NLP 2주차 스터디

작성일시	@2024년 1월 25일 오전 9:46
자료	https://colab.research.google.com/drive/1HByAEYJ7RvDLt-twztI055bfoaNSdQoE?usp=drive_link
복습	☐
텍스트	트랜스포머를 활용한 자연어처리 2.3장 ~ 3.5장 (2.3, 2.4는 NLP_1에 있음)

3장 트랜스포머 파헤치기 📖

트랜스포머 모델의 주요 구성 요소와 파이토치로 구현하는 방법 알아보기

▫ 트랜스포머 아키텍처

기계 번역에 널리 사용되는 **인코더-디코더** 구조 기반

1. 인코더

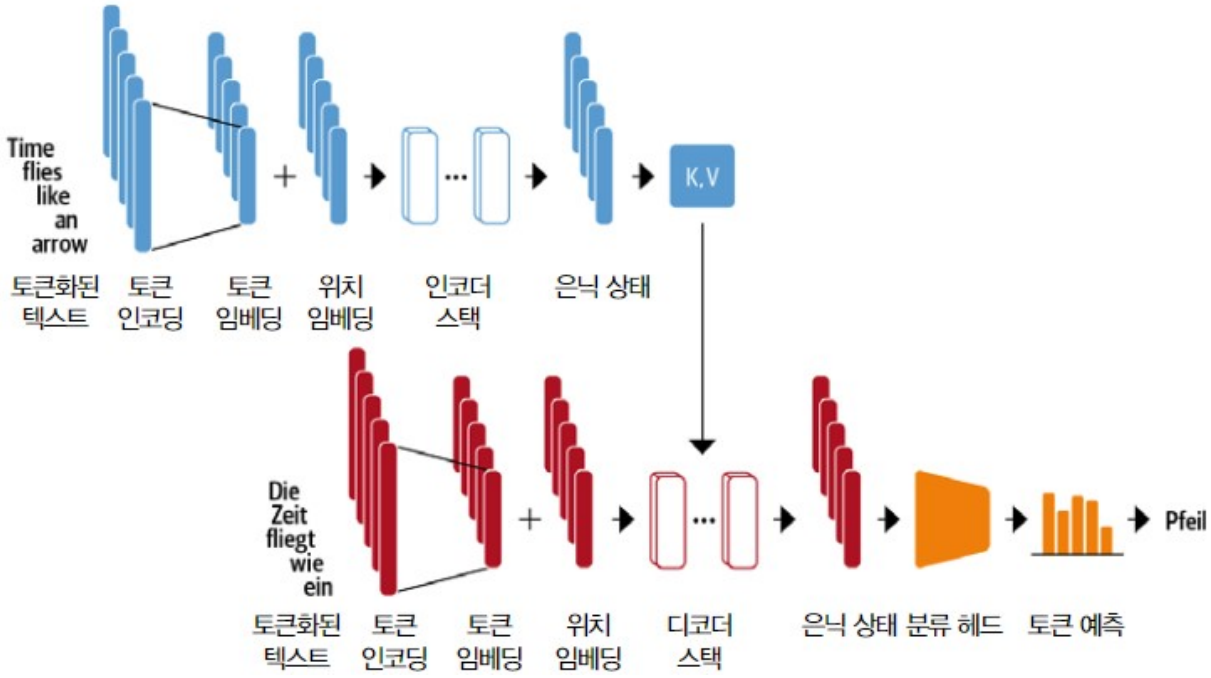
입력 토큰의 시퀀스를 hidden state 또는 context이라 부르는 임베딩 벡터의 시퀀스로 변환

+) 임베딩 : 사람이 쓰는 자연어를 기계가 이해할 수 있도록 벡터로 바꾸는 과정

2. 디코더

인코더의 은닉 상태를 사용해 출력 토큰의 시퀀스를 한 번에 하나씩 반복적으로 생성

그림의 상단이 인코더, 하단이 디코더. 둘 다 여러 개의 구성 요소로 이루어진다



▫ 트랜스포머 아키텍처의 특징

- 우선 입력 텍스트를 토큰화하고 토큰 임베딩으로 변환한다

  어텐션 메커니즘은 토큰의 상대적인 위치를 알지 못하기 때문에 텍스트의 순서 특징을

  모델링하기 위해 각 토큰의 위치 정보가 담긴 위치 임베딩을 토큰 임베딩과 합친다

- 인코더가 인코더의 층의 스택으로 구성되는데, 이는 컴퓨터 비전의 conv 층의 스택과 유사하다

  디코더도 마찬가지로 디코더 층의 스택으로 구성된다

- 디코더 층마다 인코더의 출력이 주입. 디코더는 시퀀스에서 가장 가능성 있는 다음 토큰을 예측.

  이 단계의 출력이 디코더로 다시 주입되어 다음 토큰을 생성한다

  이 과정이 EOS(end of sequence) 토큰을 예측하거나 최대 길이에 도달할 때까지 반복된다

▫ 트랜스포머 아키텍처의 유형

1. 인코더 유형

- 텍스트 시퀀스 입력을 수치 표현으로 변환
- 이런 수치 표현은 텍스트 분류나 개체명 인식(NER) 같은 작업에 적합
- BERT, RoBERTa, DistilBERT 같은 BERT 변종이 이 유형에 속함
- 이전 토큰(왼쪽)과 이후 토큰(오른쪽) 문맥에 따라 달라지기 때문에 **양방향 어텐션**이라고도 함

2. 디코더 유형

- 시작 텍스트가 주어지면 가장 가능성 있는 다음 단어를 반복해 예측하는 식
- GPT 계열이 이 유형에 속함함
- 오직 이전 토큰(왼쪽) 문맥에 따라 달라진다 >> 순차적
- 코잘 어텐션 또는 **자기회귀 어텐션(autogressive attention)**이라고도 함

3. 인코더 - 디코더 유형

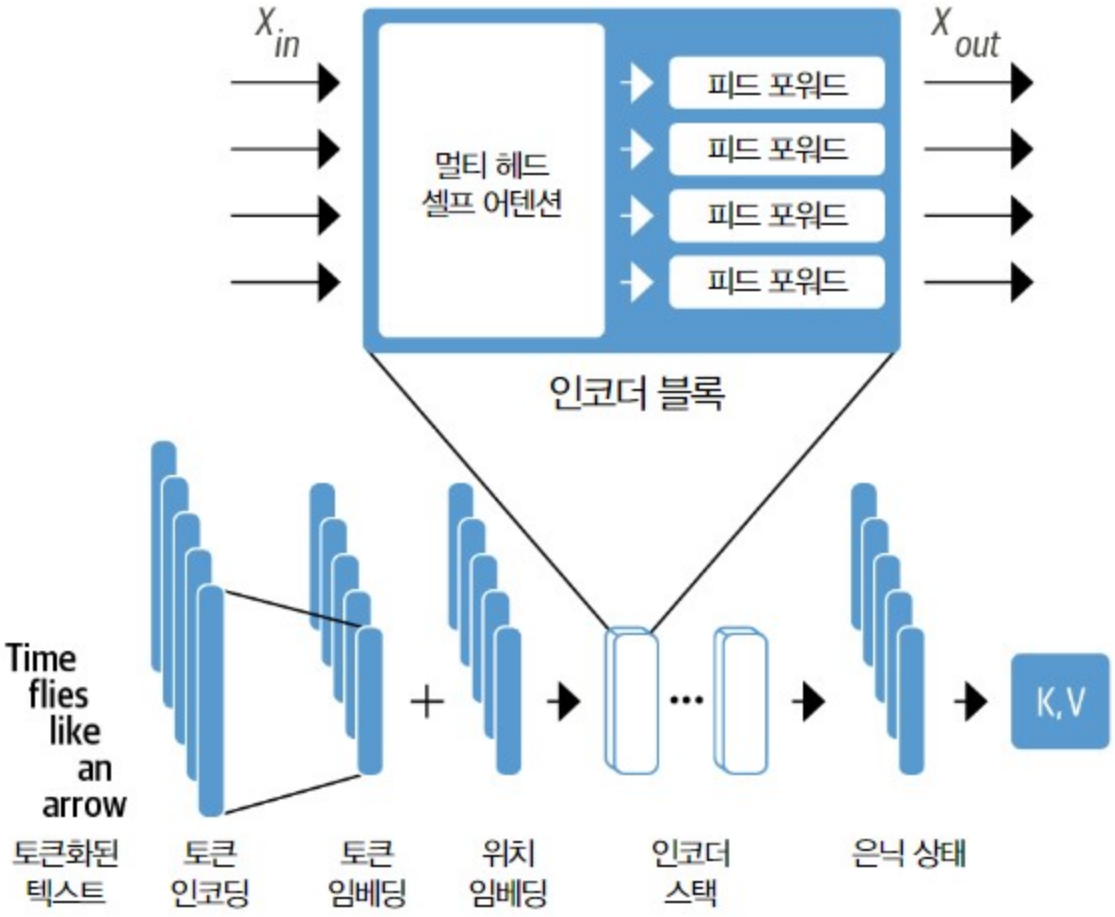
- 한 텍스트의 시퀀스를 다른 시퀀스로 매핑하는 복잡한 모델링에 사용
- 기계 번역과 요약 작업에 적합
- 인코더와 디코더를 연결한 트랜스포머 아키텍처, BART, T5 모델이 이 유형에 속함

