Transformer and GPT (Generative Pre-trained Transformer)

2025. 8

Yongjin Jeong, KwangWoon University

GPT (Transformer Decoder)

- GPT의 정의
  - Generative (생성하는): 텍스트, 코드, 이미지 등을 생성.
  - Pre-trained (사전 학습된): 방대한 인터넷 데이터를 미리 공부함.
  - Transformer (트랜스포머): 문맥을 이해하는 신경망 아키텍처.
- 작동 원리
  - 다음에 올 단어를 확률적으로 예측.
  - 예시: "대한민국의 수도는 [?] 입니다." -> "서울"이 올 확률 99%
- 토큰(Token)의 개념:
  - AI가 글자를 인식하는 단위 (단어보다 작은 단위).
- GPT의 한계:
  - 할루시네이션 (Hallucination): 그럴듯한 거짓말. 사실 여부를 판단하지 못하고 확률적으로 문장을 완성하기 때문에 발생.
  - 최신 정보의 부재 (모델 학습 시점 이후의 정보는 모름 - *검색 기능 연동으로 보완.*

NLP 모델의 진화와 패러다임 시프트

- Seq2Seq와 RNN의 한계:
  - 순차적 처리(Sequential Processing)로 인한 병렬화 불가능 문제.
  - Long-term Dependency 문제: 문장이 길어질수록 앞단의 정보 소실 (Gradient Vanishing).
- Attention Is All You Need (2017):
  - Recurrence(순환)와 Convolution(합성곱)을 제거하고 Attention만으로 모델 구성.
  - 병렬 처리 가능 -> 대규모 학습(Scaling)의 기반 마련.
- **Transformer(Encoder vs Decoder):**
  - BERT (Encoder-only): 양방향 문맥 이해, 분류/추출 태스크에 강점.
  - GPT (Decoder-only): 단방향(Autoregressive) 예측, 생성 태스크에 특화.

Seq2seq based on RNN

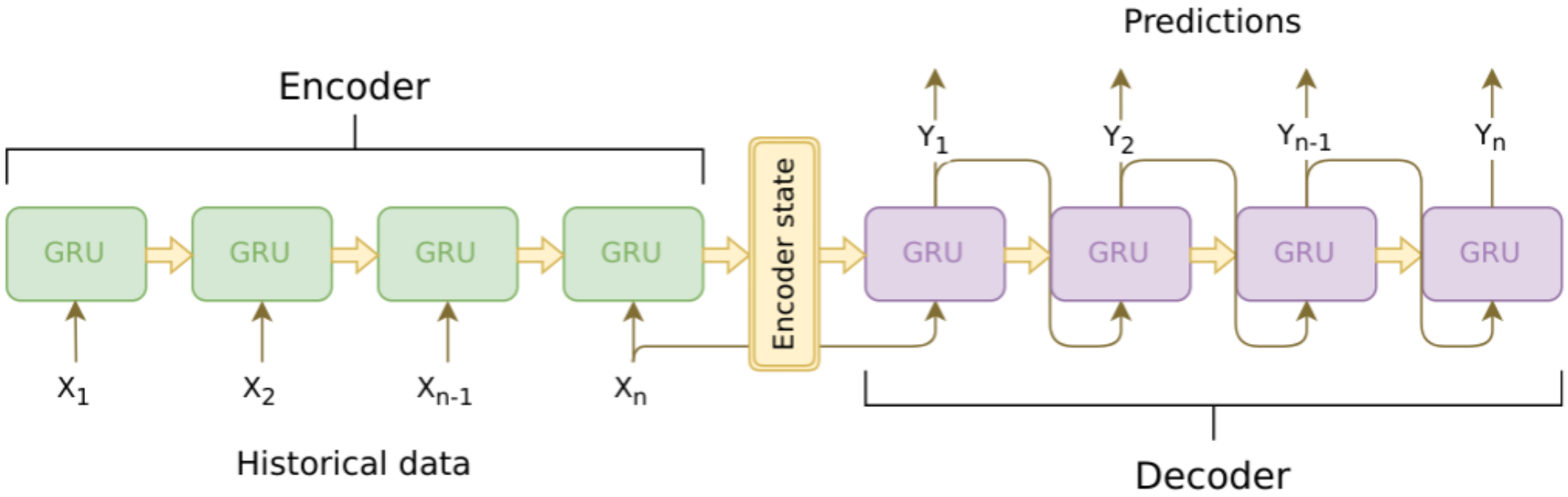


Image source: https://jeddy92.github.io/JEddy92.github.io/ts_seq2seq_intro/

- One fixed context vector (the last hidden state) is transferred to Decoder.

Attention with RNN

simply, weighted average of some values

- **General RNN Encoder-decoder**
  - Encoder

$$h_t = f\left(x_t, h_{t-1}\right)$$

$$c = q\left(\{h_1, \cdots, h_{T_x}\}\right)$$

x=($x_1, \ldots, x_{Tx}$): input sequence
$h_t \in R^n$: a hidden state at time *t*,
c: context vector generated from sequence of hidden states

  - Decoder:
    - trained to predict the next word $y_{t'}$ given the context vector (*c*) and all the previously predicted words $\{y_1, \ldots, y_{t'-1}\}$

$$p(\mathbf{y}) = \prod_{t=1}^{T} p(y_t \mid \{y_1, \cdots, y_{t-1}\}, c)$$

y=($y_1, \ldots, y_{Ty}$): output sequence

with RNN, modeled as

$$p(y_t \mid \{y_1, \cdots, y_{t-1}\}, c) = g(y_{t-1}, s_t, c)$$

S_t : hidden state of RNN

Attention with RNN – Align and Translate

- **Decoder with Attention**

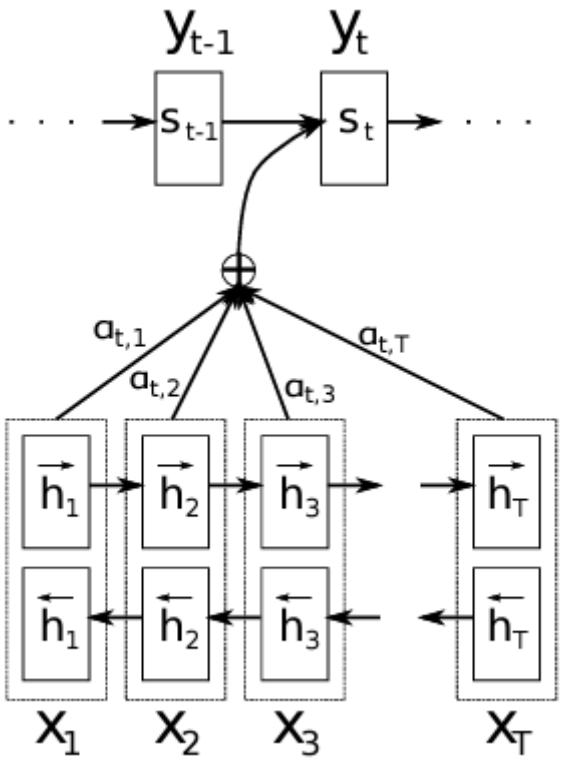
In a new model architecture, we define each conditional probability in Eq. (2) as:

$$p(y_i|y_1,\ldots,y_{i-1},\mathbf{x}) = g(y_{i-1}, s_i, c_i), \qquad (4)$$

where s_i is an RNN hidden state for time i, computed by

$$s_i = f(s_{i-1}, y_{i-1}, c_i).$$

Unlike the encoder–decoder approach(see Eq.(2)), here the probability is conditioned on a distinct context vector c_i for each target word y_i.



$$c_i = \sum_{j=1}^{T_x} \alpha_{ij} h_j$$ (weighted sum of annotations h_i)

