림 태스크 모델의 입력 값으로 쓰고, 해당 임베딩을 포함한 모델 전체를 다운스트림 태스크를 잘 해결할 수 있도록 업데이트(fine-tuning)하는 방식이 미세조정입니다.

임베딩(embedding)은 자연어를 숫자의 나열인 벡터(vector)로 바꾼 결과 혹은 그 일련의 과정 전체를 가리키는 용어입니다. 임베딩에는 말뭉치(corpus)의 의미, 문법 정보가 응축돼 있습니다. 임베딩은 벡터이기 때문에 사칙연산이 가능하며, 단어/문서 관련도(relevance) 역시 계산할 수 있습니다.

전이학습 혹은 사전훈련-미세조정 메커니즘은 사람의 학습과 비슷한 점이 있습니다. 사람은 무언가를 배울 때 제로 베이스에서 시작하지 않습니다. 사람이 새로운 사실을 빠르게 이해할 수 있는 이유는 그가 이해를 하는 데에 평생 쌓아 온 지식을 동원하기 때문입니다. 자연어 처리 모델 역시 제로에서 시작하지 않습니다. 우선 대규모 말뭉치를 학습시켜 임베딩을 미리 만들어 놓습니다(사전훈련). 이 임베딩에는 의미, 문법 정보가 녹아 있습니다. 이후 임베딩을 포함한 모델 전체를 다운스트림 태스크에 맞게 업데이트합니다(미세조정).

전이학습 모델은 제로부터 학습한 모델보다 다운스트림 태스크를 빠르게 잘 수행할 수 있습니다. 전이학습은 하나의 작업에 대해 훈련된 모델이 관련되어 있지만 다른 작업에 맞게 용도가 변경되고 미세 조정되는 기계 학습 기술입니다. 전이 학습의 기본 아이디어는 사전 훈련된 모델에서 학습한 지식을 활용하여 새롭지만 관련된 문제를 해결하는 것입니다. 이는 새 모델을 처음부터 훈련하는 데 사용할 수 있는 데이터가 제한적인 상황이나 새 작업이 원래 작업과 충분히 유사하여 사전 훈련된 모델을 약간의 수정만으로 새 문제에 적용할 수 있는 경우에 유용할 수 있습니다.

미세조정(Fine-tuning)

우리가 모델을 업스트림 태스크로 사전훈련한 근본적인 이유는 다운스트림 태스크를 잘 해결하고 싶기 때문입니다.

앞에서 설명했듯이 다운스트림 태스크는 우리가 풀어야 할 자연어 처리의 구체적인 과제들입니다. 보통 다운스트림 태스크는 사전훈련을 마친 모델을 구조 변경 없이 그대로 사용하거나 여기에 태스크 모듈을 덧붙인 형태로 수행합니다.

본 교재에서 소개하는 다운스트림 태스크의 본질은 분류(classification)입니다. 다시 말해 자연어를 입력 받아 해당 입력이 어떤 범주에 해당하는지 확률 형태로 반환합니다. 문장 생성을 제외한 대부분의 과제에서는 사전훈련을 마친 마스크 언어 모델(BERT 계열)을 사용합니다.