❗ **주의**

> NOTE_ 실제로는 디코더와 인코더 유형의 구조에 따른 애플리케이션 기준이 조금 모호합니다. 예를 들어, GPT 계열 같은 디코더 유형의 모델을 전형적인 시퀀스-투-시퀀스 작업으로 여겨지는 번역 같은 작업에 활용하기도 합니다. 또 BERT 같은 인코더 유형의 모델을 일반적으로 인코더-디코더나 디코더 유형의 모델에 관련된 요약 작업에 적용하는 경우도 있습니다.[1]

3.2 인코더

인코더의 내부 동작 알아보기



- 트랜스포머의 인코더는 여러 개의 인코더 층이 서로 쌓여 구성된다

- 그림과 같이 각 인코더 층은 임베딩 시퀀스를 받아 다음과 같은 층에 통과시킨다
  - 멀티 헤드 셀프 어텐션 층
  - 각각의 입력 임베딩에 적용되는 fully connected feed-forward 층

  ∗ 이 둘은 DNN을 효율적으로 훈련하기 위해 **skip connection**과 **층 정규화**를 사용하기도 한다

- 인코더 층의 출력 임베딩은 입력과 크기가 동일하다

인코더 스택의 주요 역할

입력 임베딩을 업데이트해 시퀀스의 문맥 정보가 인코딩된 표현을 만드는 것

▼ 예시

‘apple’이란 단어에 ‘keynote’나 ‘phone’이 가까이 있다면,

‘apple’에 대한 표현을 회사와는 가깝고 과일과는 멀게 업데이트하는 것!

▫ 셀프 어텐션

: 트랜스포머 아키텍처의 핵심

- 어텐션 메커니즘에서 신경망은 시퀀스의 각 원소에 서로 다른 가중치 또는 ‘어텐션’을 할당

  +) 텍스트 시퀀스에서 원소는 토큰 임베딩임

- 각 토큰은 고정 차원의 벡터에 매핑됨

- 셀프 어텐션의 ‘셀프’는

  이 가중치가 동일 집합에 있는 **모든** 은닉 상태에 대해 계산된다는 것을 의미

- 순환 모델과 연관된 어텐션 메커니즘은 특정 디코딩 타임스텝에서

  해당 디코더의 은닉 상태와 인코더의 각 은닉 상태가 가진 관련성을 계산함

- 각 토큰에 대해 고정된 임베딩을 사용하는 대신

  전체 시퀀스를 사용해 각 임베딩의 가중 **평균**을 계산하는 것이 셀프 어텐션의 기본 개념

▫ 셀프 어텐션 공식

$$x_i' = \sum_{j=1}^{n} w_{ji} x_j$$

토큰 임베딩의 시퀀스 xj에 어텐션 가중치 wji를 계수로 두어 새로운 임베딩 시퀀스 x'i를 만드는 것

- 여기서 x'i는 xj의 선형 결합이며 위 공식은 가중치의 합이 1이 되도록 정규화한다

토큰 임베딩의 평균을 구하는 이유?