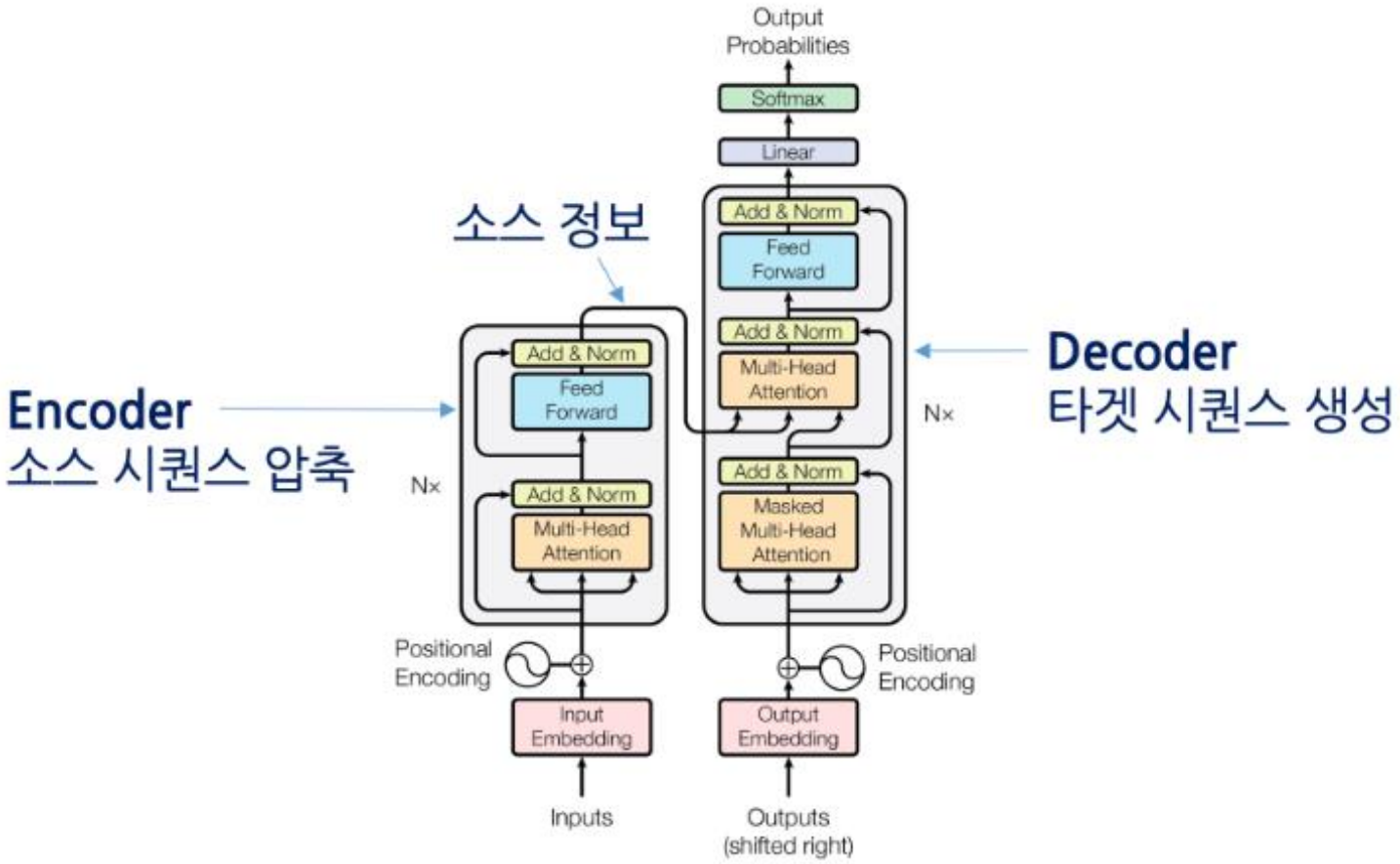
$$h_j = \left[\overrightarrow{h}_j^\top ; \overleftarrow{h}_j^\top\right]^\top$$

weights ⇒ $$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^{T_x} \exp(e_{ik})}$$,where $$e_{ij} = a(s_{i-1}, h_j)$$

alignment model ; it scores how well the inputs around position j and the output at position i match. The score is based on the RNN hidden states i−1(just before emitting yi) and the j-th annotation hj of the input sentence. (can be parameterized as a Forward Neural Network)

The Transformer

- **Without RNN**



Encoder:

- Multi-head self-attention mechanism
- A simple position-wise fully connected feed-forward network

Decoder:

- **Masked** Multi-head self-attention mechanism (ensures that the predictions for position i can depend only on the known outputs at positions less than i)
- A simple position-wise fully connected feed-forward network
- Multi-head attention over the output of the encoder stack

https://ratsgo.github.io/nlpbook/docs/language_model/tr_self_attention/

Transformer 아키텍처

- Input Processing:
  - Tokenization: BPE (Byte Pair Encoding) 알고리즘 이해.
  - Embedding Layer: 고차원 벡터 공간으로의 매핑.
  - Positional Encoding: 순서 정보가 없는 Transformer에 위치 정보 주입 (Sinusoidal 함수 활용).
- Transformer 구조:
  - Self-Attention Mechanism:
  - Feed-Forward Networks (FFN):
    - 각 위치마다 독립적으로 적용되는 완전 연결 신경망 (GELU/ReLU 활성화 함수).
  - Residual Connection & Layer Normalization:
    - Deep Network 학습 안정을 위한 필수 요소 (Add & Norm).

Transformer 아키텍처

- Self-Attention Mechanism
  - Query(Q), Key(K), Value(V)의 개념.
  - Scaled Dot-Product Attention 수식:
  - Multi-Head Attention: 여러 개의 어텐션 헤드가 서로 다른 부분 공간(Subspace)의 정보를 학습.
- Feed-Forward Networks (FFN):
  - 각 위치마다 독립적으로 적용되는 완전 연결 신경망 (GELU/ReLU 활성화 함수).
- Residual Connection & Layer Normalization:
  - Deep Network 학습 안정을 위한 필수 요소 (Add & Norm).

Transformer의 세 가지 Attention

1. **Self-Attention in Encoder**
   - 역할: 입력 문장 전체의 문맥을 이해한다.
   - 작동 방식: 문장 속의 모든 단어(토큰)가 서로에게 질문(Q), 키(K), 값(V) 을 던진다.
   - 특징: 양방향 참조가 가능하다.

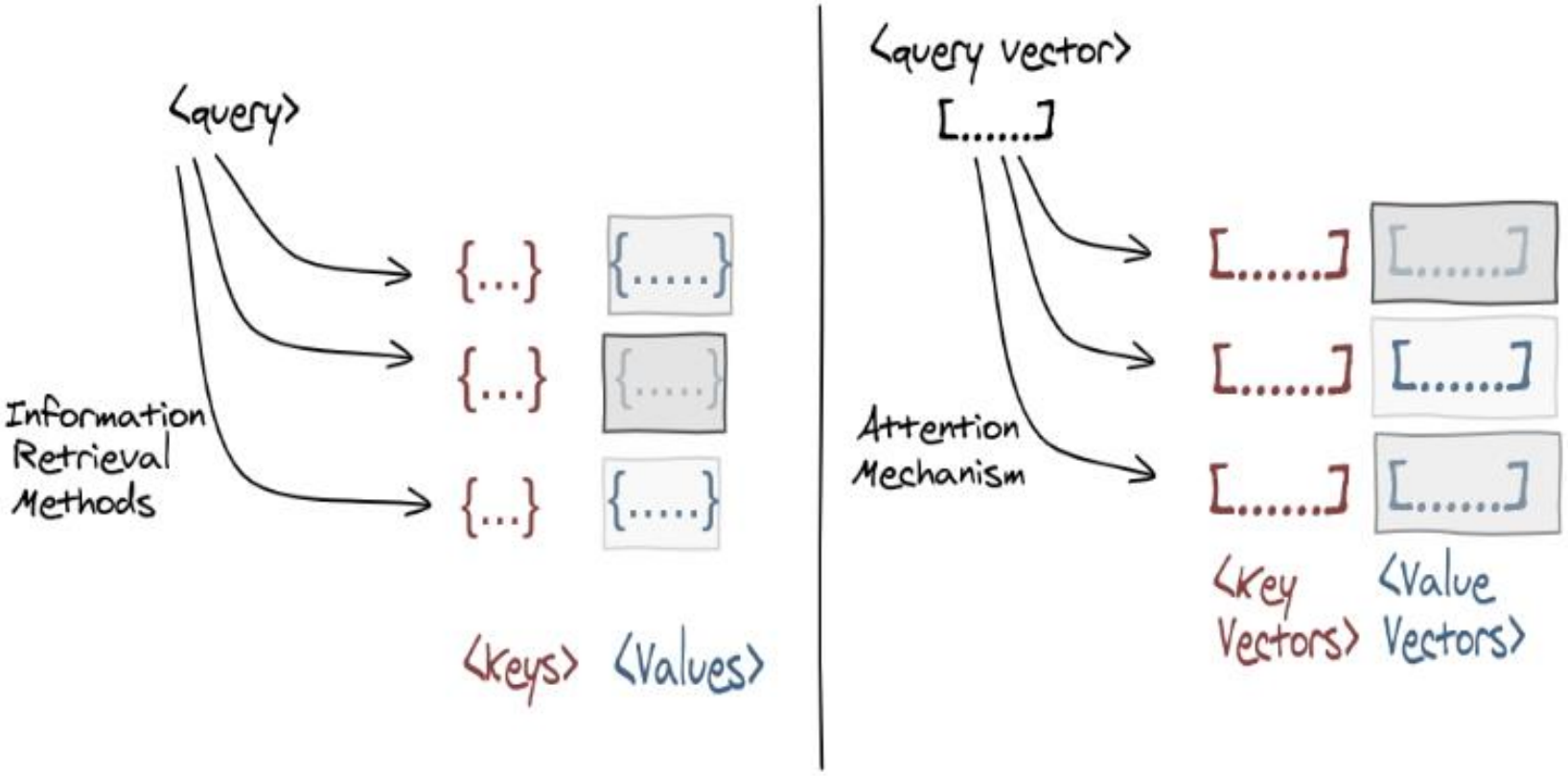
2. **Masked Self-Attention in Decoder**
   - 역할: 디코더가 출력을 순차적으로 생성(Generation)할 수 있게 한다.
   - 작동 방식: 모든 정보가 출력 시퀀스 자체에서 나오지만, 마스크(Mask)가 씌워져 있다.
   - 특징: 단방향 참조만 가능하다. 현재 토큰은 이미 생성된 이전 토큰들만 볼 수 있고, 아직 생성되지 않은 미래의 토큰은 볼 수 없다. (이것이 GPT와 같은 생성 모델의 기본 원리이다.)