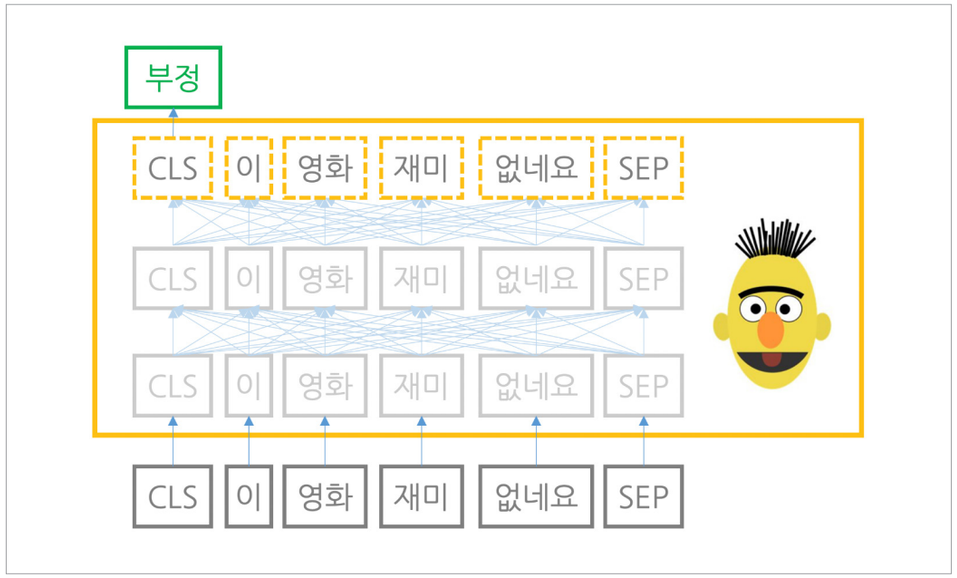
본 교재에서 설명하는 다운스트림 태스크의 학습 방식은 모두 미세조정(fine-tuning)입니다. 미세조정은 사전훈련을 마친 모델을 다운스트림 태스크에 맞게 업데이트 하는 기법입니다. 예를 들어 문서 분류를 수행할 경우 사전훈련을 마친 BERT 모델 전체를 문서 분류 데이터로 업데이트합니다. 마찬가지로 개체명 인식을 수행한다면 BERT 모델 전체를 해당 데이터로 업데이트합니다.

다운스트림 태스크는 자연어 처리의 구체적인 과제들입니다. 본 교재에서는 문서 분류, 자연어 추론, 개체명 인식, 질의응답, 문장 생성 등 총 5가지 과제에 대한 개념을 설명하고 5장에서 실습을 통해 문제를 해결하는 방안을 익힙니다.

다음은 자연어 처리의 다운스트림 태스크에 대한 개념 설명입니다.

● 문서 분류

[그림 4-4] 문서 분류



문서 분류 모델은 자연어(문서나 문장)를 입력 받아 해당 입력이 어떤 범주(긍정, 중립, 부정 따위)에 속하는지 그 확률 값을 반환합니다.

구체적으로는 사전훈련을 마친 마스크 언어 모델(노란색 실선 박스) 위에 작은 모듈(초록색 실선 박스)을 하나 더 쌓아 문서 전체의 범주를 분류합니다. 문서 분류 과제는 5-1장에서 실습합니다. 한편 그림에서 CLS, SEP는 각각 문장의 시작과 끝에 붙이는 특수한 토큰(token)입니다. 토큰 및 토큰화(tokenization)에 관한 자세한 내용은 4-3장에서 다룹니다.

● 자연어 추론

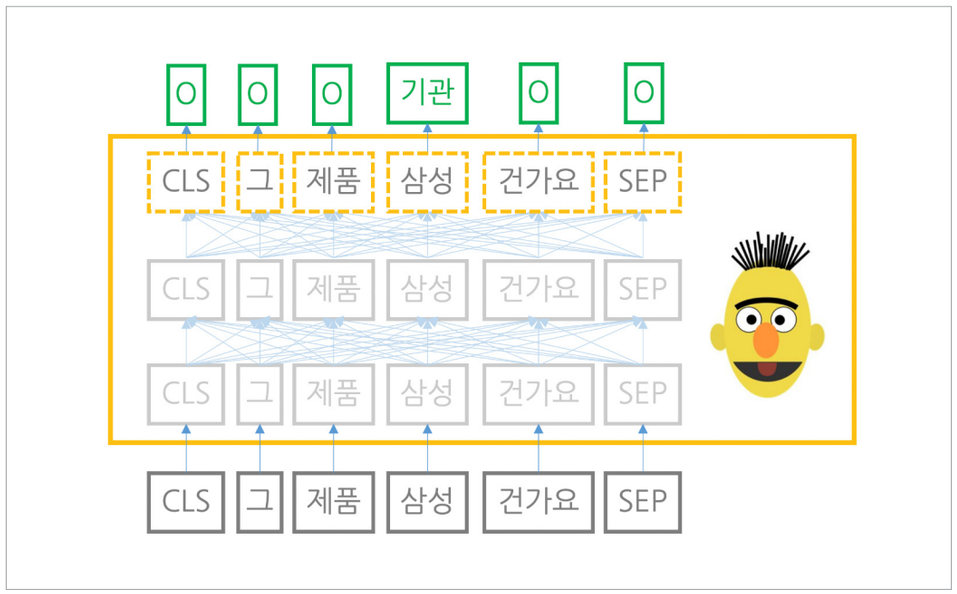
[그림 4-5] 자연어 추론

자연어 추론 모델은 문장 2개를 입력 받아 두 문장 사이의 관계가 참(entailment), 거짓 (contradiction), 중립(neutral)등 어떤 범주인지 그 확률 값을 반환합니다.

