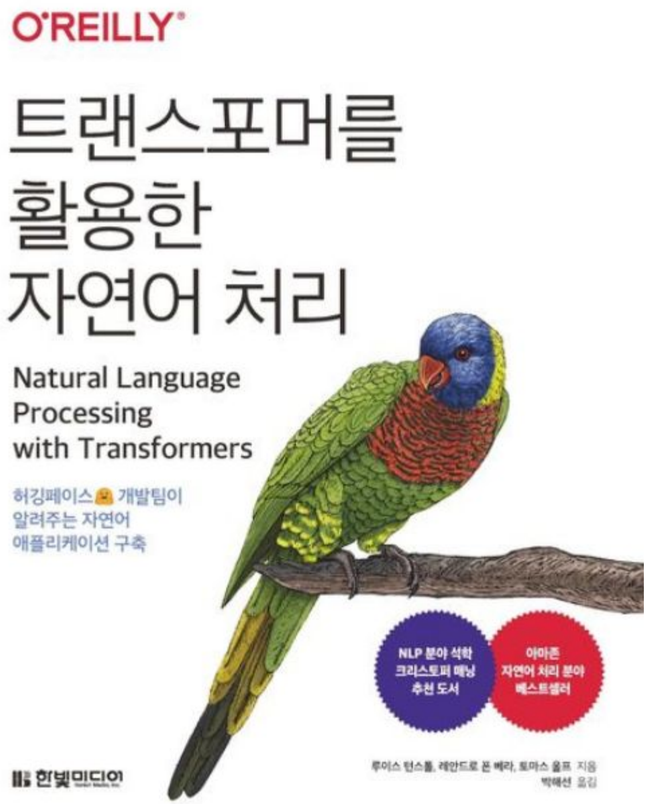
트랜스포머를 활용한 자연어처리

3장: 트랜스포머 파헤치기



소속
HUMANE랩

발표자
이다현

발표일시
2024년 07월 08일

24년 하계방학 스터디

목차

트랜스포머 아키텍처

- 인코더
  - 입력 토큰의 시퀀스를 hidden state 혹은 context라 부르는 임베딩 벡터로 변환
- 디코더
  - 인코더의 hidden state를 사용해 출력 토큰의 시퀀스를 한 번에 하나씩 반복적으로 생성
- 디코더 층마다 인코더의 출력이 주입됨
- 디코더는 시퀀스에서 가장 가능성 있는 다음 토큰을 예측
- 예측한 토큰은 디코더로 다시 주입되어 다음 토큰 생성