3. **Encoder-Decoder Attention**
   - 역할: 인코더가 이해한 소스 문맥을 디코더가 참조할 수 있게 연결해 준다. (주로 번역과 같은 Seq2Seq 모델 사용)
   - 작동 방식:
     - 질문(Q)는디코더의 현재 상태 (지금까지 생성된 단어들)에서 온다.
     - 키(K)와 값(V)은 인코더의 최종 출력에서 온다.

※ GPT는 인코더-디코더 구조가 아닌 Decoder-only 구조를 사용하기 때문에, GPT 모델 내부에는 2번 Masked Self-Attention 만 존재한다.

Attention: Key, Query, Values

- **Key, Query, Values**



https://learningturtle.github.io/Blog/posts/attention_another_perspective/

Attention: Key, Query, Values

- **데이터베이스 vs 어텐션 비교**

구분	데이터베이스	어텐션(Self-Attention)
구조	`(key, value)` 쌍 저장	`(key, value)` 쌍 + Query
Query 동작	Query와 정확히 일치하는 Key를 찾음 → 해당 Value 반환	Query와 모든 Key의 **유사도(similarity)**를 계산
결과	하나의 Value 반환	모든 Value의 **가중합** 반환
가중치	없음 (정확 매칭)	$w_i = \text{softmax}(\text{similarity}(q, k_i))$
출력	특정 Value	컨텍스트 벡터 $y = \sum_i w_i v_i$

- 데이터베이스: 정확히 일치하는 항목을 찾는 검색
- 어텐션: 비슷한 항목들을 모두 참고해서, 중요도에 따라 평균을 내는 검색 (즉, 어텐션은 하나만 찾는 것이 아니라 문맥 전체를 요약하는 것임)

Self-Attention Mechanism

- **Q, K, V**의 직관적 이해 (검색 시스템 비유):
  - Query (Q): 검색어. 현재 분석 중인 토큰 (예: 문장 속의 "Bank"). "나와 관련된 정보가 있나요?" 라고 묻는 주체.
  - Key (K): 검색 대상의 태그/라벨. 모든 토큰이 가진 식별자. "나는 어떤 정보를 담고 있어요"라고 알리는 역할.
  - Value (V): 검색 결과 내용. 실제 정보의 값. Attention 점수에 따라 가중합 되어 가져와질 정보.
  - 작동 원리: Q와 K의 내적(Dot Product)을 통해 유사도(Attention Score)를 구하고, 그 점수만큼 V를 가져와 합친다.
- **Context Vector** (문맥 벡터)의 의미:
  - 정의: Attention 연산의 결과물 ($Z = \text{Softmax}(Q K^{\top}) V$).
  - 의미: 단순한 단어 임베딩(Static Embedding)이 주변 단어와의 관계를 흡수하여 '문맥이 반영된 표현(Contextualized Representation)'으로 변환된 것.
  - 예) "Apple"이라는 단어 주변에 "Eat"이 있으면 '과일' 의미의 벡터로, "MacBook"이 있으면 'IT 기업' 의미의 벡터로 변환됨.

Self-Attention Mechanism

- **Encoder vs Decoder에서의 Context Vector 차이**:
  - Encoder (BERT 등): Bi-directional Context.
    - Q가 자신의 앞뒤 모든 단어의 K를 참조함.
    - 문장 전체의 의미를 완벽하게 파악하여 압축하는 데 목적이 있음 (이해, 분류용).
  - Decoder (GPT 등): Uni-directional (Masked) Context.
    - Q가 자신의 이전(Past) 단어들의 K만 참조 가능 (미래 단어는 Masking 처리).
    - 현재까지의 대화 흐름(History)을 요약하고, 다음에 올 단어를 예측하기 위한 단서로서의 Context를 형성함.
- **Scaled Dot-Product Attention 수식**:

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

- Multi-Head Attention: 여러 개의 어텐션 헤드가 서로 다른 관점(문법적 관계, 의미적 관계 등)에서 정보를 수집.

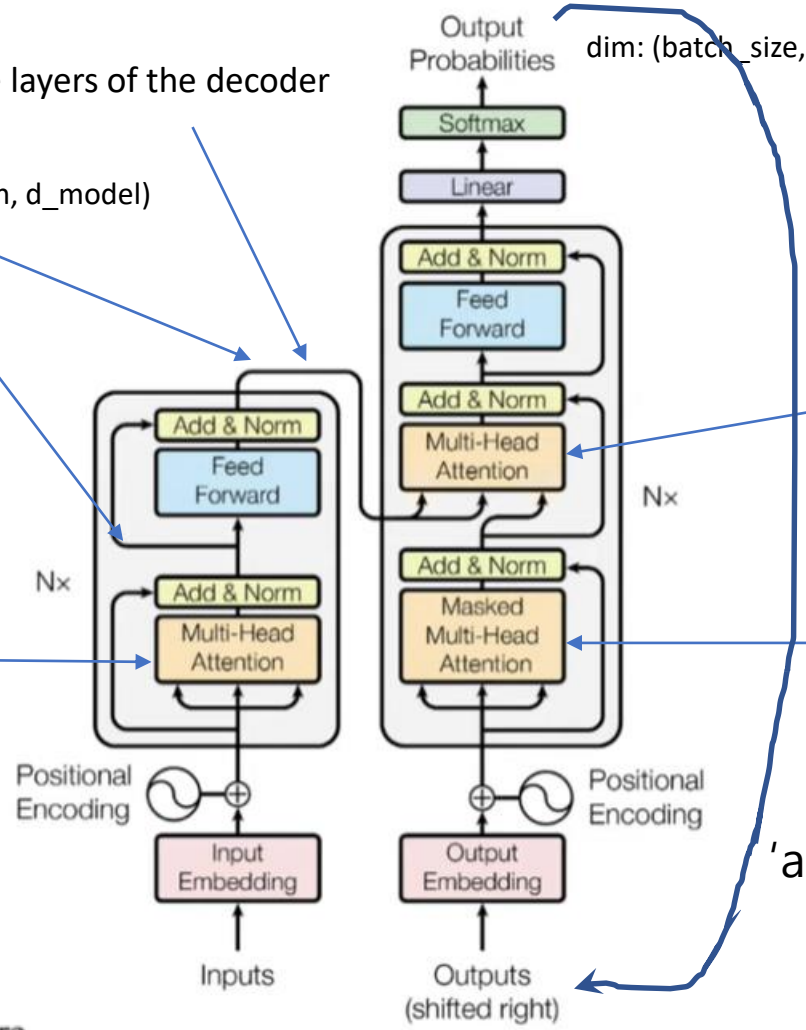
Attention by Query, Key, and Value

- Query, Key, Value
  - Simply generated with three different matrices (linear **transformation**) <- why?
  - **Query**: the input or prompt provided by the user.
  - **Key**: a piece of information used to retrieve relevant knowledge. (In the case of ChatGPT, the key is typically generated from the query.)
  - **Value**: the information retrieved by using the key
- What is the **Weight** for Value (Query * Key):
  - to calculate the (cosine) similarity or dot product between two vectors (a scalar value that represents the similarity or relevance between the key and the query)
  - Multiplying the key and query is a common mathematical operation used to calculate the similarity or **dot product** between two vectors.

Transformer

is used as the input for all the layers of the decoder

dim: (batch_size, seq_len, d_model)



dim: (batch_size, seq_len, d_model)

every position in the decoder to attend over all positions in the input sequence. This mimics the typical encoder-decoder attention mechanisms in sequence-to-sequence models (Query: represents the current position to be generated)

Each position in the encoder can attend to all positions in the previous layer of the encoder

each position in the decoder to attend to all positions in the decoder up to and including that position.