- 예시로 보는 **문맥 고려 임베딩**

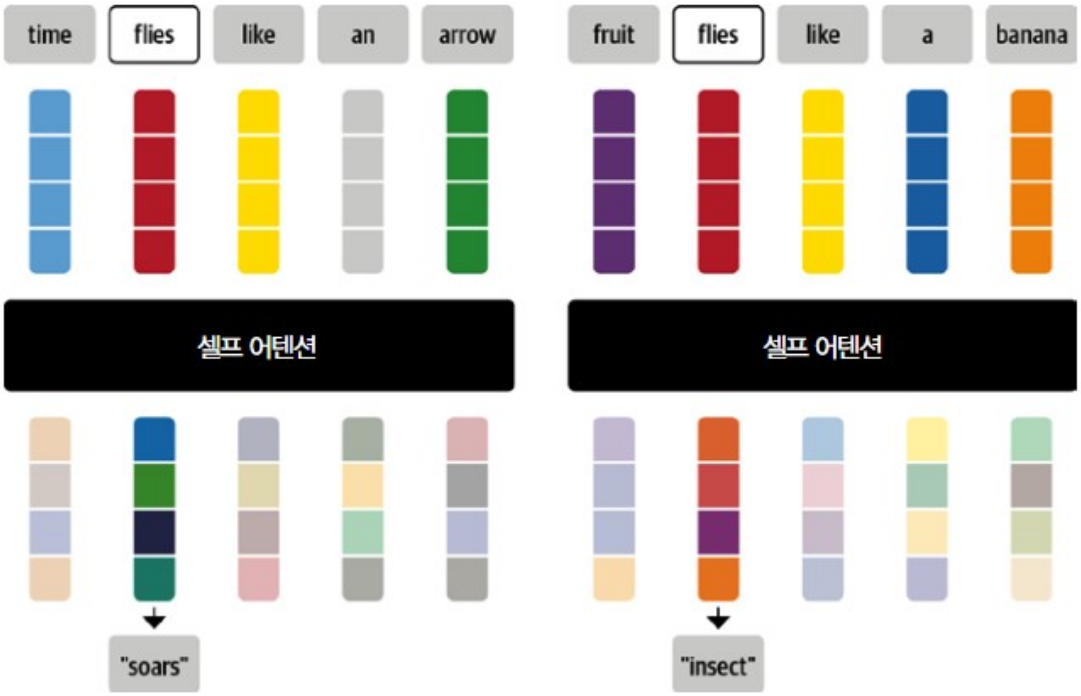


그림 3-3 셀프 어텐션이 원시 토큰 임베딩(위)을 문맥 고려 임베딩(아래)으로 업데이트해서 전체 시퀀스 정보를 통합하는 표현을 만드는 방법

'files'는 파리를 뜻하는 명사와 날아간다는 뜻의 동사 두 가지를 떠올릴 수 있음

모든 토큰 임베딩을 비율을 달리해 통합하면 문맥을 내포하는 'flies' 라는 표현이 만들어짐

예시로 'time flies like an arrow' 같은 문맥에서는 'time'과 'arrow' 토큰 임베딩에 더 큰 가중치를 할당

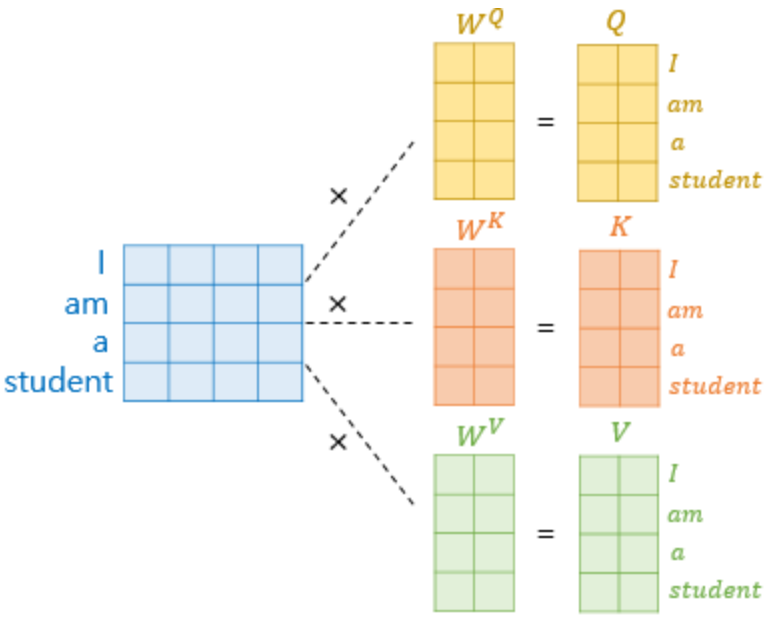
이런 식으로 생성된 임베딩이 문맥 고려 임베딩이다

▫ 스케일드 점곱 어텐션

셀프 어텐션 층을 구현하는 가장 일반적인 방법

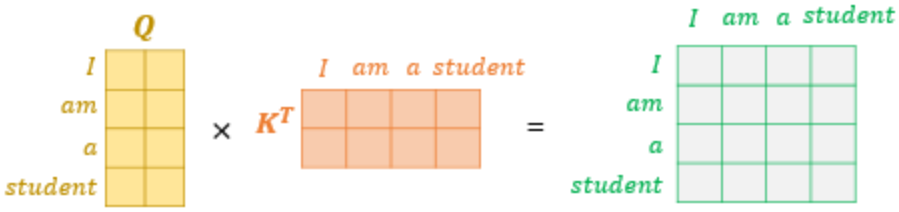
▫ 스케일드 점곱 어텐션 구현

1단계 : 각 토큰 임베딩을 쿼리(Q), 키(K), 값(Value) 세 개의 벡터로 투영



문장 행렬에 가중치 행렬을 곱해 Q, K, V 행렬 구하기

2단계 : 어텐션 점수 계산



유사도 함수를 사용해 쿼리 벡터와 키 벡터가 서로 얼마나 관련되는지 계산한다

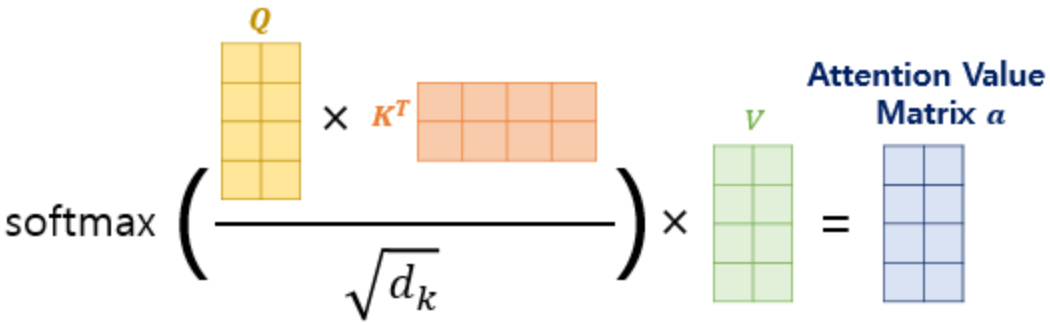
여기서 유사도 함수는 점곱이며 임베딩의 행렬 곱셈을 사용해 효율적으로 계산한다

쿼리와 키가 비슷하면 점곱 결과가 크고, 쿼리와 키에 공통 부분이 많지 않으면

겹치는 부분이 적거나 거의 없으므로 점곱 결과가 작다. 이 단계의 출력을 어텐션 점수라 하며,

n개의 입력 토큰이 있는 시퀀스의 경우 크기가 n x n인 어텐션 점수 행렬이 만들어진다

3단계 : 어텐션 가중치 계산



일반적으로 내적은 임의의 큰 수를 만들기 때문에 훈련 과정이 불안정.

이를 처리하기 위해 어텐션 점수에 **스케일링 인자?**를 곱해 분산을 정규화하고

소프트맥스 함수를 적용해 모든 열의 합이 1이 되게 한다.

이를 통해 만들어진 n x n 행렬에는 어텐션 가중치 w가 담긴다.

4단계 : 토큰 임베딩 업데이트

어텐션 가중치가 계산되면 이를 값 벡터 v1, … , vn과 곱해 임베딩을 위해 업데이트 된 표현을 얻는다.

▫ 어텐션 가중치 시각화

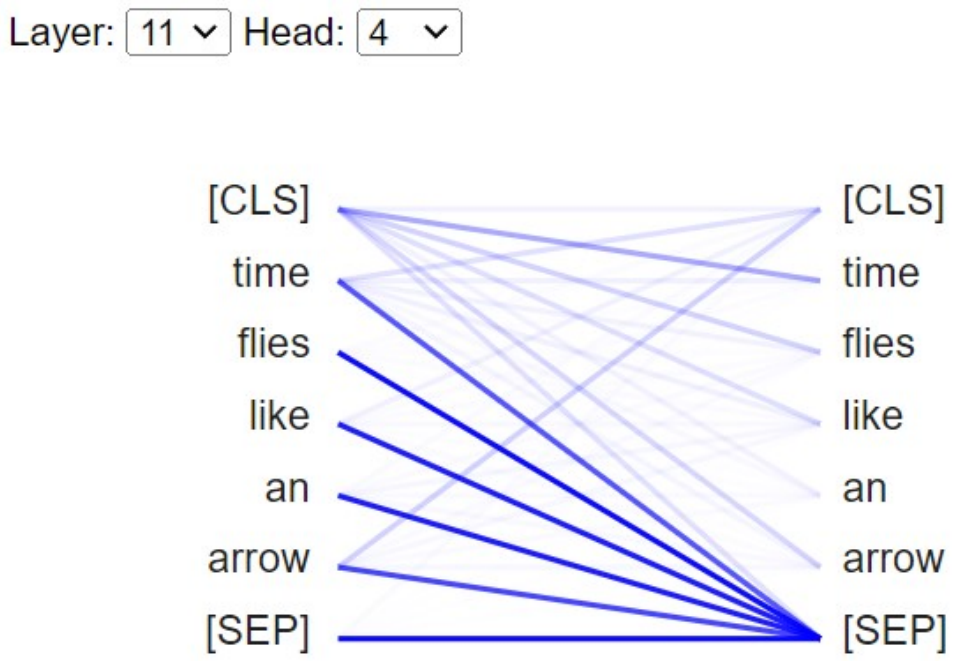
neuron_view 모듈 사용. 이 모듈은 쿼리와 키 벡터가 어떻게 결합되어

최종 가중치를 생산하는지를 가중치 계산 과정을 추적해 알려준다

▼ 시각화 코드

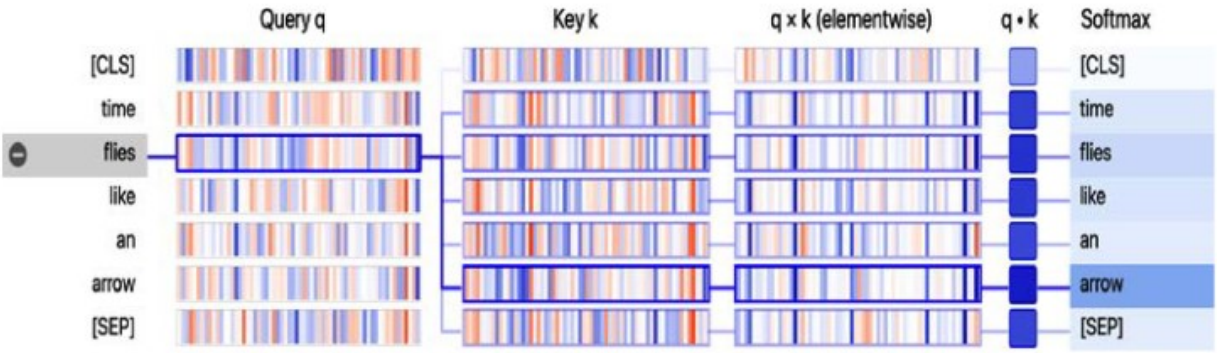
```
# neuron_view 모듈 사용
from transformers import AutoTokenizer
from bertviz.transformers_neuron_view import BertModel
from bertviz.neuron_view import show

#Bertviz의 모델 클래스 객체 초기화
model_ckpt = "bert-base-uncased"
tokenizer = AutoTokenizer.from_pretrained(model_ckpt)
model = BertModel.from_pretrained(model_ckpt)
text = "time flies like an arrow"

# 특정 인코더 층과 어텐션 헤드에 대한 시각화 생성
show(model, "bert", tokenizer, text, display_mode="light", la
```

BertViz가 모델의 어텐션 층을 활용하므로 체크포인트로 BertViz의 모델 클래스 객체 초기화



코드 출력 결과, layer와 head를 조절하며 결과 확인 가능

- 코드 실행 결과, 연결선은 토큰의 어텐션에 따라 가중된다



그림에서 수직 밴드는 쿼리와 키 벡터의 값을, 밴드의 색 농도는 값의 크기를 나타낸다

쿼리, 키, 값 이해하기

필요한 식재료를 각각 **쿼리**로 생각하면

마트 진열대에 붙은 **이름표(키)**를 훑으며 필요한 재료와 **일치(유사도 함수)**하는지 확인하기

이름표가 일치하면 진열대에서 **상품(값)**을 꺼낸다는 비유로 이해할 수 있다

셀프어텐션은 이 비유와 비슷하지만 더 추상적이고 유연

키와 쿼리의 일치 정도에 따라 마트의 **모든** 이름표가 재료에 일치한다

따라서 달걀 12개를 사려 했지만, 달걀 10개와 오믈렛 1개, 윙 1개가 선택될 때도 있다

🤗 트랜스포머 아키텍처 구현

▫ 토큰 임베딩

토큰 임베딩은 문맥과 독립적이다. ‘flies’ 같은 동음이의어의 표현이 동일하기 때문이다.