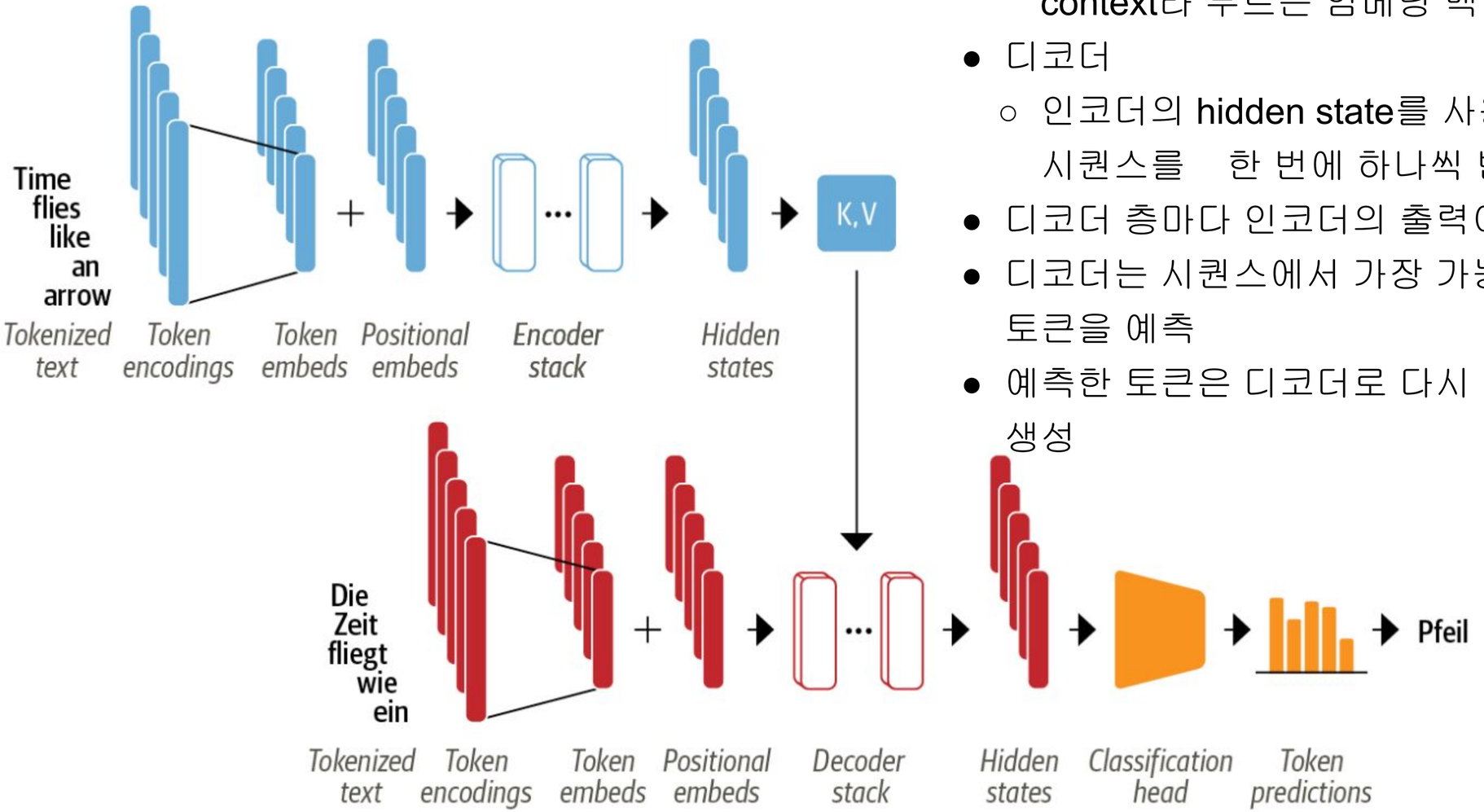


그림 3-1 트랜스포머의 인코더-디코더 아키텍처

유형

항목	세부 내용	모델
인코더 유형	· 텍스트 시퀀스 입력을 수치 표현으로 변환 · 텍스트 분류, 개체명 인식과 같은 작업에 적합 · 토큰에 대해 계산한 표현은 양쪽 문맥의 영향을 받음 · 양방향 어텐션(bidirectional attention)	BERT, RoBERTa, DistilBERT
디코더 유형	· 텍스트가 주어지면 다음 단어를 반복해 예측 · 토큰에 대해 계산한 표현은 왼쪽 문맥에만 영향을 받음 · 자기회귀 어텐션(autoregressive attention)	**GPT 계열 모델**
인코더-디코더 유형	· 한 텍스트의 시퀀스를 다른 시퀀스로 매핑할 때 사용 · 기계 번역과 요약 작업에 적합	트랜스포머 아키텍처 BART T5