'autoregressive'

Figure 1: The Transformer - model architecture.

dim: (batch_size, seq_len, d_model) # typically, d_model=d_emb

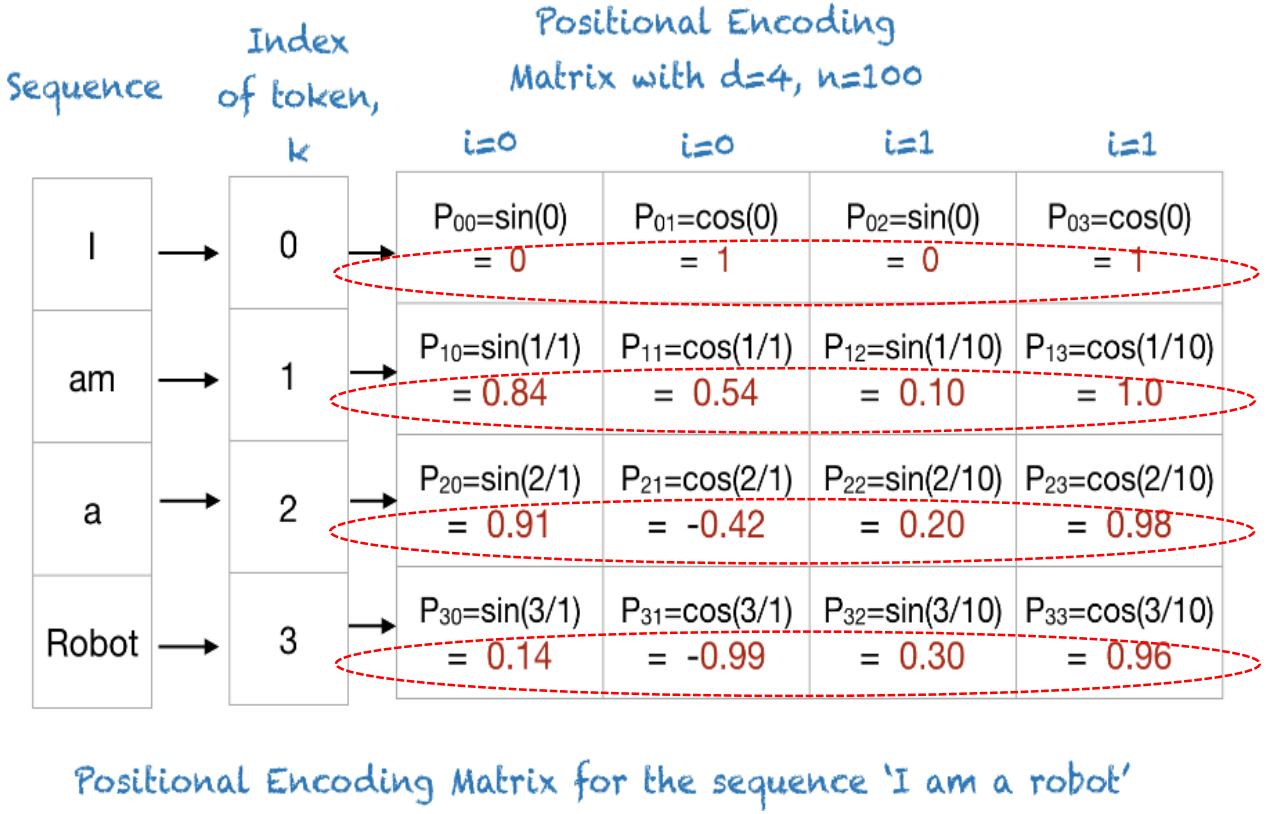
Positional Encoding

- Positional Encoding
  - Usually designed to have a small magnitude compared to the token embedding values, which allows the model to retain the meaningful information from the original token.
- Sinusoidal Positional Encoding

$$P(k, 2i) = \sin\left(\frac{k}{n^{2i/d}}\right)$$

$$P(k, 2i+1) = \cos\left(\frac{k}{n^{2i/d}}\right)$$

- k: position of an object in the input sequence ($0 \le k < L$)
- d: dimension of the output embedding
- P(k,j): position matrix
- n: user-defined scalar (e.g. set to 10,000)
- i: used for mapping to column indices $0 \le i < d/2$, with a single value of i maps to both sine and cosine functions

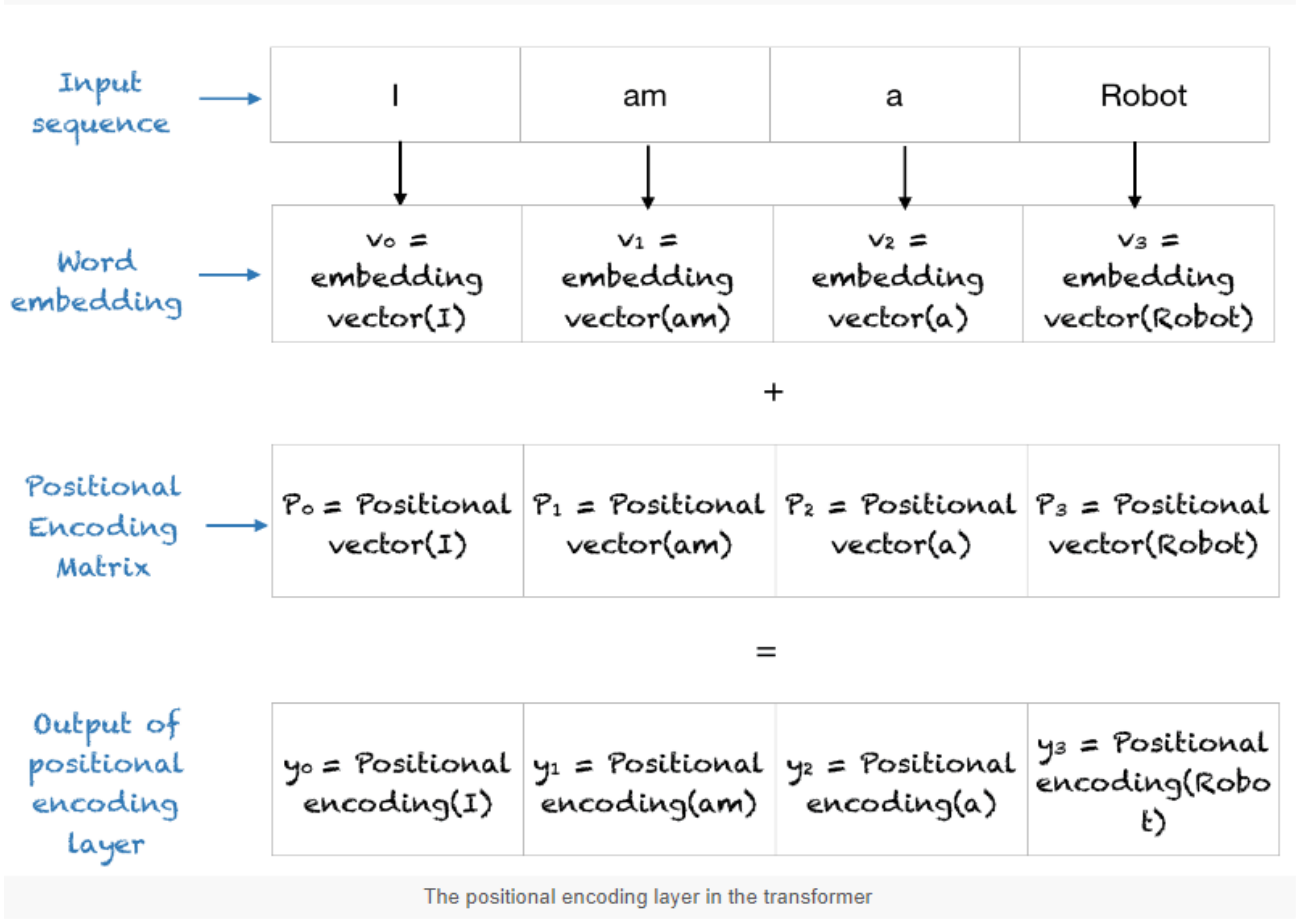


Positional Encoding Matrix for the sequence 'I am a robot'

https://machinelearningmastery.com/a-gentle-introduction-to-positional-encoding-in-transformer-models-part-1/