1. 텍스트 토큰화

토크나이저를 이용해 input_ids 추출하기

▼ 코드

```
inputs = tokenizer(text, return_tensors="pt", add_special_tol
inputs.input_ids
```

```
tensor([[ 2051, 10029,  2066,  2019,  8612]])
```

- 시퀀스에 있는 각 토큰은 토크나이저 어휘사전에서 고유한 각 ID에 매핑됨
- add_special_tokens=False 로 두어 [CLS], [SEP] 토큰 제외

2. 밀집 임베딩 만들기

밀집(dense)은 임베딩에 있는 모든 원소의 값이 0이 아니라는 의미임

+) 추가로 원-핫 인코딩은 sparse함 > 한 원소를 제외한 모든 원소가 0이기 때문

▼ 룩업테이블(?) 코드

```
from torch import nn
from transformers import AutoConfig
```

```
# AutoConfig 클래스를 사용해 bert-base-uncased 체크포인트에 관한 co
config = AutoConfig.from_pretrained(model_ckpt)
token_emb = nn.Embedding(config.vocab_size, config.hidden_siz
token_emb
```

파이토치의 torch.nn.Embedding 층은 각 입력 ID에 대해 룩업테이블처럼 동작

+) 룩업테이블 : 자주 사용하는 데이터 값들을 정리해 놓은 표

```
Embedding(30522, 768)
```

- 허깅페이스 트랜스포머스에서 모든 ckpt는 vocab_size, hidden_size 등과 같은

  다양한 하이퍼파라미터가 지정된 설정 파일이 할당됨
- 이 경우 입력 ID가 nn.Embedding에 저장된 30,522개 임베딩 벡터 중 하나에 매핑되고,

  각 벡터의 크기는 768이다
- AutoConfig 클래스는 모델 예측 포맷을 지정하는데 사용되는 추가적인 메타데이터도 저장

+) 메타 데이터 : 데이터에 대한 데이터. 메타는 한 단계 더 위에 있는 것을 가리키는 말

▼ 입력 ID 전달해 임베딩 만들기

```
inputs_embeds = token_emb(inputs.input_ids)
inputs_embeds.size()
```

```
torch.Size([1, 5, 768])
```

이 층은 [batch_size, seq_len, hidden_dim] 크기의 텐서를 출력

▫ 어텐션 층

문맥과 독립적인 토큰 임베딩을 혼합해 의미를 명확하게 하고 토큰 표현에 문맥 내용을 주입

1. 어텐션 점수 계산

Q, K, V 벡터를 만들고 점곱을 유사도 함수로 사용해 어텐션 점수 계산하기

▼ 어텐션 점수 계산 크기 코드

```
import torch
from math import sqrt

query = key = value = inputs_embeds
dim_k = key.size(-1)
# torch.bmm()은 배치 행렬 - 행렬 곱셈 수행
scores = torch.bmm(query, key.transpose(1,2)) / sqrt(dim_k)
scores.size()
```

torch.bmm 함수는 행렬 곱셈을 수행해서 크기가 [batch_size, seq_len, hidden_dim]인 쿼리와

키 벡터의 어텐션 점수 계산을 단순화한다.

```
torch.Size([1, 5, 5])
```

- 배치에 있는 샘플마다 5 x 5 크기의 어텐션 점수 행렬이 만들어진다

- 추후 임베딩에 독립적인 가중치 행렬 W를 각각 적용해 쿼리, 키 값 벡터 생성할 것
- 여기서는 간단하게 모두의 값을 같게 하였음

스케일드 점곱 어텐션에서 점곱은 임베딩 벡터의 크기로 스케일을 조정한다

이는 훈련 도중 큰 수의 빈번한 발생을 줄여 소프트 맥스 함수의 포화를 방지한다

2. 소프트맥스 함수 적용하기

▼ 소프트맥스 함수 코드

```
import torch.nn.functional as F

weights = F.softmax(scores, dim=-1)
weights.sum(dim=-1)
```

```
tensor([[1., 1., 1., 1., 1.]], grad_fn=<SumBackward1>)
```

3. 어텐션 가중치 곱하기

▼ 어텐션 가중치 코드

```
attn_outputs = torch.bmm(weights, value)
attn_outputs.shape
```

```
torch.Size([1, 5, 768])
```

전 과정이 2개의 행렬 곱셈과 소프트맥스 함수이므로

셀프 어텐션을 일종의 평균 계산으로 생각해도 괜찮음!

▼ 추후 사용을 위해 두 단계를 하나의 함수로 만들기

```
def scaled_dot_product_attention(query, key, value):
  dim_k = query.size(-1)
  scores = torch.bmm(query, key.transpose(1, 2)) / sqrt(dim_k
  weights = F.softmax(scores, dim=-1)
  return torch.bmm(weights, value)
```

동일한 쿼리와 키 벡터를 사용하는 어텐션 메커니즘은 문맥에서 동일한 단어에 큰 점수를 할당

하지만 실전에서 단어의 의미를 파악하는데 도움되는 것은 동일한 단어보단 문맥을 보완하는 단어

이걸 가능하게 해주는 것이 멀티 헤드 어텐션

▫ 멀티 헤드 어텐션

세 개의 선형 투영을 사용해 초기 토큰 벡터를

세 개의 공간에 투영하는 식으로 쿼리, 키, 값 벡터 다르게 만들기

1. 단일 어텐션 헤드

어텐션 헤드 : 각 선형 투영 집합

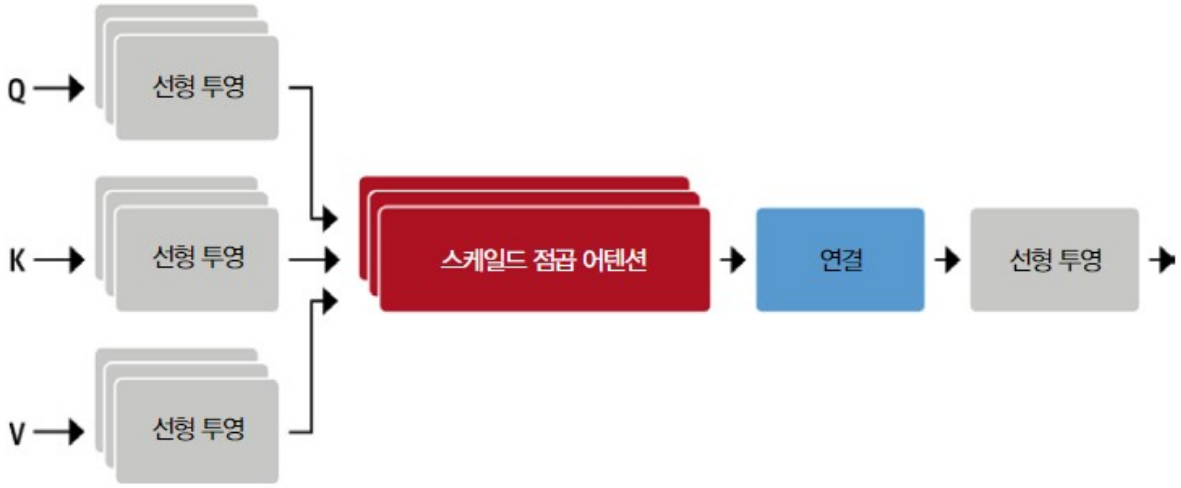


그림 3-5 멀티 헤드 어텐션

Q. 어텐션 헤드가 하나 이상 필요한 이유?

A. 한 헤드의 소프트맥스가 유사도의 한 측면에만 초점을 맞추는 경향이 있기 때문

여러 개의 헤드가 있으면 모델은 동시에 여러 측면에 초점을 맞춘다

▼ 예시

한 헤드는 주어-동사 상호작용에 초점을 맞추고, 다른 헤드는 인접한 형용사를 찾는 식

이런 관계는 모델에 수동으로 입력되지 않고 모델이 직접 데이터에서 학습한다

이는 CNN의 필터와 유사하다

▼ 단일 어텐션 헤드를 위한 코드

```
# 단일 어텐션 헤드
class AttentionHead(nn.Module):
  def __init__(self, embed_dim, head_dim):
    super().__init__()

    # 세 개의 독립된 선형층 생성
    self.q = nn.Linear(embed_dim, head_dim)
    self.k = nn.Linear(embed_dim, head_dim)
    self.v = nn.Linear(embed_dim, head_dim)

  def forward(self, hidden_state):
    attn_outputs = scaled_dot_product_attention(
        self.q(hidden_state), self.k(hidden_state), self.v(hi
    return attn_outputs
```

선형층은 임베딩 벡터에 행렬 곱셈을 적용해 [batch_size, seq_len, head_dim] 크기의 텐서를 만듦

- head_dim : 투영하려는 차원의 크기

  head_dim이 토큰의 임베딩 차원(embed_dim)보다 더 작을 필요는 없지만,
  실전에서는 헤드마다 계산이 일정하도록 embed_dim과 배수가 되게 선택한다

▼ 예시

BERT에는 어텐션 헤드가 12개 있으므로 각 헤드의 차원은 768 / 12 = 64이다

2. 멀티 헤드 어텐션 층 만들기

각 어텐션 헤드의 출력을 연결해서 완전한 멀티 헤드 어텐션 층 만들기

▼ 멀티 헤드 어텐션 층 코드

```
class MultiHeadAttention(nn.Module):
def __init__(self, config):
        super().__init__()
        embed_dim= config.hidden_size
        num_heads= config.num_attention_heads
        head_dim= embed_dim// num_heads
        self.heads= nn.ModuleList(
            [AttentionHead(embed_dim, head_dim)for _in ran
ge(num_heads)]
        )
        self.output_linear= nn.Linear(embed_dim, embed_di
m)

def forward(self, hidden_state):
        x= torch.cat([h(hidden_state)for hin self.heads],
dim=-1)
        x= self.output_linear(x)
return x
```