인코더

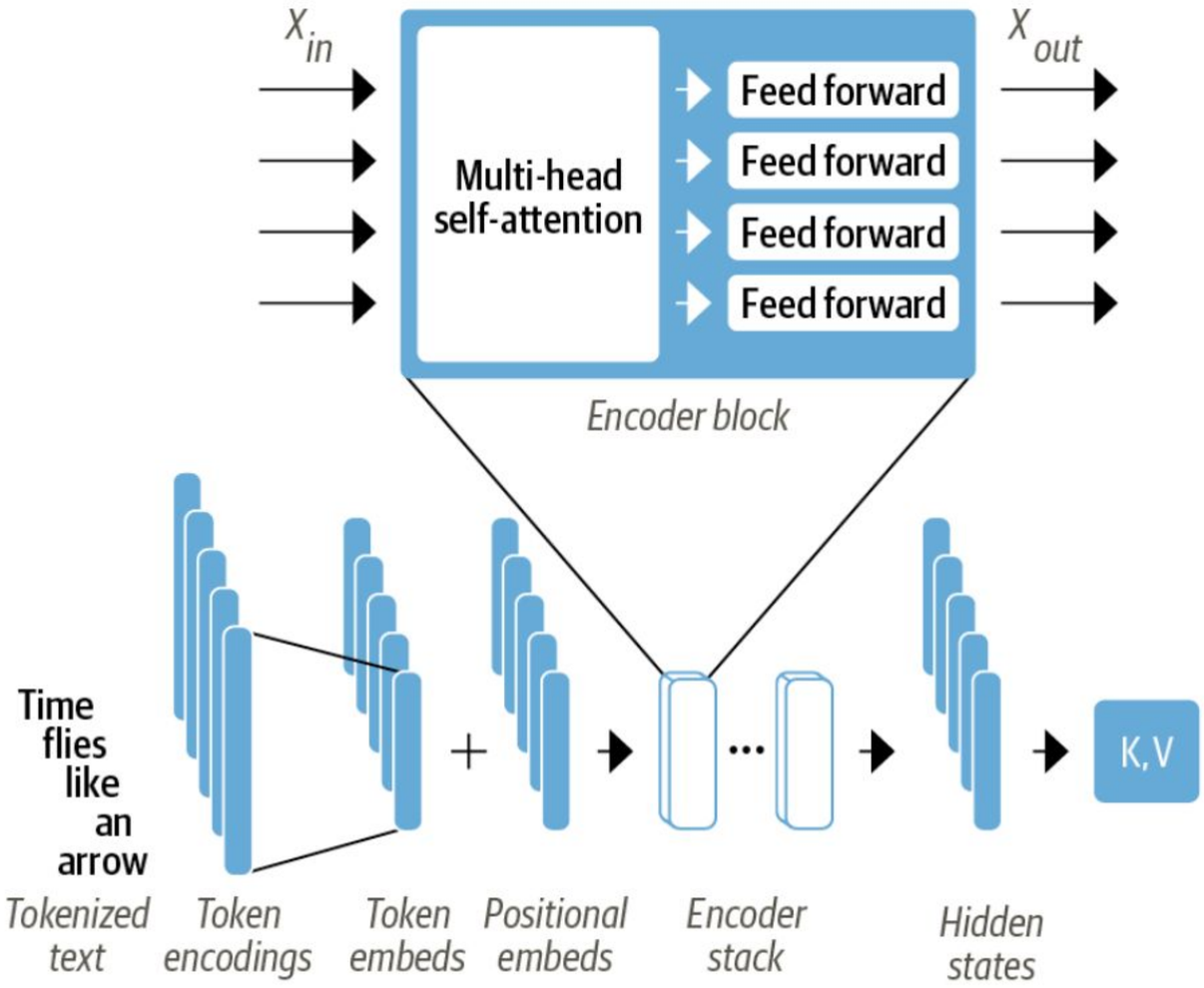


그림 3-2 인코더 층을 확대한 그림

- 임베딩 시퀀스를 입력으로 받아 다음 layer에 통과
  - 멀티 헤드 셀프 어텐션 layer
  - 각각 입력 임베딩에 적용되는 완전 연결 피드 포워드 layer
- 입력 임베딩과 크기가 동일한 임베딩 출력
  - 시퀀스의 문맥 정보가 반영되도록 입력 임베딩 업데이트