Positional Encoding

- Final output of the Positional Encoding Layer



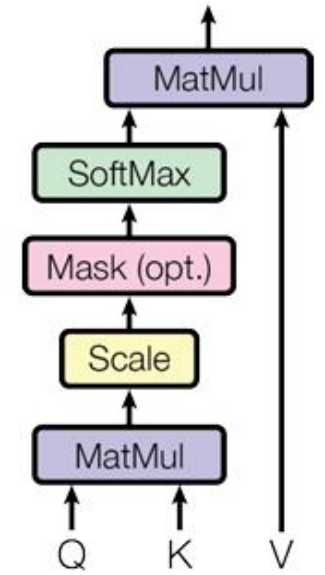
The positional encoding layer in the transformer

https://machinelearningmastery.com/a-gentle-introduction-to-positional-encoding-in-transformer-models-part-1/

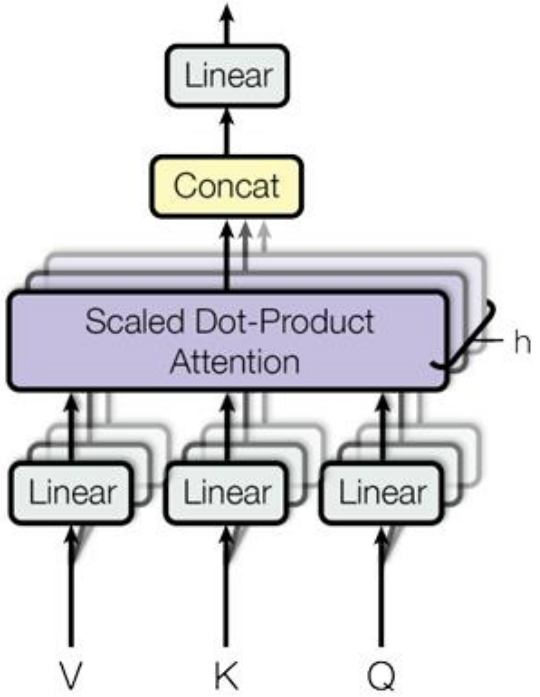
Multi-Head Attention

- **Attention**
  - Mapping a query and a set of key-value pairs to an output
  - The output is computed as a weighted sum of the values, where the weight is computed by a compatibility function of the query with the corresponding key
  - Query, key, and value are packed into matrices Q, K, and V respectively.
  - Usually, $d_k = d_q$, and $d_v = d_{model} = d_{emb}$

Scaled Dot-Product Attention



Multi-Head Attention



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

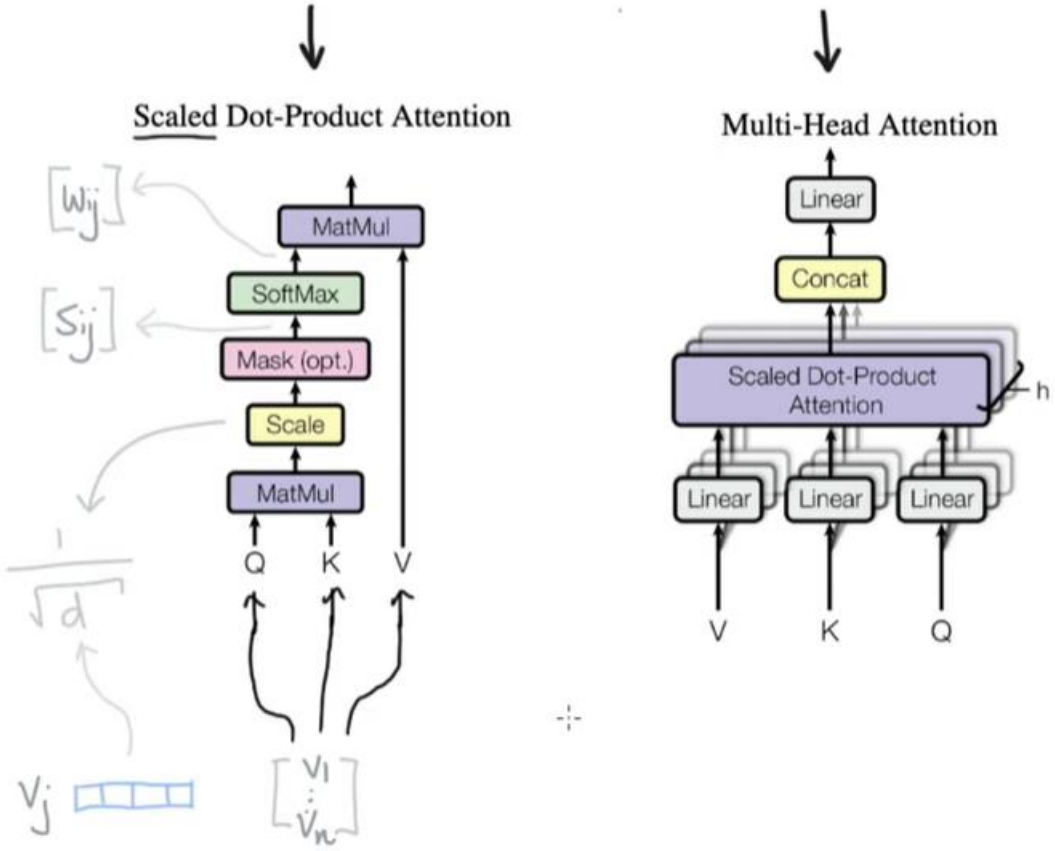
$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_\text{h})W^O$$
$$\text{where head}_\text{i} = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

*attention(q,**k**,**v**) = $\sum_i$ similarity(q,k_i) * v_i*

$$W_i^Q \in \mathbb{R}^{d_{\text{model}} \times d_k}, W_i^K \in \mathbb{R}^{d_{\text{model}} \times d_k}, W_i^V \in \mathbb{R}^{d_{\text{model}} \times d_v}$$

$$W^O \in \mathbb{R}^{hd_v \times d_{\text{model}}}$$

Multi-Head Attention



- The operation "Attention(Q, K, V)" in the context of the Transformer model means the process of assigning weights or importance scores to different elements in a sequence (or "values") based on their relevance to a specific element (or "query").

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

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_\text{h})W^O$$

$$\text{where head}_\text{i} = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

$$W_i^Q \in \mathbb{R}^{d_{\text{model}} \times d_k}, W_i^K \in \mathbb{R}^{d_{\text{model}} \times d_k}, W_i^V \in \mathbb{R}^{d_{\text{model}} \times d_v}$$

$$W^O \in \mathbb{R}^{hd_v \times d_{\text{model}}}$$

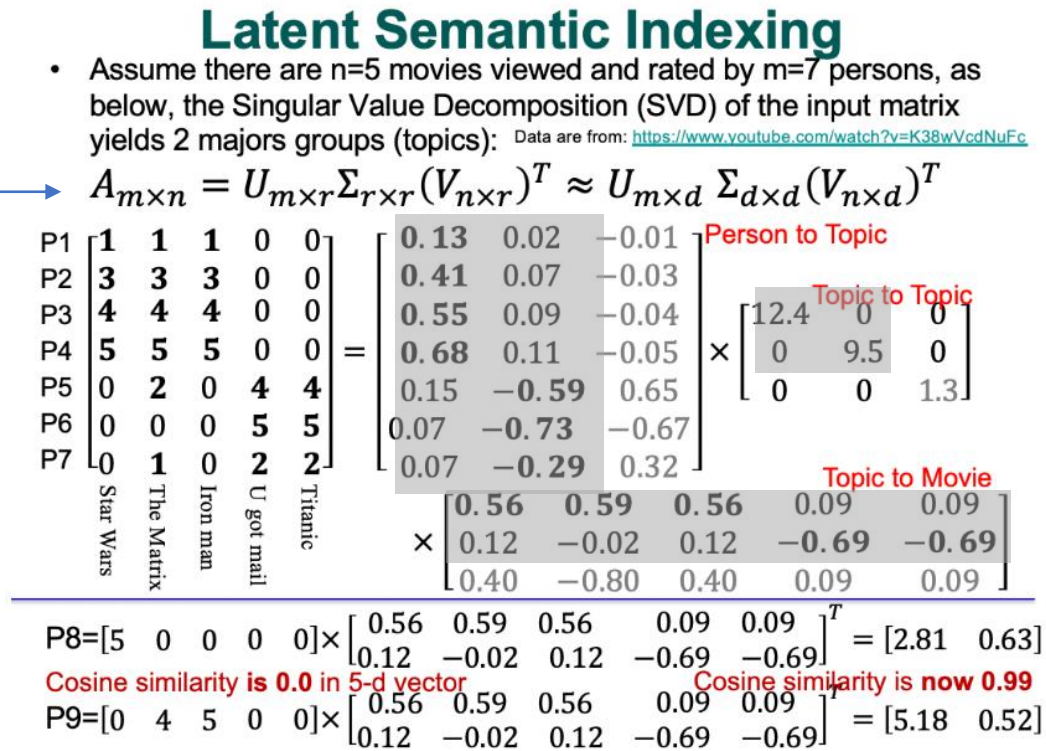
Linear Transformation (Q,K,V)