구체적으로는 사전훈련을 마친 마스크 언어 모델(노란색 실선 박스)위에 작은 모듈(초록색 실선 박스)을 하나 더 쌓아 두 문장의 관계 범주를 분류합니다. 자연어 추론 과제는 5-2장에서 실습합니다.

● 개체명 인식

[그림 4-6] 개체명 인식



개체명 인식 모델은 자연어(문서나 문장)를 입력 받아 단어별로 기관명, 인명, 지명 등 어떤 개체명 범주에 속하는지 그 확률 값을 반환합니다.

구체적으로는 사전훈련을 마친 마스크 언어 모델(노란색 실선 박스) 위에 단어별로 작은 모듈(초록색 실선 박스)을 쌓아 단어 각각의 개체명 범주를 분류합니다. 개체명 인식 과제는 5-3장에서 실습합니다.

● 질의응답

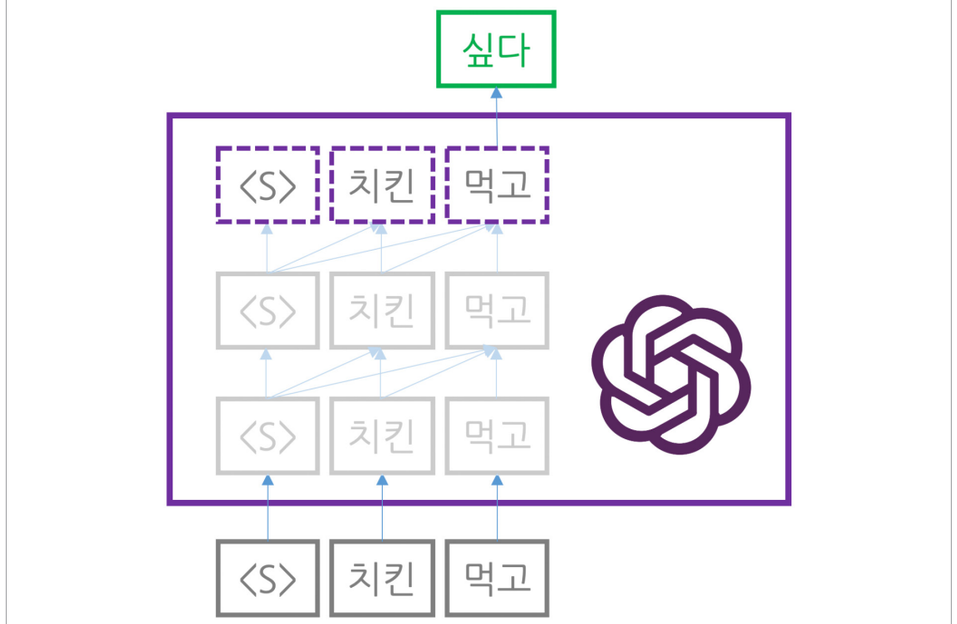
[그림 4-7] 질의응답

질의응답 모델은 자연어(질문+지문)를 입력 받아 각 단어가 정답의 시작일 확률 값과 끝일 확률 값을 반환합니다.

구체적으로는 사전훈련을 마친 마스크 언어 모델(노란색 실선 박스)위에 단어별로 작은 모듈을 쌓아 전체 단어 가운데 어떤 단어가 시작(초록색 실선 박스)인지 끝(붉은색 실선박스)인지 분류합니다. 질의응답 과제는 5-4장에서 실습합니다.

