Chapter 02 LLM의 중추, 트랜스포머 아키텍처 살펴보기

2024 초급 LLM with AI application

이동준

RNN과의 비교 및 한계로부터 나타난 트랜스포머 아키텍처

1. 시퀀스 처리의 순차적 특성

- 시퀀스를 처리할 때 순차적으로 처리하여 속도가 느려지는 RNN과 달리 트랜스포머는 시퀀스 전체를 한 번에 병렬로 처리할 수 있어 속도가 매우 빠르다

2. 그래디언트 소실(gradient vanishing)

- RNN은 긴 시퀀스에서 장거리 데이터 간 학습하기 어려운 문제가 있었던 반면 트랜스포머는 시퀀스 내 모든 단어가 서로 주목(attention)하게 만들어 그래디언트 소실 문제를 획기적으로 해결했다

3. GPT, BERT 등 현재는 트랜스포머의 시대

- 트랜스포머의 높은 성능은 갖추되, 빠른 학습이 가능한 새로운 아키텍처로 맘바(Mamba)가 대두(16장)

트랜스포머 아키텍처

1. Embedding
2. Positional Encoding
3. Multi-Head Attention
4. Encoder-Decoder Architecture
5. Feed forward Neural Network
6. Layer Normalization

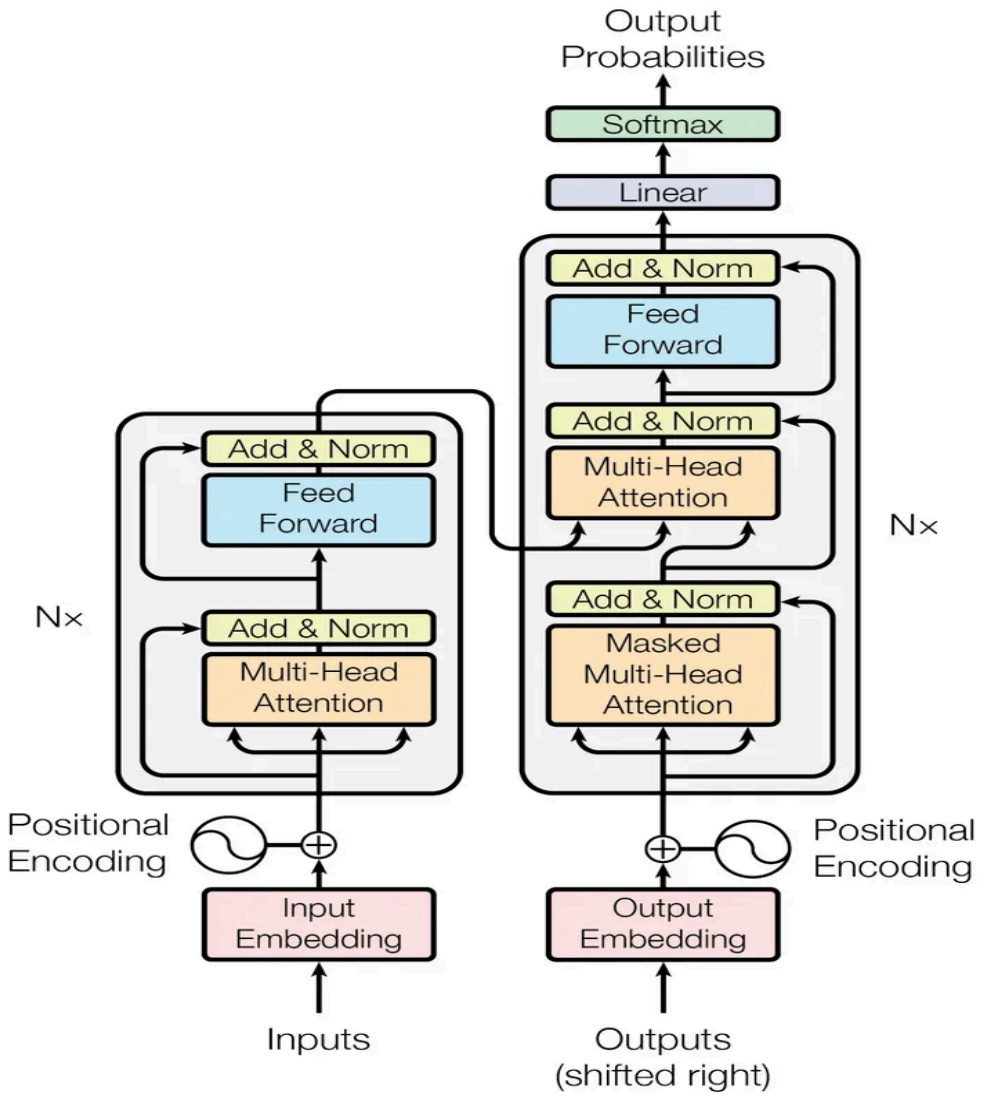

Figure 1: The Transformer - model architecture.