- Why **transformation** for Query, Key, Value?
  - we could still use the original encoder state vectors as the queries, keys, and values. So, **why we need the transformation**?
  - Simply a matrix multiplication: (I is the input (encoder) state vector)

    Query = I x W^Q , Key = I x W^K , Value = I x W^V
  - (1) If we do not transform input vectors, the dot product for computing the weight for each input's value will always yield a maximum weight score for the individual input token itself. (input *i* -번째 위치에서 *n* 개의 attention weight 를 계산하면 그 자신에서 가장 높은 값이 나옴)
    - For example, for the **pronoun** token, we need it to attend to its referent, not the pronoun token itself.
  - (2) The transformation **may yield better representations for Query, Key, and Value**. (see the next slide)
  - (3) conversion of the input vector into a space with a desired dimension (practically useful)

Linear Transformation (Q,K,V)

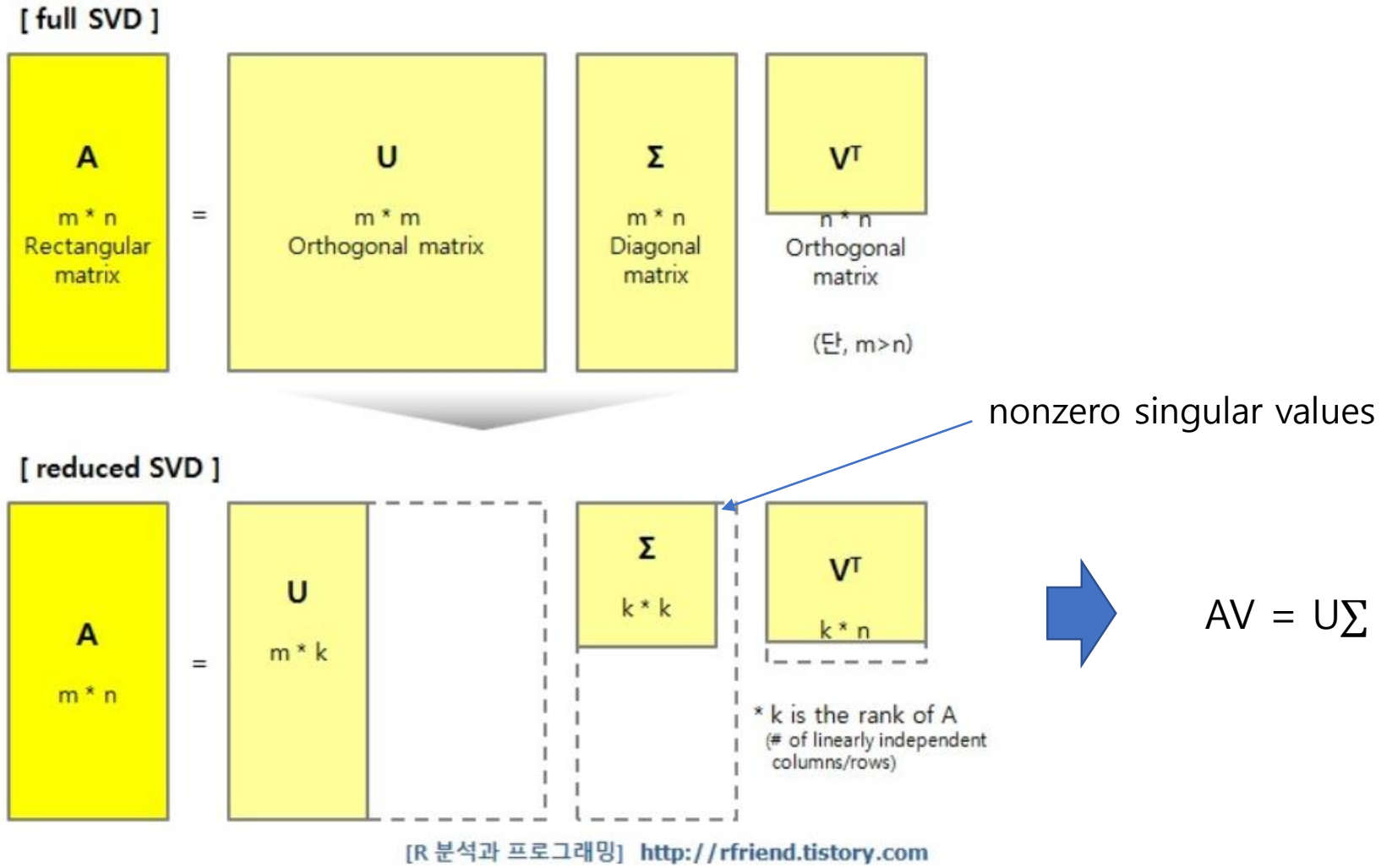
- Transformation may yield better representation for computing similarity between two vectors. (https://youtu.be/K38wVcdNuFc?t=10)



- By multiplying an input vector with a matrix V (from the SVD), we get a better representation for computing the compatibility between two vectors, if these two vectors are similar in the topic space as shown in the example in the figure.
- And, these matrices for transformation can be learned in a neural network!
- (before transformation) similarity(P8,P9)=0
- (after transformation) similarity(P8',P9') = 0.99

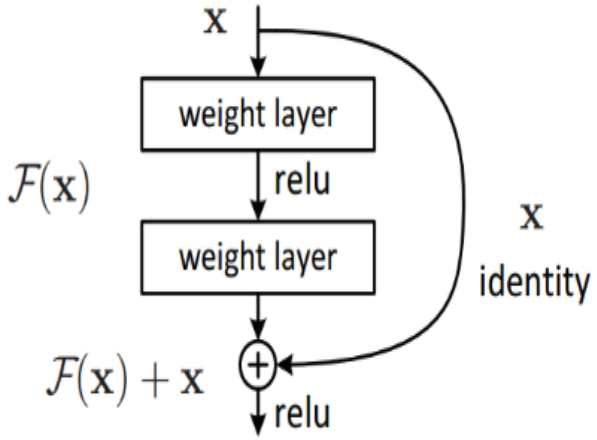
Reduced SVD

- Reduced SVD



Residual Neural Networks (ResNets)

- Why ResNets?
  - Addressing Vanishing Gradients: Skip connections allow gradients to flow directly to earlier layers, and the gradients have a shorter path to propagate, making it easier to update the weights of earlier layers and improve the model's overall performance.
  - Facilitating Training of Deeper Networks: By allowing information to flow through shortcut connections, ResNets enable the training of much deeper neural networks.
  - Promoting Feature Reuse: The skip connections in ResNets facilitate the direct flow of information from earlier layers to later layers. This allows the later layers to access the activations and features from earlier layers, promoting feature reuse.
  - Easier Optimization: provide a form of identity mapping that makes it easier for the network to optimize its weights. This can speed up the convergence during training and enable more efficient optimization.