self-attention

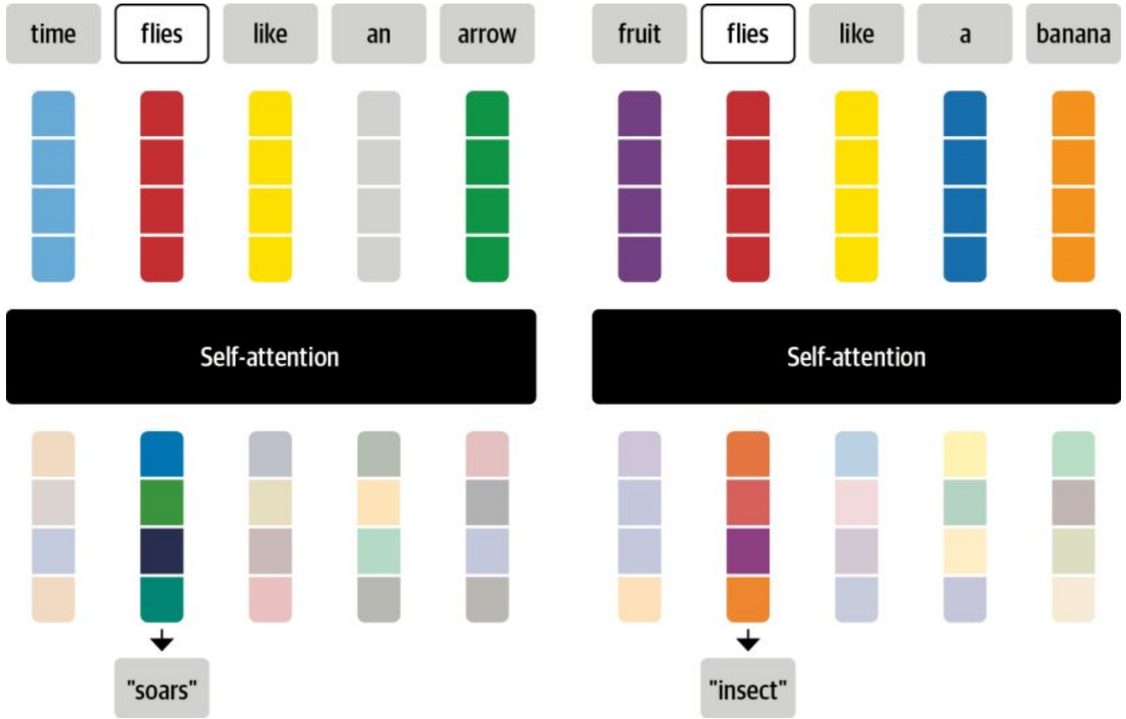


그림 3-3 self attention이 문맥 고려 임베딩을 업데이트해서 전체 시퀀스 정보를 통합하는 표현을 만드는 방법

- 어텐션 메커니즘: 신경망이 시퀀스의 각 원소에 다른 양의 가중치(attention)을 할당
- self-attention?
  - 입력 시퀀스의 모든 hidden state에 대해 계산
- 순환 모델과 연관된 attention
  - 특정 디코딩 타임스텝에서 디코더의 hidden state와 인코더의 각 hidden state가 가진 관련성을 계산
- self-attention의 기본 개념: contextualized embedding
  - 시퀀스에 대해 고정된 임베딩을 사용하는 대신 전체 시퀀스를 사용해 각 임베딩의 가중 평균을 계산
  - 토큰 임베딩의 시퀀스(x_1, x_2, ..., x_n)이 주어지면
  - 새로운 임베딩 시퀀스(x_1', x_2', ..., x_n')을 생성
  - x_i'는 x_j의 선형 결합
  - 계수 w_{ji} = 어텐션 가중치
    - 합이 1이 되도록 정규화 됨

$$\mathbf{x}_i' = \sum_{j=1}^{n} w_{ij}\mathbf{x}_j$$

문맥 고려 임베딩 계산 공식

Scaled dot-production attention

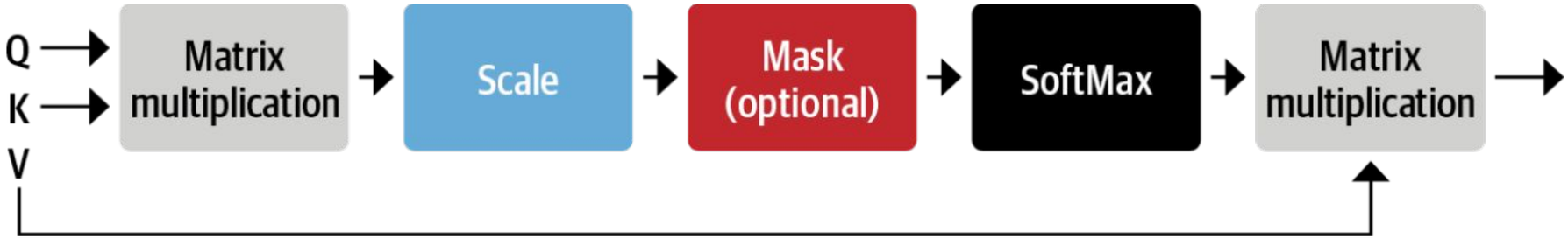


그림 3-4 scaled dot-production attention의 연산과정

1. 각 토큰 임베딩을 쿼리, 키, 값 세 개의 벡터로 투영
   - Query: 필요한 식재료
   - Key: 마트 진열대에 붙은 이름표
   - Value: 최종적으로 고른 상품(필요한 식재료(Query)와 가장 유사하다고 여겨지는 이름표(Key)가 붙은 상품)
2. 어텐션 점수 계산
   - 유사도 함수(여기서는 dot-product)를 사용해 Query Key가 서로 얼마나 관련되는지 계산
3. 어텐션 가중치 계산
4. 토큰 임베딩 업데이트
   - 어텐션 가중치와 Value 벡터를 곱해서 업데이트된 토큰 임베딩 표현을 얻음

Scaled dot-production attention

- 토큰 인코딩

```
from transformers import AutoTokenizer
model_ckpt = "bert-base-uncased"
text = "time flies like an arrow"
tokenizer = AutoTokenizer.from_pretrained(model_ckpt)
```

```
inputs = tokenizer(text, return_tensors="pt", add_special_tokens=False)
inputs.input_ids
```

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

- 입력 임베딩의 크기 확인

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

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

- 쿼리, 키, 값 벡터 설정 및 어텐션 스코어 계산

```
import torch
from math import sqrt

query = key = value = inputs_embeds
dim_k = key.size(-1)
scores = torch.bmm(query, key.transpose(1,2)) / sqrt(dim_k)
scores.size()
```

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

- 어텐션 가중치 계산

```
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>)
```

- 토큰 임베딩 업데이트

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

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

```
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)
```