어텐션 헤드의 출력을 연결한 다음 최종 선형 층으로 주입해서

[batch_size, seq_len, hidden_dim] 크기의 출력 텐서 만들기

3. 멀티 헤드 어텐션 층이 기대하는 입력 크기를 만드는지 확인하기

MultiHeadAttention 모듈을 초기화할 때 앞서 사전 훈련된 BERT 모델에서 로드한 설정 전달

>> BERT와 동일한 설정을 사용

▼ 코드

```
multihead_attn = MultiHeadAttention(config)
attn_output = multihead_attn(inputs_embeds)
```

```
attn_output.size()
```

```
torch.Size([1, 5, 768])
```

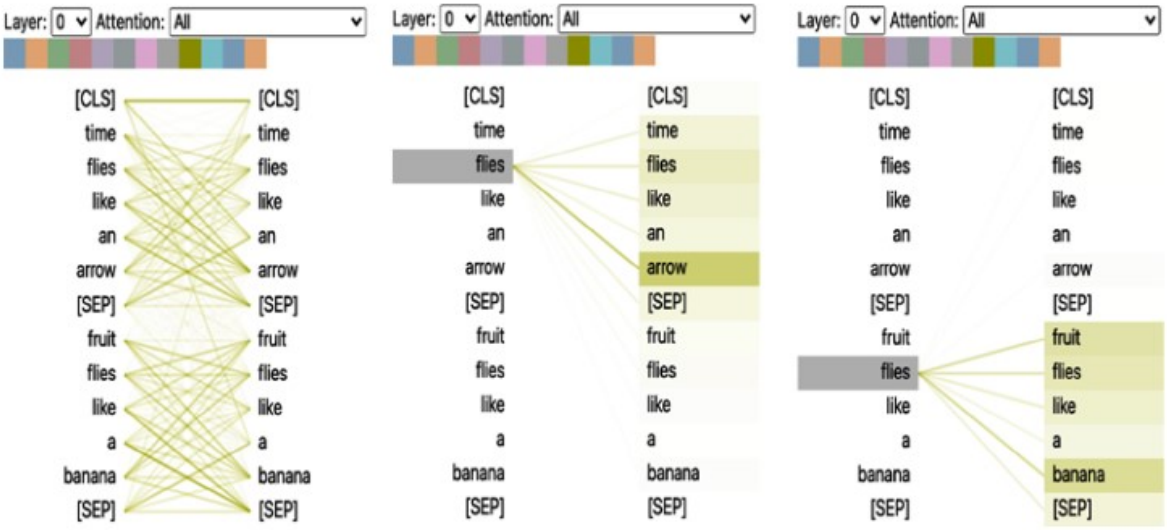
결과 굿굿

BertViz 사용해 다른 용도로 사용된 ‘flies’ 단어의 어텐션 시각화하기

head_view() 함수를 사용하기 위해 사전 훈련된 ckpt의 어텐션을 계산하고 문장 경계 위치 지정하기

▼ 코드

```
from bertviz import head_view
from transformers import AutoModel

model = AutoModel.from_pretrained(model_ckpt, output_attentio

sentence_a = "time flies like an arrow"
sentence_b = "fruit flies like a banana"

viz_inputs = tokenizer(sentence_a, sentence_b, return_tensors
attention = model(**viz_inputs).attentions
sentence_b_start = (viz_inputs.token_type_ids == 0).sum(dim=1
tokens = tokenizer.convert_ids_to_tokens(viz_inputs.input_ids

head_view(attention, tokens, sentence_b_start, heads=[8])
```



- 어텐션 가중치는 임베딩이 업데이트된 토큰 (왼쪽)과
  주의를 기울인 단어 (오른쪽)을 연결한 직선으로 표시 중.
- 직선의 진하기는 어텐션 가중치의 강도를 나타내고,
  짙은 색은 1에 가까운 값, 흐린색은 0에 가까운 값을 나타냄

  시각화 결과, 어텐션 가중치는 동일한 문장에 속한 단어 간에 가장 강하게 나타남
  즉, BERT가 동일한 문장에 있는 단어에 주의를 기울이라고 알려주는 중인 것

▫ 피드 포워드 층

간단한 두 개의 층으로 구성된 완전 연결 신경망.

하지만 전체 임베딩 시퀀스를 하나의 벡터로 처리하지 않고 각 임베딩을 독립적으로 처리한다

>> 이런 이유로 위치별 피드 포워드 층이라고도 불림

: CV에서는 커널 크기가 1인 1차원 합성곱이라고도 부른다

일반적으로 첫 번째 층의 크기를 임베딩의 4배로 하고, GELU 활성화 함수를 가장 널리 사용.

이 층은 대부분의 용량과 기억이 일어나는 곳으로 간주되며,

모델을 확장할 때 가장 많이 늘리는 부분.

▼ 피드 포워드 층 구현

```
class FeedForward(nn.Module):
  def __init__(self, config):  # config : 학습에 필요한 파라미터들
    super().__init__()
    self.linear_1 = nn.Linear(config.hidden_size, config.inte
    self.linear_2 = nn.Linear(config.intermediate_size, confi
    self.gelu = nn.GELU()
    self.dropout = nn.Dropout(config.hidden_dropout_prob)

  def forward(self, x):
    x = self.linear_1(x)
    x = self.gelu(x)
    x = self.linear_2(x)
    x = self.dropout(x)
    return x
```

- nn.Linear 같은 피드 포워드 층은 일반적으로 (batch_size, input_dim) 크기의 텐서에 적용되며

  배치 차원의 각 원소에 독립적으로 동작한다. 마지막 차원을 제외한 모든 차원에 독립적으로

  적용. 따라서 (batch_size, seq_len, hidden_dim) 크기의 텐서를 전달하면 배치와 시퀀스의

  모든 토큰 임베딩에 이 피드 포워드 층이 독립적으로 적용된다.

- nn.Dropout 클래스의 dropout은 신경망의 일반화 성능을 높이기 위해 자주 쓰이는 테크닉

  신경망 구조 학습 시, 레이어 간 연결 중 일부를 랜덤하게 삭제하면, 여러 개의 네트워크를

  앙상블 하는 효과를 낼 수 있고, 이로 인해 일반화 성능이 높아진다고 한다.

▼ 어텐션 출력을 전달해 테스트하기

```
feed_forward = FeedForward(config)
ff_outputs = feed_forward(attn_outputs)
ff_outputs.size()
```

```
torch.Size([1, 5, 768])
```

스킵 연결과 층 정규화를 배치할 위치 정하기 > 이 위치가 모델 구조에 어떤 영향을 미치는지 알아보자

▫ 층 정규화 추가하기

트랜스포머 아키텍쳐는 층 정규화와 스킵 연결을 사용한다

- 층 정규화 (layer normalization)

배치에 있는 각 입력을 평균이 0이고 단위 분산(분산이 1)을 가지도록 정규화한 것

- 스킵 연결 (skip connection)

처리하지 않은 텐서를 모델의 다음 층으로 전달한 후 처리된 텐서와 더하는 것

1. 사후 층 정규화

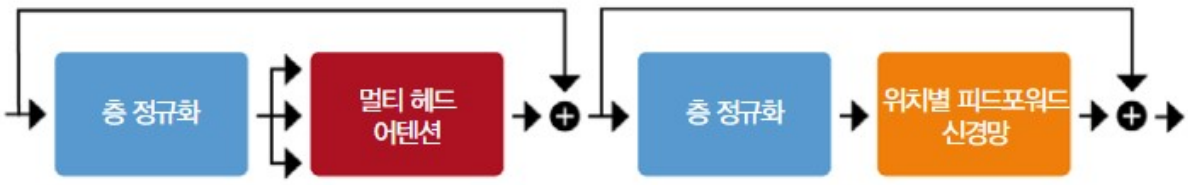
사후 층 정규화

트랜스포머 논문에서 사용한 방법. 스킵 연결 사이에 층 정규화를 놓는다

이 방식은 그래디언트가 발산하는 경우가 생겨 처음부터 훈련하기가 까다롭다

이런 이유로 훈련하는 동안 학습률을 최댓값까지 점진적으로 증가시키는 **학습률 웜업** 개념 사용

2. 사전 층 정규화



사전 층 정규화

다른 논문에서 많이 사용하는 방법. 스킵 연결 안에 층 정규화를 놓는다

훨씬 안정적으로 훈련되는 경향이 있으며 보통 **학습률 웜업**이 필요하지 않다

▼ 사전 층 정규화 방식의 코드

```
class TransformerEncoderLayer(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.layer_norm_1 = nn.LayerNorm(config.hidden_size)
        self.layer_norm_2 = nn.LayerNorm(config.hidden_size)
        self.attention = MultiHeadAttention(config)
        self.feed_forward = FeedForward(config)

    def forward(self, x):
        # 층 정규화를 적용하고 입력을 쿼리, 키, 값으로 복사한다
        hidden_state = self.layer_norm_1(x)
        # 어텐션에 스킵 연결을 적용한다
        x = x + self.attention(hidden_state)
        # 스킵 연결과 피드 포워드 층을 적용한다
        x = x + self.feed_forward(self.layer_norm_2(x))
        return x
```

앞서 만든 구성 요소를 간단히 연결하면 된다

▼ 입력 임베딩으로 테스트하기

```
encoder_layer = TransformerEncoderLayer(config)
inputs_embeds.shape, encoder_layer(inputs_embeds).size()
```

```
(torch.Size([1, 5, 768]), torch.Size([1, 5, 768]))
```

트랜스포머 인코더 층 처음부터 구현 완.