● 문장 생성

[그림 4-8] 문장 생성



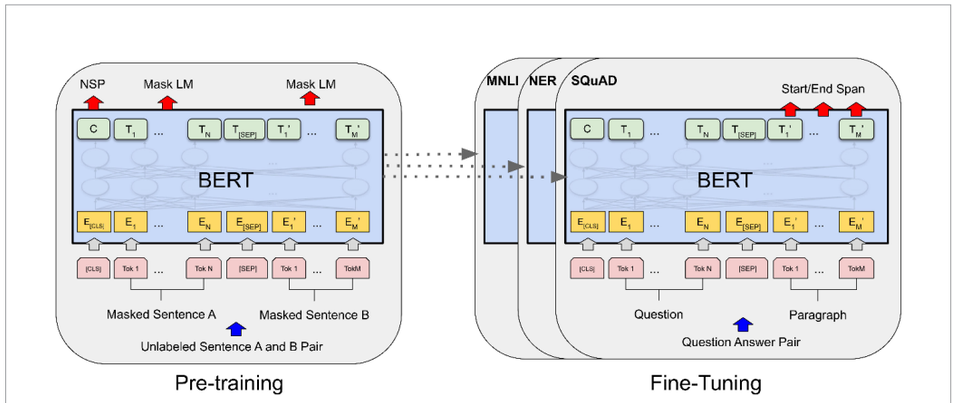
문장 생성 모델은 GPT 계열 언어 모델이 널리 쓰입니다. 문장 생성 모델은 자연어(문장)를 입력 받아 어휘 전체에 대한 확률 값을 반환합니다. 이 확률 값은 입력된 문장 다음에 올 단어로 얼마나 적절한지를 나타내는 점수입니다.

구체적으로는 사전훈련을 마친 언어 모델을 구조 변경 없이 그대로 사용해, 문맥에 이어지는 적절한 다음 단어를 분류하는 방식입니다. 문장 생성 과제는 5-5장에서 실습합니다.

03 BERT 모델 구조의 이해

본 교재의 4장의 3절과 4절에서는 트랜스포머 아키텍처를 기본 뼈대로 하는 BERT와 GPT 모델의 구조를 이해합니다. 또한, BERT와 GPT의 공통점과 차이점을 중심으로 살펴봅니다.