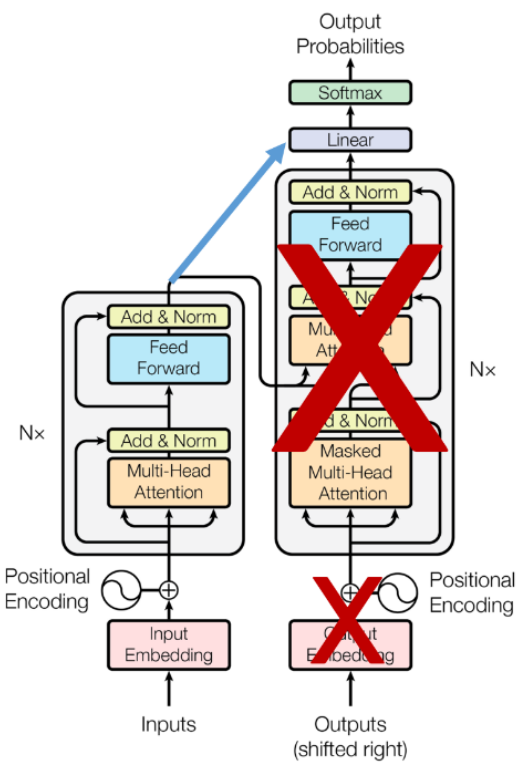


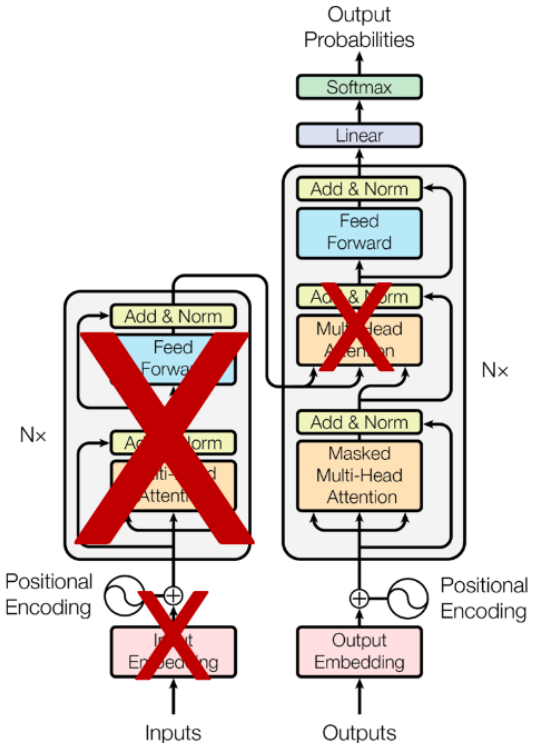
Transformer 응용: GPT and BERT

- GPT (Generative Pre-trained Transformer)
  - 구조: 디코더-온리 (Decoder-only) 아키텍처
  - 특징: 오직 Masked Self-Attention만 사용한다. 입력 프롬프트를 조건(Context)으로 삼아 텍스트를 순차적으로 생성(Generation)하는 데 최적화되어 있다.
  - 주요 목표: 언어 생성, 챗봇, 작문.
  - Context Vector의 의미: 입력 프롬프트와 지금까지 생성된 모든 토큰들 간의 관계를 압축한 시퀀스 기반의 문맥 정보.
- BERT(Bidirectional Encoder Representations from Transformers)
  - 구조: 인코더-온리 (Encoder-only) 아키텍처
  - 특징: 일반적인 Self-Attention을 사용하여 입력 문장 전체를 양방향으로 파악한다. 문장 내의 관계를 이해하고 분류/인식하는 데 강력하다.
  - 주요 목표: 텍스트 분류, 개체명 인식, 질의응답(NLU).
  - Context Vector의 의미: 입력 문장 전체의 의미를 모든 토큰에 분산하여 깊이 있게 이해한 결과.

Transformer 응용: GPT and BERT

- GPT:
  - Language model (predicts the next word)
  - Unidirectional
- BERT(Bidirectional Encoder Representations from Transformers)
  - Masked Language Model
  - Predicts the blanks in the middle of sentences (bidirectional)



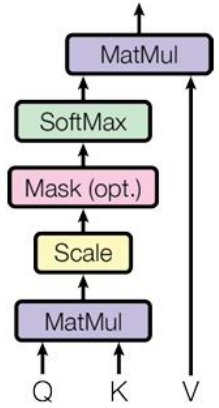


GPT
어제 카페 갔었어 거기 사람 많더라

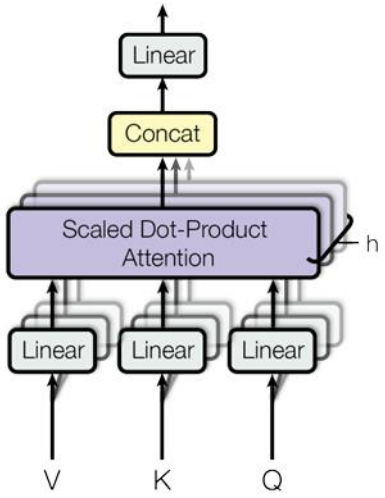
BERT
어제 카페 갔었어 ☐ 사람 많더라

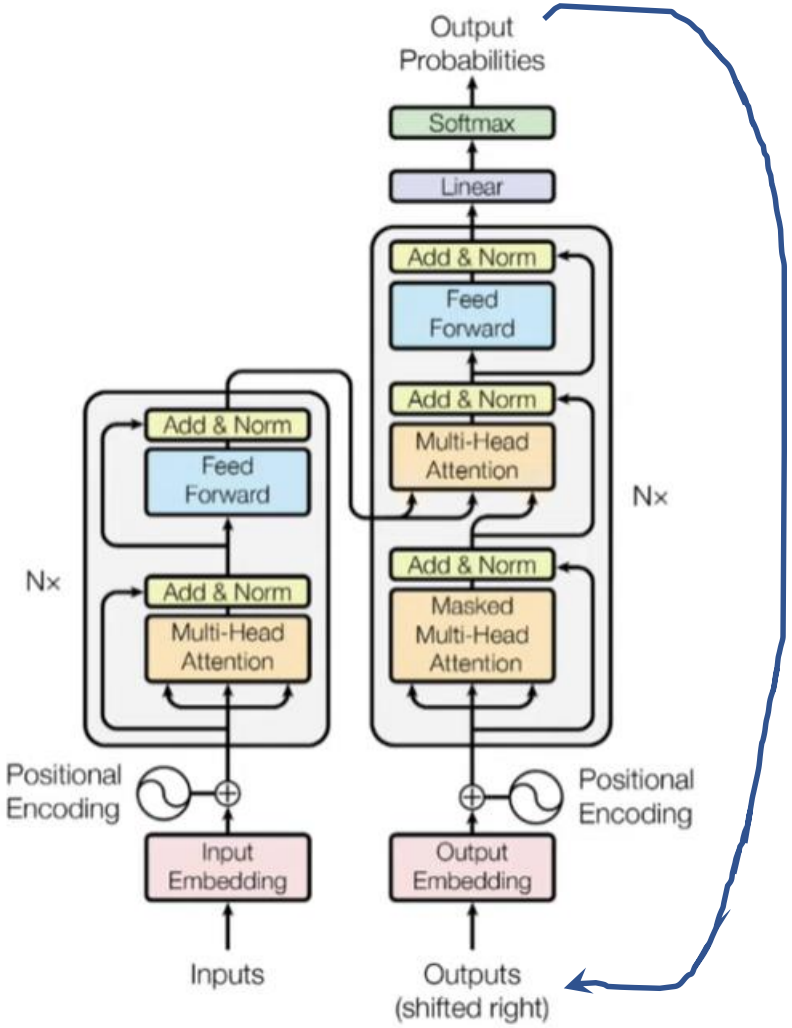
Data Dimensions in Transformer

Scaled Dot-Product Attention



Multi-Head Attention





sequence length = L
target sequence length = T
embedding dim = d_model
head = h
head dimension = d_k = d_v = d_model/h
FFN: 차원을 확장했다가 다시 줄이는 구조 (d_ff 는 hidden layer 차원)

- Encoder:
  Input: (L, d_model)
  Q,K,V: (L, d_model)
  Heads: (h, L, d_k)
  Attention scores: (h, L, L)
  Output: (L, d_model)
  FFN: (L, d_ff) → (L, d_model)

- Decoder:
  Input: (T, d_model)
  Masked Q,K,V: (T, d_model)
  Masked Heads: (h, T, d_k)
  Masked Att scores: (h, T, T)

  Cross-Q: (h, T, d_k)
  Cross-K/V: (h, L, d_k/d_v)
  Cross scores: (h, T, L)
  Cross output: (T, d_model)

- Final FFN: (T, d_ff) → (T, d_model)
- Logits: (T, vocab_size)

GPT – Transformer Decoder-Only

- ✔ GPT는 디코더만 존재
- ✔ 입력은 Prompt 전체
- ✔ Training에서는 문장 전체(T)를 한 번에 병렬로 계산
- ✔ Inference에서는 매 step마다 T가 커지고 전체를 다시 계산
- ✔ logits는 항상 (T, vocab)
- ✔ 생성 시에는 logits[-1]만 사용

항목	GPT Training	GPT Inference
입력	전체 문장(T)	Prompt(T_0) + 생성된 토큰
T의 의미	입력 문장 전체 길이	현재까지의 전체 문맥 길이
T 증가	증가하지 않음 (고정)	step마다 $T_0 \rightarrow T_0+1 \rightarrow T_0+2 \rightarrow \dots$
Self-attention	(T × T) 1회	(T × T) 매 step 재계산
계산 방식	**병렬**	**순차**
결과 logits	(T, vocab) 전체 예측	마지막 토큰 하나만 사용
Encoder	없음	없음

Transformer Decoder (GPT)