※ 인코더 층을 설정할 때 주의할 점

> 💡 이 층은 토큰의 위치와 상관이 없다.
>
> 멀티 헤드 어텐션 층은 따지고 보면 가중 합이기 때문에 토큰 위치에 대한 정보가 사라진다
>
> 전문 용어로는 셀프 어텐션과 피드 포워드 층이 '**순열 등변**' 하다고 한다
>
> 즉, 입력 순서가 바뀌면 층의 해당 출력도 정확히 같은 식으로 바뀐다는 것

▫ 위치 임베딩

벡터에 나열된 값의 위치 패턴으로 토큰 임베딩을 보강하는 것

이 패턴에 위치의 특징이 담겼다면, 각 스택에 있는 어텐션 헤드와 피드 포워드 층은

위치 정보와 변환을 통합하는 방법을 배운다

1. 학습 가능한 패턴을 사용하는 방법

- 이 방법은 사전 훈련 데이터셋이 충분히 큰 경우에 좋다.
- 토큰 임베딩과 정확히 같은 식으로 동작하지만, 입력으로 토큰 ID가 아닌 위치 인덱스를 사용.

  이를 통해 훈련하는 동안 토큰 위치를 인코딩하는 효과적인 방법을 학습.

▼ 사용자 정의 Embeddings 모듈 만들기

```
class Embeddings(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.token_embeddings = nn.Embedding(config.vocab_siz
                                             config.hidden_si
        self.position_embeddings = nn.Embedding(config.max_po
                                                config.hidden
        self.layer_norm = nn.LayerNorm(config.hidden_size, ep
        self.dropout = nn.Dropout()

    def forward(self, input_ids):
        # 입력 시퀀스에 대해 위치 ID를 만들기
        seq_length = input_ids.size(1)
        position_ids = torch.arange(seq_length, dtype=torch.l
        # 토큰 임베딩과 위치 임베딩 만들기
        token_embeddings = self.token_embeddings(input_ids)
        position_embeddings = self.position_embeddings(positi
        # 토큰 임베딩과 위치 임베딩 합치기
        embeddings = token_embeddings + position_embeddings
        embeddings = self.layer_norm(embeddings)
        embeddings = self.dropout(embeddings)
        return embeddings
```

```
embedding_layer = Embeddings(config)
embedding_layer(inputs.input_ids).size()
```

이 클래스는 input_ids를 밀집 은닉 상태에 투영하는 토큰 임베딩 층과 position_ids에 동일한

작업을 수행하는 위치 임베딩을 통합한다. 단순히 두 임베딩을 더해 최종 임베딩을 만든다

```
torch.Size([1, 5, 768])
```

이제 이 임베딩 층은 토큰마다 하나의 밀집 임베딩을 만든다

2. 절대 위치 표현

- 이 경우 트랜스포머 모델은 변조된 사인 및 코사인 신호로 구성된 정적 패턴을 사용해 토큰 위치를 인코딩한다.
- 이 방식은 가용한 데이터가 많지 않을 때 잘 작동한다.

3. 상대 위치 표현

- 임베딩을 계산할 때 절대적인 위치보다 주위 토큰을 중요하게 여겨 상대 위치를 인코딩.
- 상대적 임베딩은 주위를 기울이는 시퀀스의 위치에 따라 토큰에 대한 상대적 임베딩이 바뀌기 때문에, 처음부터 상대적 임베딩 층을 새로 추가하는 식으로 해결 할 수 없음.
- 대신 어텐션 메커니즘 자체에 토큰의 상대 위치를 고려하는 항을 추가한다.
- DeBERTa 같은 모델이 이런 표현을 사용함.

4. 순환 위치 임베딩

- 절대 위치 표현과 상대 위치 표현의 개념을 합친 것
- 많은 작업에서 훌륭한 결과를 달성한다.

- GPT - Neo가 순환 위치 임베딩을 사용하는 모델의 예이다.

▼ 임베딩과 인코더 층을 연결해 완전한 트랜스포머 인코더 만들기

```
class TransformerEncoder(nn.Module):
  def __init__(self, config):
    super().__init__()
    self.embeddings = Embeddings(config)
    self.layers = nn.ModuleList([TransformerEncoderLayer(con
                                 for _ in range(config.num_hi

  def forward(self, x):
    x = self.embeddings(x)
    for layer in self.layers:
      x = layer(x)
    return x
```

▼ 인코더의 출력 크기 확인하기

```
encoder = TransformerEncoder(config)
encoder(inputs.input_ids).size()
```

배치에 있는 각 토큰에 대한 은닉 상태를 얻었음.

이런 출력 포맷은 아키텍처를 유연하게 만들며, 마스크드 언어 모델링에서 누락된 토큰을

예측하는 애플리케이션이나 질문 답변에서 답변의 시작과 끝 위치를 예측하는 애플리케이션 등

다양한 경우에 쉽게 적용됨.

▫ 분류 헤드 추가하기

트랜스포머 모델은 일반적으로 작업에 독립적인 바디와 작업에 특화된 헤드로 나뉜다.

지금까지 만든 것은 **바디**로 텍스트 분류 모델을 만들고 싶다면 바디에 연결할 **분류 헤드**가 필요함.

각 토큰에 대한 은닉 상태가 있어 토큰마다 예측을 만들 수도 있지만 필요한 예측은 단 하나이므로

일반적으로는 모델의 첫 번째 토큰을 예측에 사용하고 드롭아웃 층과 선형 층을 추가해

분류 예측을 만든다.

▼ 시퀀스 분류를 위해 기존 인코더를 확장한 클래스

```
class TransformerForSequenceClassification(nn.Module):
    def __init__(self, config):
        super().__init__() # 부모 클래스의 속성 및 메소드를 가져오는
        self.encoder = TransformerEncoder(config)
        self.dropout = nn.Dropout(config.hidden_dropout_prob)
        self.classifier = nn.Linear(config.hidden_size, confi

    def forward(self, x):
        x = self.encoder(x)[:, 0, :] # [CLS] 토큰의 은닉 상태를
        x = self.dropout(x)
        x = self.classifier(x)
        return x
```

▼ 모델을 초기화하기 전에 예측하려는 클래스 개수를 정의한다

```
config.num_labels = 3
encoder_classifier = TransformerForSequenceClassification(
```

```
encoder_classifier(inputs.input_ids).size()
```

```
torch.Size([1, 3])
```

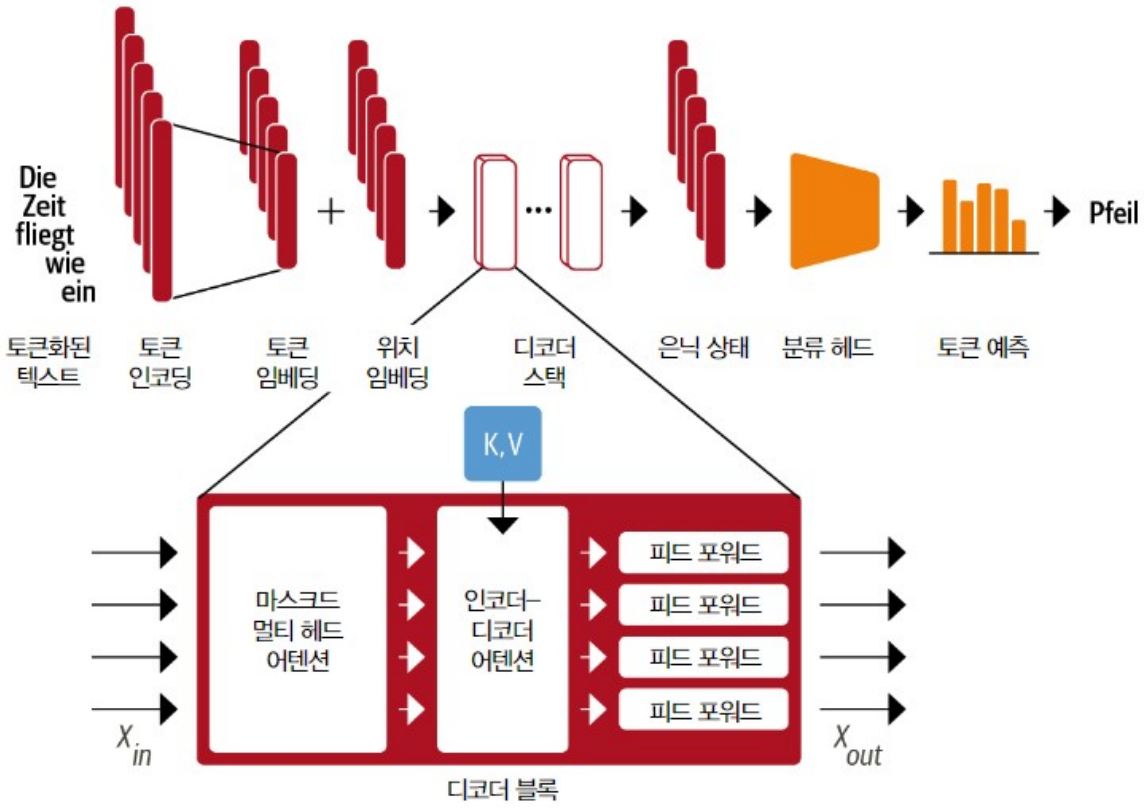
배치에 있는 각 샘플에 대해 출력 클래스마다 정규화되지 않은 로짓이 반환된다.