Multi-head attention

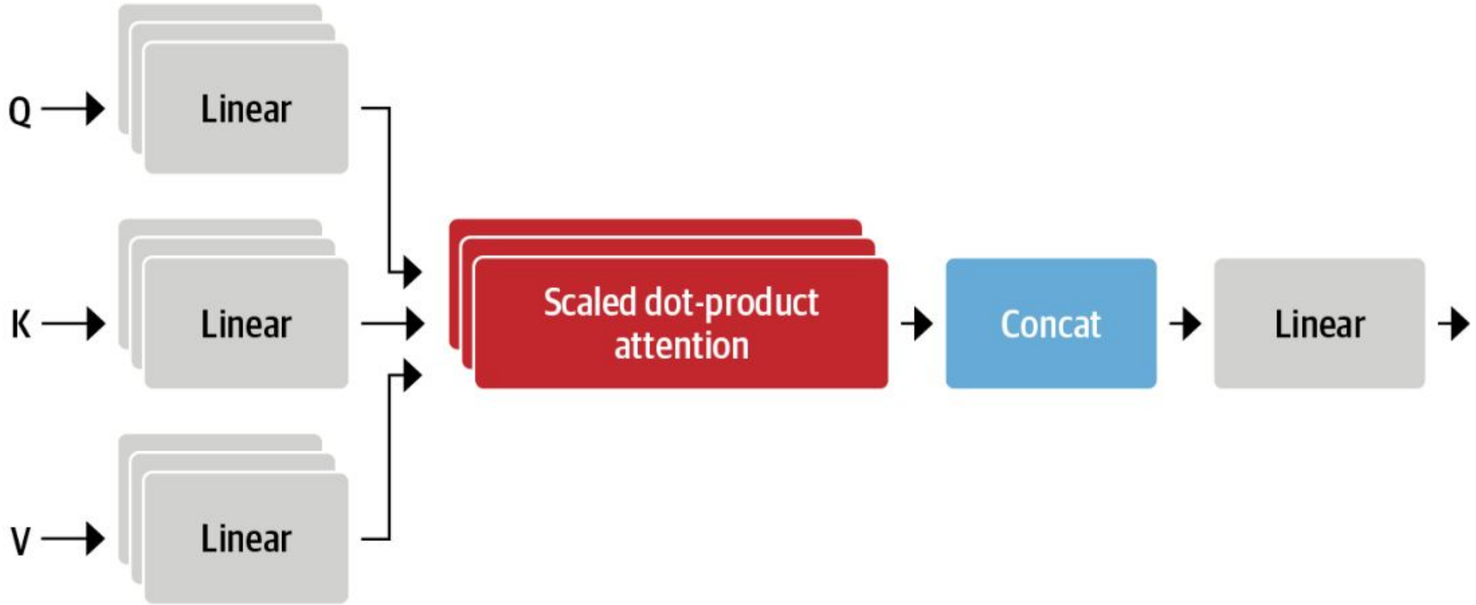


그림 3-5 Multi-head attention

- Attention Head: 입력 시퀀스의 각 단어에 대해 독립적인 쿼리, 키, 값 벡터를 생성하고 어텐션을 계산하는 기본 단위
- Why Multi-head?
  - 하나의 헤드의 소프트맥스가 유사도의 한 측면에만 초점을 맞추는 경향이 있기 때문
  - 여러 개의 헤드가 있으면 동시에 다양한 관계와 유사성을 학습할 수 있어, 모델이 더 정교하고 풍부한 표현을 생성할 수 있음

Multi-head attention

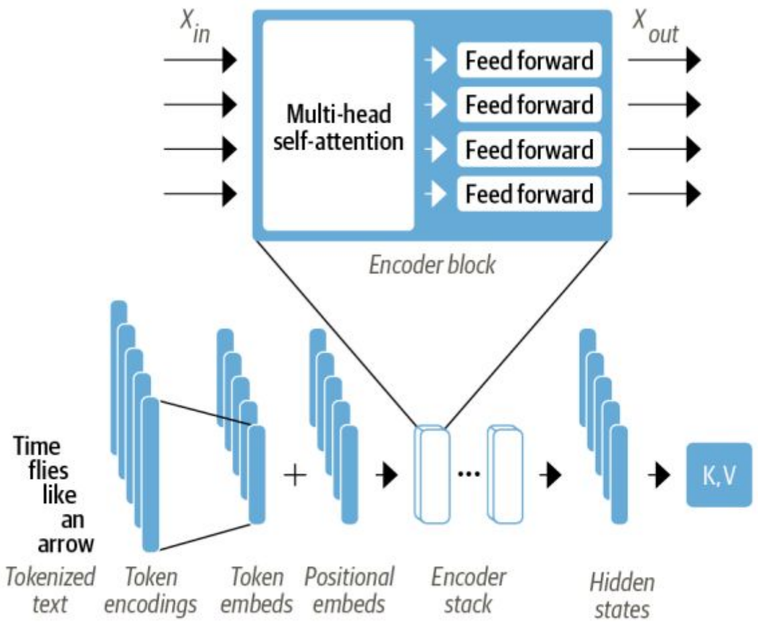
```
class AttentionHead(nn.Module):
    def __init__(self, embed_dim, head_dim):
        super().__init__()
        self.q = nn.Linear(embed_dim, head_dim)
        self.k = nn.Linear(embed_dim, head_dim)
        self.v = nn.Linear(embed_dim, head_dim)

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