[그림 4-9] BERT 모델을 사용한 미세조정 아키텍처

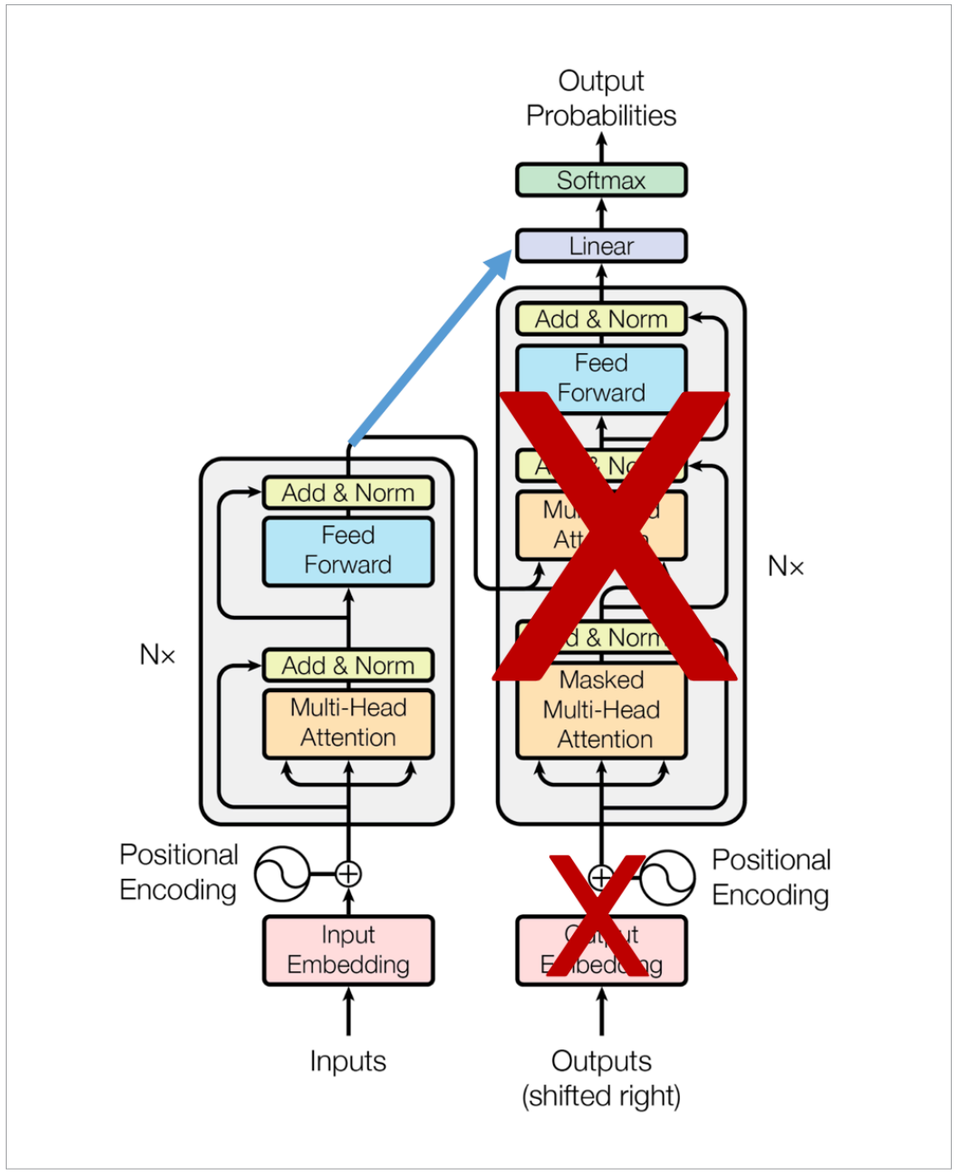


BERT(BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding)는 마스크 언어 모델입니다. 그림 4-9처럼 문장 중간에 빈칸을 만들고 해당 빈칸에 어떤 단어가 적절할지 맞히는 과정에서 사전훈련 합니다. 빈칸 앞뒤 문맥을 모두 살필 수 있다는 점에서 양방향(bidirectional) 성격을 가집니다.

BERT는 문장내 임의의 단어를 마스킹하고 이를 예측하도록 하는 마스크 언어 모델(Masked Language Model)이라고 말씀드렸습니다. 사전훈련 시 다음 문장 예측(Next Sentence Prediction)이라는 또 하나의 아이디어가 적용됩니다. 다음 문장 예측은 사전훈련 시 두 문장을 주고 두 번째 문장이 코퍼스

내에서 첫 번째 문장의 바로 다음에 오는지 여부를 예측하도록 하는 방식입니다. 자연어 추론이나 질의 응답 등의 다운스트림 태스크를 잘 해결하기 위해서 다음 문장을 예측하는 방법이 사용되었습니다.

[그림 4-10] BERT 구조



BERT는 문장의 의미를 추출하는데 강점을 갖게 됩니다. BERT는 그림 4-10처럼 트랜스포머 아키텍처에서 디코더를 제외하고 인코더만 취해 사용하는 구조입니다.

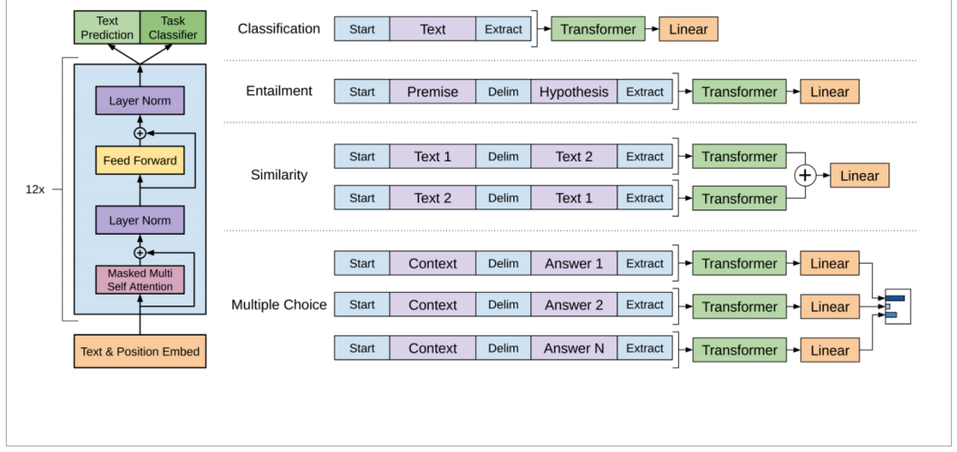
입력 단어 시퀀스가 "어제 카페 갔었어 [MASK] 사람 많더라"라고 가정해 보겠습니다. BERT는 마스크 토큰 앞뒤 문맥을 모두 참고할 수 있습니다. 앞뒤 정보를 준다고 해서 정답을 미리 알려주는 것이 아니기 때문입니다.