+) 로짓 : 로지스틱 회귀에서 승산(Odds)에 로그 값을 취한 함수

이는 2장에서 트윗의 감정을 감지하기 위해 사용한 BERT 모델과 동일.

3.3 디코더

인코더와 디코더의 주요 차이점은 디코더에는 두 개의 어텐션 층이 있다는 것



1. 마스크드 멀티 헤드 셀프 어텐션 층

- 타임스텝마다 지난 출력과 예측한 현재 토큰만 사용하여 토큰을 생성
  이렇게 하지 않으면 디코더는 훈련하는 동안 단순히 타깃 번역을 복사하는 방식의 부정행위가 가능하기 때문
- 이런 식으로 입력을 마스킹하면 작업이 어려워지는 단점이 있음

2. 인코더-디코더 어텐션 층

- 디코더의 중간 표현을 쿼리처럼 사용해 인코더 스택의 출력 키와 값 벡터에 멀티 헤드 어텐션을 수행함

- 두 개의 다른 시퀀스(ex) 두 개의 다른 언어)에 있는 토큰을 연관짓는 방법을 학습
- 디코더는 각 블록에서 인코더의 키와 값을 참조함

💡 셀프 어텐션 층과 달리, 인코더-디코더 어텐션의 키와 쿼리 벡터는 길이가 다른 경우가 존재.

이는 인코더와 디코더 입력이 일반적으로 길이가 다른 시퀀스를 다루기 때문임.

결과적으로 이 층의 어텐션 점수 행렬은 정사각행렬이 아닌 직사각행렬.

▼ 비유로 인코더-디코더 어텐션 이해하기

📌 **디코더**가 수업 중에 시험을 본다 하자. 이전 단어(**디코더 입력**)을 기반으로 다음 단어를 예측하는 시험이다. 이는 간단해 보이지만 어려운 시험이다. 다행히 짝꿍(**인코더**)이

전체 텍스트를 갖고 있다. 그런데 안타깝게도 외국 교환 학생이라 텍스트가 짝꿍의

모국어로 되어 있다. 잔꾀 많은 디코더는 텍스트를 간단한 그림(**쿼리**)으로 그려서

짝꿍에게 준다. 짝꿍은 설명에 매칭된 구절(**키**)을 찾아내고 이 구절의 다음 단어를

설명하는 그림(**값**)을 그려 디코더에게 다시 보낸다.

▫ 마스크드 셀프 어텐션 층

셀프 어텐션 층에 마스킹을 포함시키기 위해 필요한 수정 내용 살펴보기

▼ 마스크 행렬 코드

```
seq_len = inputs.input_ids.size(-1)
mask = torch.tril(torch.ones(seq_len, seq_len)).unsqueeze(0)
mask[0]
```

파이토치 tril() 함수 이용하여 하삼각행렬 만들기

```
tensor([[1., 0., 0., 0., 0.],
        [1., 1., 0., 0., 0.],
        [1., 1., 1., 0., 0.],
        [1., 1., 1., 1., 0.],
        [1., 1., 1., 1., 1.]])
```

마스크드 셀프 어텐션은 대각선 아래는 1이고 대각선 위는 0인 **마스크 행렬**을 도입해 구현

▼ Tensor.maseked_fill()

```
scores.masked_fill(mask == 0, -float("inf"))
```

- Tensor.maseked_fill()을 사용해 0을 음의 무한대로 바꾸면 어텐션 헤드가 미래 토큰을 보지 못함
- 대각선 위의 값을 음의 무한대로 설정하면, 소프트맥스 함수를 적용할 때 $e^{(-\infty)} = 0$

  이므로 어텐션 가중치가 모두 0이 된다. (소프트맥스 함수는 정규화된 지수 값을 계산)

▼ 초반에 구현한 스케일드 점곱 어텐션 함수를 수정해 마스킹 동작 추가하기

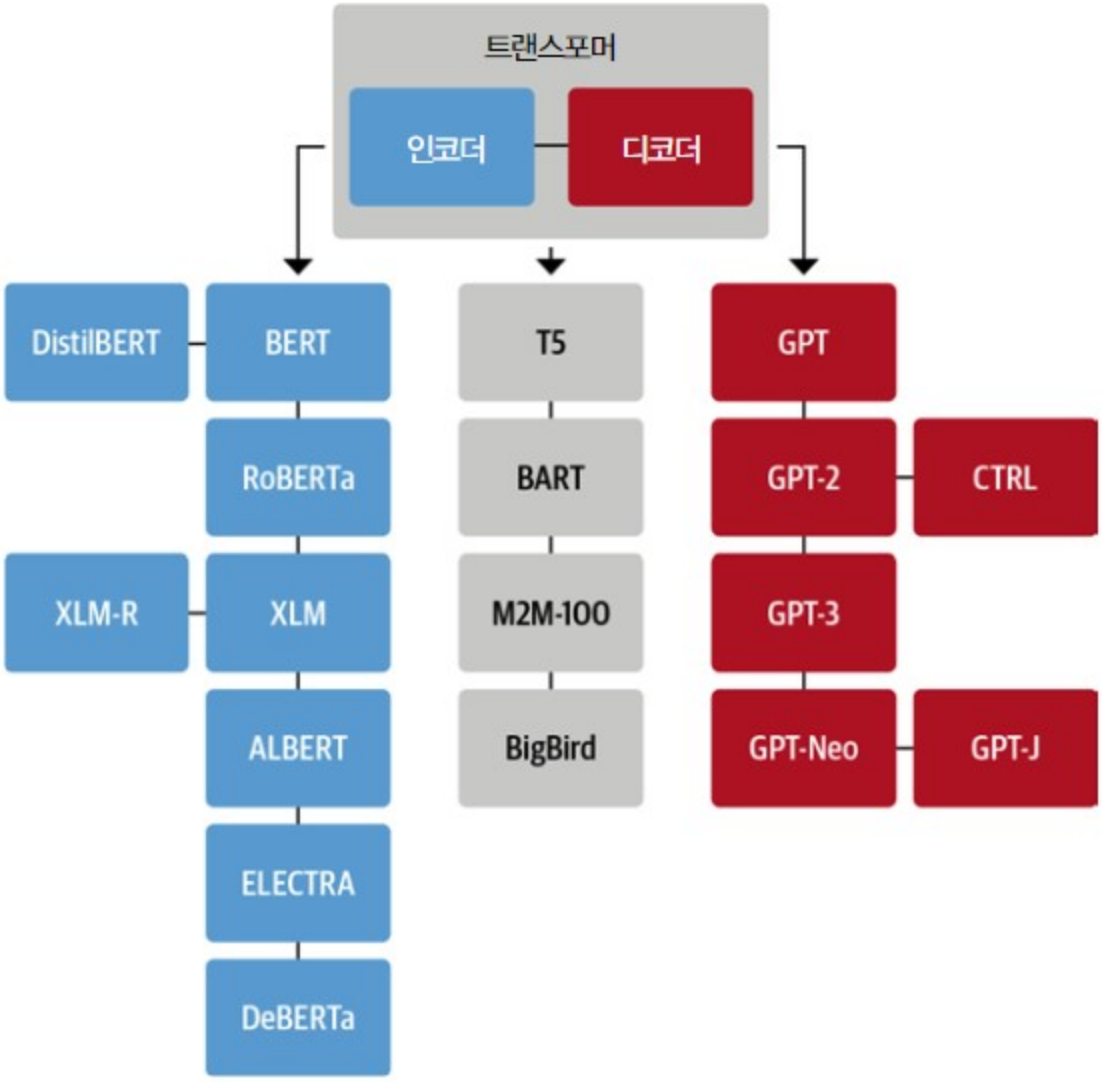
```
def scaled_dot_product_attention(query, key, value, mask=None
  dim_k = query.size(-1)
  scores = torch.bmm(query, key.transpose(1, 2)) / sqrt(dim_k
  if mask is not None:
    scores = scores.masked_fill(mask == 0, float("-inf"))
  weights = F.softmax(scores, dim=-1)
  return weights.bmm(value)
```