```
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 range(num_heads)]
        )
        self.output_linear = nn.Linear(embed_dim, embed_dim)

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

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

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

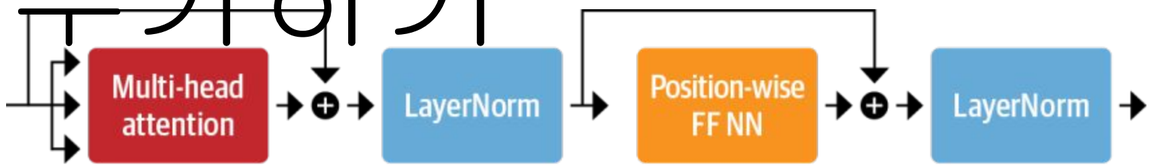
feed-forward layer



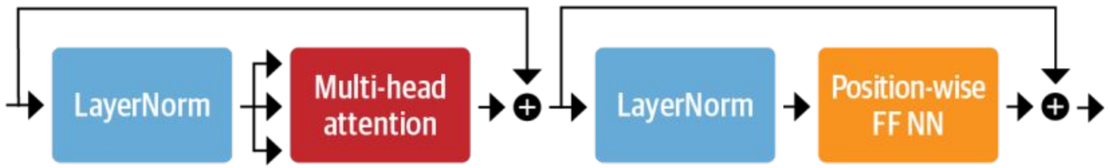

```python
class FeedForward(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.linear_1 = nn.Linear(config.hidden_size, config.intermediate_size)
        self.linear_2 = nn.Linear(config.intermediate_size, config.hidden_size)
        self.gelu = nn.GELU()
        self.dropout = 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
```

- 두 개의 층으로 구성된 완전 연결 신경망
- 각 임베딩을 독립적으로 처리(position-wise feed-forward layer)
- 출력 텐서 크기 = torch.Size([1, 5, 768])

층 정규화 추가하기



사후 층 정규화



사전 층 정규화

```python
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 복사합니다.
        hidden_state = self.layer_norm_1(x)
        # 어텐션에 스킵 연결을 적용합니다.
        x = x + self.attention(hidden_state)
        # 스킵 연결과 피드 포워드 층을 적용합니다.
        x = x + self.feed_forward(self.layer_norm_2(x))
        return x
```

그림 3-6 트랜스포머 인코더 층의 층 정규화 배치 방법

- skip connection: 처리하지 않은 텐서를 모델의 다음 층으로 전달 후 처리된 텐서와 더함
- 층 정규화: batch의 각 입력을 정규화(평균 0, 분산 1)
  - 사후 층 정규화: skip connection 사이에 층 정규화 위치
    - 그레디언트가 발산하는 경우가 생김 → learning rate warm-up(훈련하는 동안 학습률을 점진적으로 증가시키는 것) 사용
  - 사전 층 정규화: skip connection 안에 층 정규화 위치

위치 임베딩

```
class Embeddings(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.token_embeddings = nn.Embedding(config.vocab_size,
                                             config.hidden_size)
        self.position_embeddings = nn.Embedding(config.max_position_embeddings,
                                                config.hidden_size)
        self.layer_norm = nn.LayerNorm(config.hidden_size, eps=1e-12)
        self.dropout = nn.Dropout()

    def forward(self, input_ids):
        # 입력 시퀀스에 대해 위치 ID를 만듭니다.
        seq_length = input_ids.size(1)
        position_ids = torch.arange(seq_length, dtype=torch.long).unsqueeze(0)
        # 토큰 임베딩과 위치 임베딩을 만듭니다.
        token_embeddings = self.token_embeddings(input_ids)
        position_embeddings = self.position_embeddings(position_ids)
        # 토큰 임베딩과 위치 임베딩을 합칩니다.
        embeddings = token_embeddings + position_embeddings
        embeddings = self.layer_norm(embeddings)
        embeddings = self.dropout(embeddings)
        return embeddings
```

- 학습 가능한 패턴 사용
  - 벡터에 나열된 값의 위치 패턴으로 토큰 임베딩을 보강
  - 각 스택에 있는 어텐션 헤드와 피드 포워드 층은 위치 정보와 변환을 통합하는 방법을 배움
- 절대 위치 표현
  - 변조된 sine 및 cosine 신호로 구성된 정적 패턴을 사용해 위치를 인코딩
  - 가용할 데이터가 많지 않을 때 특히 잘 동작
- 상대 위치 표현
  - 임베딩을 계산할 때는 주위 토큰을 더 중요하게 여길 수도 있음
  - 주의를 기울이는 시퀀스의 위치에 따라 토큰에 대한 상대적 임베딩이 바뀜
  - 어텐션 메커니즘 자체에 토큰의 상대 위치를 고려하는 항을 추가
    - 예: DeBERTa 모델