- **Transformer 디코더 단계별 문장 생성 과정 (GPT 모델)**

단계	디코더 입력	**Query** 생성	어텐션 계산	**Context Vector**	출력 단어
1. 시작	`<BOS>` (문장 시작 토큰)	$q_1 = e_{BOS} W^Q$	q_1과 모든 Key 비교 → 가중치 w_i	$c_1 = \sum_i w_i v_i$	"나는"
2. 두 번째 단어	"나는"	$q_2 = e_{나는} W^Q$	q_2와 모든 Key 비교 → 가중치 w_i	$c_2 = \sum_i w_i v_i$	"밥을"
3. 세 번째 단어	"나는 밥을"	$q_3 = e_{밥을} W^Q$	q_3와 모든 Key 비교 → 가중치 w_i	$c_3 = \sum_i w_i v_i$	"먹는다"
4. 종료	"나는 밥을 먹는다"	$q_4 = e_{먹는다} W^Q$	q_4와 모든 Key 비교 → 가중치 w_i	$c_4 = \sum_i w_i v_i$	`<EOS>` (문장 끝)

1. **입력(Prompt)**: 시작 토큰이나 이전 출력 단어가 디코더 입력으로 들어감
2. **Query 생성**: 입력 임베딩에 W^Q를 곱해 Query 벡터 생성
3. **어텐션 계산**: Query와 Key들의 유사도를 계산해 가중치 w_i 산출
4. **Context Vector**: Value들의 가중합으로 문맥 벡터 생성
5. **출력 단어 예측**: Context Vector를 기반으로 Softmax에서 다음 단어 선택

GPT 학습 파이프라인 (How GPT learns)

- **Stage 1: Pre-training (사전 학습) - 비지도 학습:**
  - Objective: Autoregressive Language Modeling (Next Token Prediction).
  - Loss Function: Cross-Entropy Loss.
  - 특징: 시퀀스 내의 **모든 토큰**에 대해, 이전 토큰들을 바탕으로 현재 토큰을 맞추는 방식으로 학습한다. 즉, $P(w_t \mid w_{1:t-1})$
  - Data Example (Continuous Prediction): 문장: "The cat sat on the mat"
    - Step 1: input: "The" → Target: "cat"
    - Step 2: Input: "The cat" → Target: "sat"
    - Step 3: Input: "The cat sat" → Target: "on"
    - … (문장 끝까지 반복)
    - 실제로는 한 번의 Forward Pass 에서 Masking을 통해 병렬적으로 Loss 를 계산함.

GPT 학습 파이프라인 (How GPT learns)

- **Stage 2: 지도 미세 조정 (SFT: Supervised Fine-Tuning) - 지도 학습:**
  - 데이터셋: 사람이 직접 작성하거나 검수한 고품질의 (Prompt, Response) 쌍.
  - Training 방식 (Loss Masking):
    - Training sequence: [Instruction], [input], [Response] 를 묶어 하나의 긴 시퀀스로 연결.
    - Stage 1 과 동일한 방식으로 다음 토큰 예측 수행 (Next Token Prediction). 다만, 가장 큰 차이점은 Loss Masking 이다.

```
{
  "instruction": "다음 문장을 3줄로 요약해줘.",
  "input": "(긴 뉴스 기사 텍스트...)",
  "output": "1. 사건의 개요는... 2. 주요 원인은... 3. 결론적으로..."
}
```

  - Loss Masking의 중요성
    - X (입력): 모델은 Instruction + Input 전체를 조건 (Context) 로 받는다.
    - y (정답 및 학습): Stage 1 과는 달리 이 긴 시퀀스 중에서 Response 부분에 해당하는 토큰들에서 발생한 예측 오차(Loss)만 계산하여 모델을 업데이트한다.
    - 이유: 사용자의 질문을 다시 생성하는 것은 목표가 아니며, 질문에 대한 '답변'을 생성하는 능력만 학습해야 하기 때문.

GPT 학습 파이프라인 (How GPT learns)

- **Stage 3: RLHF (Reinforcement Learning from Human Feedback) - 강화 학습:**
  - 핵심 개념: '인간 선호도'를 AI 코치로 변환하여 모델을 훈련한다
    - Stage 2까지는 지시를 잘 따르지만, 때로는 엉뚱하거나, 편향되거나, 위험한 답변을 할 수 있다.
    - RLHF는 인간이 선호하는 답변 에 주입하는 최종 단계이다.
  - Step 1: 보상 모델 훈련(Reward Model, RM)
    - 데이터: GPT가 생성한 여러 답변을 인간이 직접 순위(Ranking)를 매긴 데이터.
    - RM의 역할 (비유: 인간 코치): 인간의 선호도를 모방하도록 학습된 AI 평가 모델.
    - 결과: RM은 생성된 답변을 보고 보상점수(R)을 부여할 수 있게 된다.
    - 즉, Reward(보상)를 계산하는 환경(Environment)을 만든다.
  - Step 2: 정책 모델 훈련 (PPO, Proximal Policy Optimization) – 강화학습 적용
    - 상태 (State): 사용자 프롬프트
    - 행동 (Action): 정책모델(SFT-GPT)이 생성한 답변
    - 보상 (Reward): RM 이 부여한 보상 점수
    - 학습 목표: 정책 모델(Agent)은 RM이 부여하는 보상 점수 R을 최대화하는 방향으로 자신의 답변 생성 정책 업데이트
    - KL Divergence Penalty (안정성 확보): 업데이트 시, Stage 2의 SFT 모델과 너무 멀리 벗어나지 않도록 KL 페널티를 적용하여 학습의 안정성을 확보한다.

Prompt Engineering

- **프롬프트(Prompt)란?**: AI에게 내리는 지시어, 명령어.
- **좋은 프롬프트의 3원칙**:
  - 구체성 (Specific): "글 써줘" (X) -> "신제품 마케팅을 위한 500자 내외의 블로그 글을 써줘" (O)
  - 맥락 (Context): "너는 10년 차 마케터야", "독자는 20대 대학생이야".
  - 형식 (Format): "표로 정리해줘", "개조식으로 써줘".
- **핵심 기법 (The Techniques)**
  - 페르소나 부여 (Persona): "너는 지금부터 베테랑 영어 교사야."
  - 퓨샷 러닝 (Few-shot Learning): 예시를 몇 개 보여주고 시키기.
    - 입력: 사과 -> 빨강, 바나나 -> 노랑, 포도 -> ?
  - CoT (Chain of Thought): "단계별로 생각해서 답해줘(Think step by step)." -> 논리적 추론 능력 향상.
  - 구분자 사용: 텍스트를 """ 또는 --- 로 감싸서 지시문과 자료를 분리하기.

Prompt Engineering

- **프롬프트 엔지니어링 (Prompt Engineering)**
  - 개념: LLM이 최적의 성능을 내도록 질문(프롬프트)을 설계하는 기법.
  - 핵심 기법:
    - CoT (Chain-of-Thought): 사고 과정을 명시하도록 지시하여 복잡한 추론 능력 향상.
    - Few-Shot Learning: 몇 가지 예시를 제공하여 원하는 응답 포맷이나 스타일을 안내.
- **추론 파라미터 제어**
  - 모델의 출력 품질과 다양성을 제어하는 핵심 파라미터.
  - Temperature (τ):
    - 의미: 출력의 무작위성(Randomness) 제어.
    - 값: 0.0 (가장 결정적/보수적) ~ 1.0 이상 (가장 창의적/무작위)
    - 활용: 논문 요약 등 정확성이 중요할 때는 낮게, 소설 창작 등 창의성이 중요할 때는 높게 설정.
  - Top-p (Nucleus Sampling)
    - 의미: 모델이 다음 단어를 고를 때 전체 단어 중에서 어느 수준까지 남길지 정하는 확률 필터.
    - 확률이 높은 상위 토큰들(p) 의 누적 확률 범위 내에서만 다음 토큰을 샘플링하도록 제한.
    - 활용: 응답의 품질을 유지하면서 어느 정도의 다양성을 확보할 때 사용된다.

추론 파라미터 제어 사용 예

- 대화형(ChatGPT 웹/앱) 사용 예:
  - temperature=0.2 (보수적, 논리적 스타일)

```
너는 매우 논리적이고 정확하며 사실 중심적으로 대답하는 어시스턴트이다.
창의적인 답변을 하지 말고, 근거 기반으로 단정적인 설명을 해줘.
질문: 하늘이 왜 파란지 설명해줘.
```

  - temperature=1.0 (창의적, 자유로운 스타일)

```
답변을 매우 자유롭고 창의적으로 해줘.
비유나 새로운 관점을 사용해도 좋아.
질문: 하늘이 왜 파란지 설명해줘.
```

- Python API 사용 예:

```python
from openai import OpenAI

client = OpenAI()

response = client.chat.completions.create(
    model="gpt-4.1",
    messages=[
        {"role": "system", "content": "You are a logical and concise assistant."},
        {"role": "user", "content": "Explain why the sky is blue."}
    ],
    temperature=0.2,      # 자유도
    top_p=1.0,            # 단어 선택 범위의 넓이를 조절 (1.0 = 필터링 없음)
    max_tokens=200,
)

print(response.choices[0].message["content"])
```