"[MASK]"라는 단어에 대응하는 BERT 마지막 레이어의 출력 결과에 선형 변환과 소프트맥스를 적용해 요솟값 각각이 확률이고 학습 대상 언어의 어휘 수만큼 차원 수를 가진 벡터가 되도록 합니다. 빈칸의 정답인 "거기"에 해당하는 확률은 높이고 나머지 단어의 확률은 낮아지도록 모델 전체를 업데이트합니다.

04

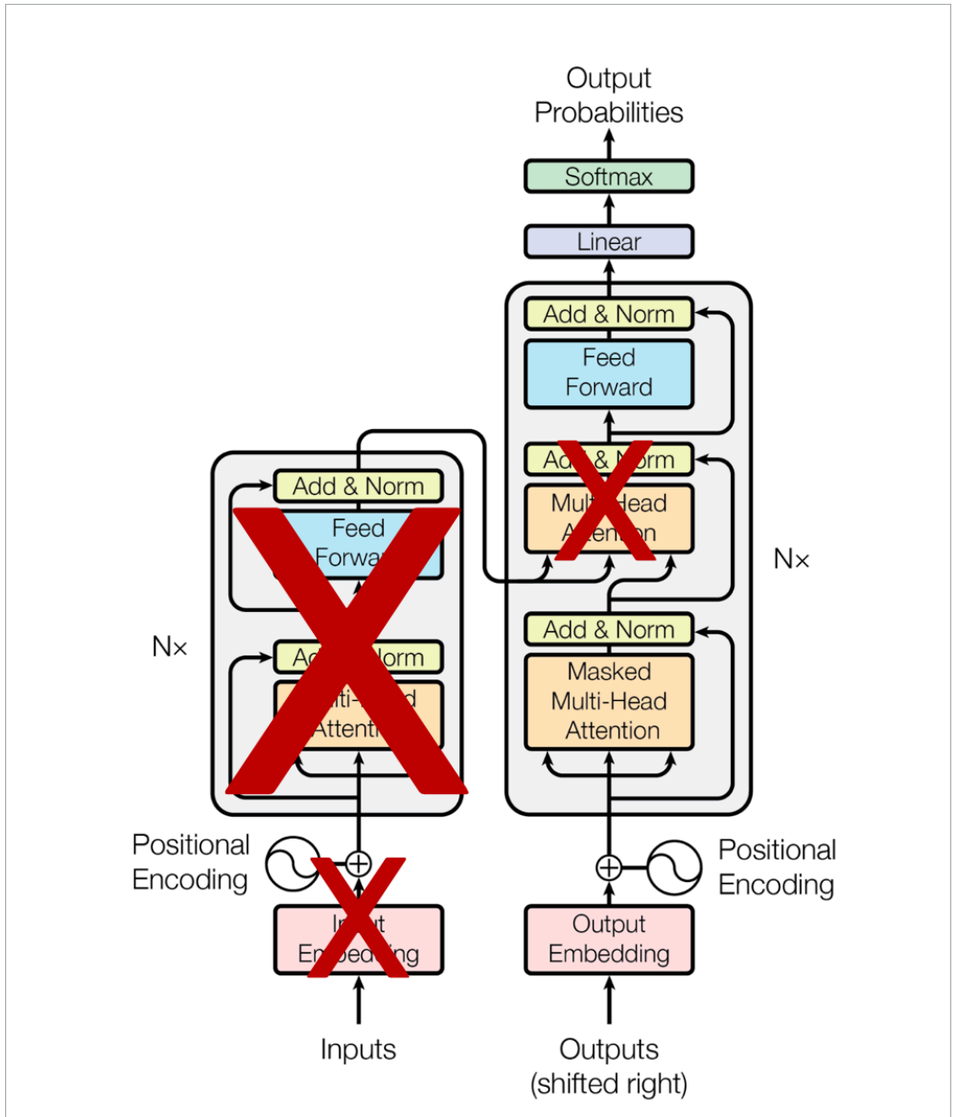
GPT 모델 구조의 이해

[그림 4-11] GPT 모델을 사용한 미세조정 아키텍처



GPT는 언어 모델입니다. 이전 단어들이 주어졌을 때 다음 단어가 무엇인지 맞히는 과정에서 사전훈련 합니다. 문장 왼쪽부터 오른쪽으로 순차적으로 계산한다는 점에서 일방향(unidirectional)입니다.

[그림 4-12] GPT 구조



GPT는 문장 생성에 강점을 갖게 됩니다. GPT는 그림 4-12처럼 트랜스포머 아키텍처에서 인코더를 제외하고 디코더만 취해 사용하는 구조입니다.

그림 4-12에서 오른쪽 디코더 블록을 자세히 보면 인코더 쪽에서 보내오는 정보를 받는 모듈(멀티 헤드 어텐션) 역시 제거돼 있음을 확인할 수 있습니다.

입력 단어 시퀀스가 “어제 카페 갔었어 거기 사람 많더라”이고 이번이 “카페”를 맞혀야 하는 상황이라고 가정해 보겠습니다. 이때 GPT는 정답 단어 “카페”를 맞힐 때 “어제”라는 단어만 참고할 수 있습니다.

따라서 정답 단어 이후의 모든 단어(“카페~많더라”)를 볼 수 없도록 처리해 줍니다. 구체적으로는 밸류 벡터들을 가중합할 때 참고할 수 없는 단어에 곱하는 점수가 0이 되도록 합니다.