여기부터 디코더 층을 만드는 작업은 간단하므로 생략

▫ 트랜스포머 유니버스

트랜스포머 가계도

그림은 가장 유망한 모델과 그 후손을 나타낸다



▫ 인코더 유형

- 트랜스포머 아키텍처 기반의 첫 인코더 유형은 BERT
- 공개될 당시, 유명한 GLUE 벤치마크에서 최상의 모델을 모두 능가하였음

  +) GLUE 벤치마크 : 난이도가 다양한 작업으로 자연어 이해를 측정함

- 인코더 유형은 연구와 NLU 작업에 널리 쓰이며 텍스트 분류, 개체명 인식, 질문 답변 등이 속함

BERT

BERT의 사전 훈련 목표

1. 마스크드 언어 모델링 (MLM) : 텍스트에서 마스킹된 토큰을 예측하는 것
2. 다음 문장 예측 (NSP) : 한 텍스트 구절이 다른 텍스트 구절 뒤에 나올 확률을 판단하는 것

DistilBERT

DistilBERT는 사전 훈련 과정에서 지식 정제 기술을 사용하여

BERT보다 적은 메모리를 사용하고 더 빠르면서 성능은 유사

RoBERTa

RoBERTa는 BERT의 사전 훈련 방법을 수정해 더 많은 훈련 데이터로 더 큰 배치에서

더 오래 훈련하며 NSP(다음 문장 예측) 작업을 포함하지 않고 성능을 향상시킴

XLM

GPT 유사 모델의 자기회귀 언어 모델링과 BERT의 MLM을 포함해

XLM에서 다중 언어 모델을 만들기 위한 사전 훈련 방법이 연구되었음.

또 XLM 사전 훈련 논문 저자들은 MLM을 다중 언어 입력으로 확장한 TLM을 소개하였음

XLM-RoBERTa

XLM-RoBERTa나 XLM-R 모델이 훈련 데이터를 대규모로 확장해 다중 언어 사전 훈련을 발전시킴

커먼 크롤 말뭉치를 사용해 2.5TB의 텍스트 데이터를 만들고 데이터셋에서 MLM으로 인코더를 훈련

이 데이터셋은 상대 텍스트(번역)가 없는 데이터만 포함하기 때문에 XML의 TML 목표가 제외됨

데이터가 부족한 언어에서 유용함

ALBERT

인코더 구조를 더 효율적으로 만들기 위해

1. 은닉 차원에서 토큰 임베딩 차원을 분리해 임베딩 차원을 줄임

   >> 어휘사전이 큰 경우 파라미터가 절약됨
2. 모든 층이 동일한 파라미터를 공유함으로써 실제 파라미터 개수를 크게 줄임
3. NSP 목표를 문장 순서 예측으로 바꿈.

   즉 모델은 두 문장이 함께 속해 있는지가 아니라 연속된 두 문장의 순서가 바뀌었는지를 예측

이런 변경을 통해 더 적은 파라미터로 더 큰 모델을 훈련하고 NLU 작업에서 뛰어난 성능을 달성함

ELECTRA

표준적인 MLM 사전 훈련 목표는 각 훈련 스텝마다 마스킹된 토큰 표현만 업데이트되고

그 외 입력 토큰은 업데이트되지 않는다는 제약이 있어 이를 해결하고자 두 개의 모델 사용

모델 1. 표준적인 마스크드 언어 모델처럼 동작해 마스킹된 토큰을 예측

모델 2. 판별자 (discriminator)란 이름의 두 번째 모델은 첫 번째 모델의 출력에 있는 토큰 중

어떤 것이 원래 마스킹된 것인지 예측하는 작업을 수행함. 따라서, 판별자는 모든 토큰에

이진 분류를 수행하는데, 이를 통해 훈련 효율을 높임. 판별자는 후속 작업을 위해

표준 BERT 모델처럼 미세 튜닝됨

DeBERTa

이 모델은 두 가지 구조 변경이 존재

1. 각 토큰이 두 개의 벡터로 표현됨

   하나는 콘텐츠용이고 하나는 상대 위치를 위한 것이다.

   토큰 콘텐츠와 상대 위치를 분리하면 셀프 어텐션 층이 인접한 토큰 쌍의 의존성을 더 잘 모델링.

2. 토큰 디코딩 헤드의 소프트맥스 층 직전에 절대 위치 임베딩 추가

   디코딩에서 단어의 절대 위치가 중요하기 때문.

▫ 디코더 유형

- OpenAI는 트랜스포머 디코더 모델의 발전을 주도
- 디코더 모델은 문장에서 다음 단어를 예측하는데 뛰어나므로 대부분 텍스트 생성 작업에 사용

GPT

GPT는 NLP의 핵심인 트랜스포머 디코더 아키텍처와 전이 학습을 결합

모델은 이전 단어를 기반으로 다음 단어를 예측하도록 훈련됨

이 모델은 BookCorpus에서 훈련되고 분류와 같은 후속 작업에서 뛰어난 결과를 달성

GPT-2

원본 모델과 훈련 세트를 확장해 만든 모델로 일관성 있는 긴 텍스트 시퀀스를 만듦

남용 가능성이 우려되어 모델을 단계적으로 릴리스함

CTRL

GPT-2 같은 모델은 입력 시퀀스 (prompt라고 부르기도 함)의 뒤를 이음

하지만 생성된 시퀀스의 스타일은 거의 제어하지 못함.

CTRL 모델은 시퀀스 시작 부분에 **제어토큰**을 추가해 이 문제를 해결

이를 통해 생성 문장의 스타일을 제어해 다양한 문장을 생성함

GPT-3

다양한 규모에서 언어 모델의 동작을 자세히 분석해 계산량, 데이터셋 크기, 모델 크기,

언어 모델 성능의 관계를 관장하는 간단한 거듭제곱 규칙을 발견하였음

이런 통찰에서 착안해, 파라미터가 1,750억 개인 GPT-3을 만듦

- GPT-3 모델은 사실적인 텍스트 구절을 생성할 뿐 아니라 퓨샷 학습 능력도 갖춤
- 텍스트를 코드로 변환하는 것 같이 모델이 새로운 작업에서 샘플 몇 개를 학습해 새로운 샘플 처리 가능

GPT-Neo / GPT-J-6B

이 두 모델은 GPT-3 규모의 모델을 만들고 릴리스하기 위한 EleutherAI에서 훈련한 GPT 유사 모델

현재 모델은 1,750억 개 파라미터가 있는 모델보다 작은 버전으로

13억 개, 27억 개, 60억 개 파라미터가 있음.

▫ 인코더-디코더 유형

T5

T5 모델은 모든 NLU와 NLG 작업을 텍스트-투-텍스트 작업으로 변환해 통합

모든 작업이 시퀀스-투-시퀀스 문제로 구성되므로 인코더-디코더 구조를 선택

T5 아키텍처는 원본 트랜스포머 아키텍처를 사용함

이 모델은 대규모로 크롤링된 C4 데이터셋을 사용하며 텍스트-투-텍스트 작업으로

변환된 SuperGLUE 작업과 마스크드 언어 모델링으로 사전 훈련됨

BART

BART는 인코더-디코더 아키텍처 안에 BERT와 GPT의 사전 훈련 과정을 결합

입력 시퀀스는 간단한 마스킹에서 문장 섞기, 토큰 삭제, 문서 순환에 이르기까지

가능한 여러 가지 변환 중 하나를 거친다

변경된 입력이 인코더를 통과하면 디코더는 원본 텍스트를 재구성

이는 모델을 유연하게 만들어 NLU와 NLG 작업에 모두 사용 가능, 양쪽에서 최상의 성능 달성

M2M-100

M2M-100은 100개 언어를 번역하는 최초의 번역 모델로 데이터가 부족한 언어의 번역에 활용

이 모델은 **접두어 토큰** ([CLS] 특수 토큰과 유사)을 사용해 소스 언어와 타깃 언어를 나타냄

BigBird

트랜스포머 모델은 어텐션 메커니즘에 필요한 메모리가

시퀀스 길이의 제곱에 비례하기 때문에 최대 문맥 크기라는 제약점이 존재

BigBird는 선형적인 크기가 늘어나는 **희소 어텐션**을 사용해 이 문제를 해결

문맥 크기가 대부분의 BERT 모델에서 사용하는 512토큰에서 4,096으로 크게 늘어남

텍스트 요약과 같이 긴 의존성을 보존해야 할 때 유용

❗ 이 절에서 다룬 모든 모델의 사전 훈련된 체크포인트를 허깅페이스 허브에서 제공 중