분류 헤드 추가하기

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

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

- 임베딩과 인코더 층을 연결한 완전한 트랜스포머 인코더

```python
class TransformerForSequenceClassification(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.encoder = TransformerEncoder(config)
        self.dropout = nn.Dropout(config.hidden_dropout_prob)
        self.classifier = nn.Linear(config.hidden_size, config.num_labels)

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

- 시퀀스 분류를 위해 기존 인코더를 확장
  - 드롭아웃 층과 선형 층을 추가
- 배치 내 샘플의 출력 클래스마다 정규화되지 않은 logit 반환
- 첫 번째 토큰을 예측에 사용

디코더

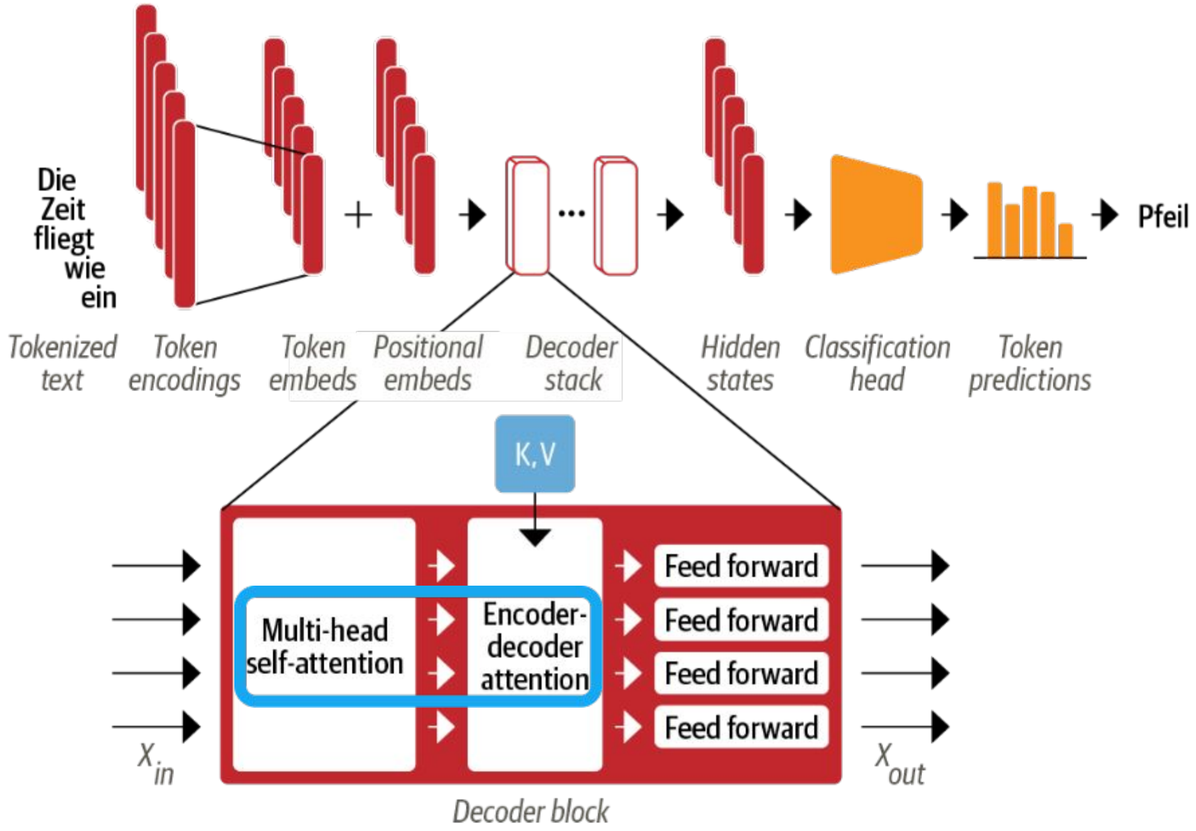


그림 3-7 트랜스포머 디코더 층을 확대한 그림

- Masked Multi-head self-attention Layer
  - 타임스텝마다 지난 출력과 예측한 현재 토큰만을 사용하여 토큰 생성
  - 훈련 동안 단순히 타깃 번역을 복사하는 것을 예방
- Encoder-decoder attention Layer
  - 디코더의 중간 표현을 Query처럼 사용하여 인코더 스택의 출력 Key, Value 벡터에 멀티 헤드 어텐션 수행
  - 두 개의 다른 시퀀스에 있는 토큰을 연관짓는 방법을 학습
  - 각 블록에서 인코더의 Key, Value 참조

디코더