트랜스포머 아키텍처 : 임베딩(Embedding)

1. **임베딩(Embedding)** 은 입력 데이터를 **고차원 벡터** 로 변환하는 과정
   트랜스포머는 텍스트를 처리할 때, 단어를 고차원 공간의 벡터로 표현한다. 임베딩 벡터는 텍스트의 의미를 반영하며, 그 벡터는 어텐션 메커니즘에서 중요한 역할을 수행한다. 임베딩을 통해 모델이 단어 간의 관계와 문맥을 학습 가능할 수 있다.

2. 임베딩의 필요성
   컴퓨터는 정량적 수치로만 작업을 처리하므로 텍스트 데이터를 수치적 표현으로 바꿔야 한다. 단어를 벡터로 변환함으로써, 컴퓨터는 단어 간의 의미적 관계를 학습하고, 이를 바탕으로 유사성이나 패턴을 분석할 수 있다.

3. 임베딩의 원리
   임베딩은 주로 고차원 벡터 공간에서 각 단어를 희소(sparse)하지 않게 **밀집 벡터(Dense Vector)** 로 표현한다. 임베딩은 단어 간의 유사성을 벡터 간의 거리로 나타낼 수 있다.

트랜스포머 아키텍처 : 임베딩(Embedding)

예제 2.1 입력 문장을 받아서 토큰화

```
# 띄어쓰기 단위로 분리
input_text = "나는 최근 파리 여행을 다녀왔다"
input_text_list = input_text.split()
print("input_text_list: ", input_text_list)

# 토큰 -> 아이디 딕셔너리와 아이디 -> 토큰 딕셔너리 만들기
str2idx = {word:idx for idx, word in enumerate(input_text_list)}
idx2str = {idx:word for idx, word in enumerate(input_text_list)}
print("str2idx: ", str2idx)
print("idx2str: ", idx2str)

# 토큰을 토큰 아이디로 변환
input_ids = [str2idx[word] for word in input_text_list]
print("input_ids: ", input_ids)
```

트랜스포머 아키텍처 : 임베딩(Embedding)

예제 2.2 토큰화한 입력 아이디를 벡터로 변환

```
import torch
import torch.nn as nn # 파이토치 라이브러리 사용해서 임베딩

embedding_dim = 16
embed_layer = nn.Embedding(len(str2idx), embedding_dim) # 파이토치 라이브러리 사용

input_embeddings = embed_layer(torch.tensor(input_ids)) # (5, 16)
input_embeddings = input_embeddings.unsqueeze(0) # (1, 5, 16)
input_embeddings.shape
```

트랜스포머 아키텍처 : 포지셔널 인코딩(Positional Encoding)

1. 포지셔널 인코딩 : 임베딩 데이터에 순서를 매기는 과정
2. 포지셔널 인코딩의 원리
   (책에서는 간단하게 구현했지만) 사인 함수와 코사인 함수의 주기에 따라 결정된다. 단어 임베딩에 위치 정보를 포함한 벡터를 더해, 각 단어의 위치를 명시적으로 모델에 알려주고, 트랜스포머는 단어 간의 순서를 인식하게 된다.
3. 사인 함수와 코사인 함수를 사용하는 이유
   i. 위치값이 너무 크면 안된다
   ii. 위치값의 차이를 거리로 사용할 수 있어야 한다
   iii. 값은 어느 정도 일정할 필요가 있다

참고

- https://gaussian37.github.io/dl-concept-positional_encoding/#positional-encoding의-필요성-1

트랜스포머 아키텍처 : 포지셔널 인코딩(Positional Encoding)

예제 2.3 절대적 위치 인코딩. 여기서는 단순하게 인덱싱 처리함

```
embedding_dim = 16
max_position = 12
# 토큰 임베딩 층 생성
embed_layer = nn.Embedding(len(str2idx), embedding_dim)
# 위치 인코딩 층 생성
position_embed_layer = nn.Embedding(max_position, embedding_dim)

position_ids = torch.arange(len(input_ids), dtype=torch.long).unsqueeze(0)
position_encodings = position_embed_layer(position_ids)
token_embeddings = embed_layer(torch.tensor(input_ids)) # (5, 16)
token_embeddings = token_embeddings.unsqueeze(0) # (1, 5, 16)
# 토큰 임베딩과 위치 인코딩을 더해 최종 입력 임베딩 생성
input_embeddings = token_embeddings + position_encodings
input_embeddings.shape
```

트랜스포머 아키텍처 : 포지셔널 인코딩(Positional Encoding)