“어제”라는 단어에 대응하는 GPT 마지막 레이어의 출력 결과에 선형 변환과 소프트맥스를 적용해 요솟값 각각이 확률이고 학습 대상 언어의 어휘 수만큼 차원 수를 가진 벡터가 되도록 합니다. 그리고 이번 차례의 정답인 “카페”에 해당하는 확률은 높이고 나머지 단어의 확률은 낮아지도록 모델 전체를 업데이트합니다.

이번에는 “갔었어”를 맞혀야 하는 상황입니다. 이때 GPT는 정답 단어 “갔었어”를 맞힐때 “어제”와 “카페”라는 단어를 참고할 수 있습니다.

“카페”라는 단어에 대응하는 GPT 마지막 레이어의 출력 결과에 선형 변환과 소프트맥스를 적용해 요솟값 각각이 확률이고 학습 대상 언어의 어휘 수만큼 차원 수를 가진 벡터가 되도록 합니다. 그리고 이번 차례의 정답인 “갔었어”에 해당하는 확률은 높이고 나머지 단어의 확률은 낮아지도록 모델 전체를 업데이트합니다.

“거기”를 맞혀야 하는 상황이라면 모델은 “어제, 카페, 갔었어” 세 단어를 참고할 수 있습니다.

“갔었어”라는 단어에 대응하는 GPT 마지막 레이어의 출력 결과에 선형 변환과 소프트맥스를 적용해 요솟값 각각이 확률이고 학습 대상 언어의 어휘 수만큼 차원 수를 가진 벡터가 되도록 합니다. 그리고 이번 차례의 정답인 “거기”에 해당하는 확률은 높이고 나머지 단어의 확률은 낮아지도록 모델 전체를 업데이트합니다.

위의 설명과 같이 GPT는 문장의 다음 단어를 예측하는 모델입니다. GPT와 같은 방법의 모델은 학습 데이터가 훨씬 더 방대해지고 모델의 크기가 매우 커지는 방향으로 발전하게 됩니다. 이를 통해, 대규모 언어 모델(Large Language Model, LLM)이 발전하게 되었습니다. 최근에는 GPT-3.5 또는 GPT-4 기반의 ChatGPT 등의 서비스가 출시되어 생성형 인공지능과 대화형 인공지능 유행이 불어오게 되었습니다. 오픈소스 기반의 LLM을 미세조정 하는 방법은 6장에서 다룰 예정입니다.