```
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.]])
```

→ 마스크 행렬

```
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)
```

- 값을 음의 무한대("-inf")로 설정하면, 소프트맥스 함수 적용 시 어텐션 가중치가 0이 됨

트랜스포머 유니버스

- 인코더 유형
  - 사용하는 경우: Context를 만드는데까지만 관심 있고 생성(Decoder)은 관심이 없는 경우
- 디코더 유형
  - 사용하는 경우: LM이 하는 것처럼 주어진 텍스트 다음에 올 단어 예측에만 관심 있고 문장 → 문장 변환에 관심 없는 경우

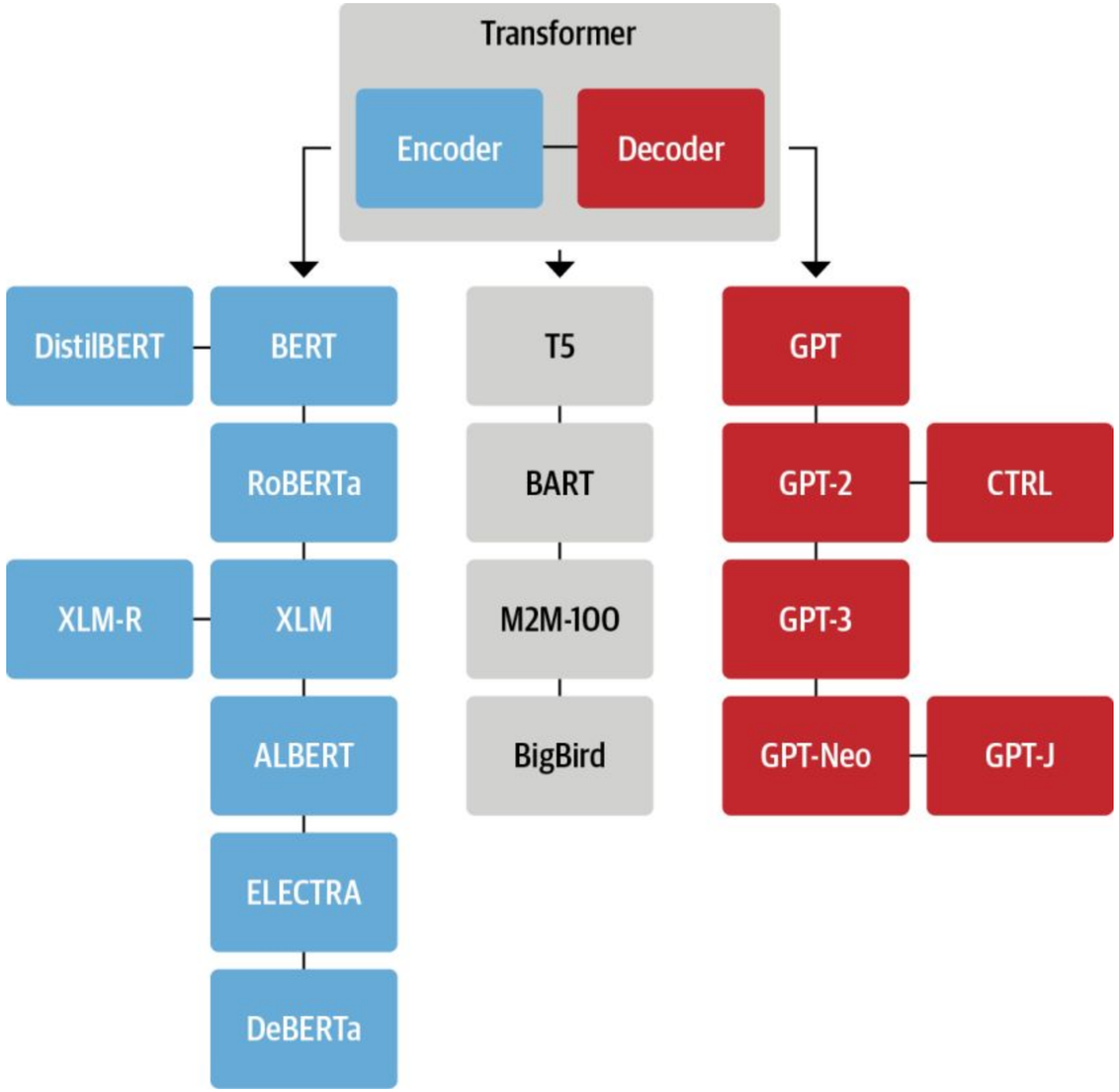


그림 3-8 가장 대표적인 트랜스포머 아키텍처

마무리

- Self-attention: 트랜스포머 아키텍처의 핵심
- 트랜스포머 인코더 모델을 만드는데 필요한 구성 요소
  - 위치 정보를 위한 임베딩 층 추가
  - 어텐션 헤드를 보완하기 위한 피드 포워드 층
  - 예측을 위한 분류 헤드 추가
- 트랜스포머 디코더

다음 장은 다중 언어 개체명 인식 모델