번외. 사인 코사인 함수를 이용해서 정말로 구해낸 포지셔널 인코딩
https://gaussian37.github.io/dl-concept-positional_encoding/#positional-encoding의-필요성-1

```
import numpy as np

def getPositionEncoding(seq_len, d, n=10000):
    P = np.zeros((seq_len, d))
    for k in range(seq_len):
        for i in np.arange(int(d/2)):
            denominator = np.power(n, 2*i/d)
            P[k, 2*i] = np.sin(k/denominator)
            P[k, 2*i+1] = np.cos(k/denominator)
    return P
```

트랜스포머 아키텍처 : 어텐션(Attention)의 구조

1. 어텐션
   어텐션 메커니즘은 기존 모델이 가진 문제를 해결하여 모델이 입력 시퀀스의 모든 위치를 ***한 번에 고려*** 하여 중요한 정보에 더 많은 가중치를 부여하는 방식으로, 장거리 의존성 문제를 해결하고 병렬 처리를 가능하게 한다.
2. *Q*uery, *K*ey, *V*alue 벡터
   i. Query(Q): 현재 단어의 정보 요청을 나타내는 벡터
   ii. Key(K): 각 단어의 특성 정보를 나타내는 벡터
   iii. Value(V): 각 단어가 담고 있는 실제 정보를 나타내는 벡터
3.

$$\text{Attention}(Q, K, V) = \text{Softmax}\left(\frac{Q \cdot K^T}{\sqrt{d_k}}\right)V$$

- Q와 K의 벡터 내적을 계산하여 유사도 측정
- $\sqrt{d_k}$(임베딩 차원 수의 제곱근)으로 스케일링하여 너무 큰 값 방지
- **소프트맥스(softmax)** 를 적용해 각 단어에 대한 가중치를 계산
- 가중치와 V를 곱해서 최종 출력

트랜스포머 아키텍처 : 어텐션(Attention)의 구조

예제 2.4 쿼리, 키, 값 벡터를 만드는 nn.Linear 층

```
head_dim = 16

# 쿼리, 키, 값을 계산하기 위한 변환
weight_q = nn.Linear(embedding_dim, head_dim)
weight_k = nn.Linear(embedding_dim, head_dim)
weight_v = nn.Linear(embedding_dim, head_dim)
# 변환 수행
querys = weight_q(input_embeddings) # (1, 5, 16)
keys = weight_k(input_embeddings) # (1, 5, 16)
values = weight_v(input_embeddings) # (1, 5, 16)
```

트랜스포머 아키텍처 : 어텐션(Attention)의 구조

예제 2.4 어텐션을 계산하는 방법

```
from math import sqrt
import torch.nn.functional as F

def compute_attention(querys, keys, values, is_causal=False):
        dim_k = querys.size(-1) # 16
        scores = querys @ keys.transpose(-2, -1) / sqrt(dim_k) # @ 표시는 행렬 곱셈
        weights = F.softmax(scores, dim=-1)
        return weights @ values
```

$$\text{Attention}(Q, K, V) = \text{Softmax}\left(\frac{Q \cdot K^T}{\sqrt{d_k}}\right)V$$

트랜스포머 아키텍처 : 피드포워드 네트워크(Feed forward)

1. 피드포워드 네트워크
   피드포워드 네트워크는 **잔차 연결(residual connection)** 과 **드롭아웃(dropout)** 과 결합하여 트랜스포머의 깊은 학습에서 학습의 안정성을 유지하고 과적합을 방지하는 데 기여한다.

2. 드롭아웃(dropout)
   딥러닝 모델에서 과적합(overfitting)을 방지하기 위한 기법 중 하나로. 여기서는 이를 완화하기 위해 신경망의 학습 과정 중 일부 뉴런을 무작위로 꺼버리는 방법을 적용한다.

3. 잔차 연결(residual connection)
   잔차 연결(Residual Connection)은 딥러닝 모델에서 학습이 더 잘 되도록 돕는 기법으로, 여기서는 어텐션 결과에 드롭아웃을 취한 다음, 입력값에 더한다.

4. 층 정규화(layer normalization)
   신경망에서 각 층의 출력을 정규화하는 방법으로(배치를 정규화하면 배치 정규화라고 한다), 여기서는 사전 정규화와 사후 정규화 중 효과가 검증된 방법인 사전 정규화를 수행한다.

트랜스포머 아키텍처 : 피드포워드 네트워크(Feed forward)

예제 2.10. 피드 포워드 층 코드

```
class PreLayerNormFeedForward(nn.Module):
  def __init__(self, d_model, dim_feedforward, dropout):
    super().__init__()
    self.linear1 = nn.Linear(d_model, dim_feedforward) # 선형 층 1
    self.linear2 = nn.Linear(dim_feedforward, d_model) # 선형 층 2
    self.dropout1 = nn.Dropout(dropout) # 드랍아웃 층 1
    self.dropout2 = nn.Dropout(dropout) # 드랍아웃 층 2
    self.activation = nn.GELU() # 활성 함수
    self.norm = nn.LayerNorm(d_model) # 층 정규화

  def forward(self, src):
    x = self.norm(src)
    x = x + self.linear2(self.dropout1(self.activation(self.linear1(x))))
    x = self.dropout2(x)
    return x
```

트랜스포머 아키텍처 : 인코더-디코더 구조

1. 인코더(Encoder)
   인코더는 입력 시퀀스를 받아 이를 고차원 벡터로 변환하는 역할을 수행하며, 인코더는 다음과 같은 구조를 갖는다.

   i. 멀티헤드 어텐션(Multi-head Attention): 입력 시퀀스 간의 관계를 학습하며, 각 위치의 토큰이 다른 모든 위치의 토큰과 상호작용한다.
   ii. 피드포워드 네트워크(Feed-Forward Network): 각 위치에서 독립적으로 작동하며 비선형 변환을 적용한다.

2. 디코더(Decoder)
   디코더는 인코더의 출력을 받아 최종 출력 시퀀스를 생성하는 역할을 수행하며, 디코더는 다음과 같은 구조를 갖는다.

   i. 셀프 어텐션(Self-Attention): 디코더의 이전 출력 토큰들을 기반으로 다음 토큰을 예측한다. 여기서는 미래의 토큰을 참조하지 못하게 하는 **마스크 어텐션(Masked Attention)** 이 사용된다.
   ii. 크로스 어텐션(Cross-Attention): 인코더에서 나온 정보와 디코더의 정보를 결합하여 현재 시점에서 어떤 정보를 사용할지 결정한다.

트랜스포머 아키텍처 : 인코더-디코더 구조

예제 2.11. 인코더 층

```
class TransformerEncoderLayer(nn.Module):
  def __init__(self, d_model, nhead, dim_feedforward, dropout):
    super().__init__()
    self.attn = MultiheadAttention(d_model, d_model, nhead) # 멀티 헤드 어텐션 클래스
    self.norm1 = nn.LayerNorm(d_model) # 층 정규화
    self.dropout1 = nn.Dropout(dropout) # 드랍아웃
    self.feed_forward = PreLayerNormFeedForward(d_model, dim_feedforward, dropout) # 피드포워드

  def forward(self, src):
    norm_x = self.norm1(src)  # 사전 정규화
    attn_output = self.attn(norm_x, norm_x, norm_x)  # 멀티 헤드 어텐션
    x = src + self.dropout1(attn_output) # 잔차 연결

    # 피드 포워드
    x = self.feed_forward(x)
    return x
```

트랜스포머 아키텍처 : 인코더-디코더 구조

예제 2.14. 크로스 어텐션이 포함된 디코더 층

```
class TransformerDecoderLayer(nn.Module):
  def __init__(self, d_model, nhead, dim_feedforward=2048, dropout=0.1):
    super().__init__()
    self.self_attn = MultiheadAttention(d_model, d_model, nhead)
    self.multihead_attn = MultiheadAttention(d_model, d_model, nhead)
    self.feed_forward = PreLayerNormFeedForward(d_model, dim_feedforward, dropout)
    self.norm1 = nn.LayerNorm(d_model)
    self.norm2 = nn.LayerNorm(d_model)
    self.dropout1 = nn.Dropout(dropout)
    self.dropout2 = nn.Dropout(dropout)

  def forward(self, tgt, encoder_output, is_causal=True):
    # 셀프 어텐션 연산
    x = self.norm1(tgt)
    x = x + self.dropout1(self.self_attn(x, x, x, is_causal=is_causal))
    # 크로스 어텐션 연산
    x = self.norm2(x)
    x = x + self.dropout2(self.multihead_attn(x, encoder_output, encoder_output))
    # 피드 포워드 연산
    x = self.feed_forward(x)
    return x
```

트랜스포머 아키텍처 : 인코더-디코더 구조

모델	아키텍처 구조	인코더 사용	디코더 사용	주요 사용 분야	특징
GPT	디코더 기반	X	O	텍스트 생성	단방향 디코더 (왼쪽에서 오른쪽)로 작동, 언어 모델링에 특화
BERT	인코더 기반	O	X	텍스트 이해	양방향 인코더로 작동, 문맥 이해 및 텍스트 분류에 강점
BART, T5	인코더-디코더	O	O	다양한 텍스트 작업	BERT의 인코더 + GPT의 디코더

들어주셔서 감사합니다