Large Language Model

1.Định nghĩa

LLM (Large Language Model) là một loại mô hình trí tuệ nhân tạo (AI) được thiết kế để xử lý và tạo ngôn ngữ tự nhiên. Chúng được huấn luyện trên lượng dữ liệu văn bản khổng lồ, cho phép chúng hiểu và tạo văn bản giống như con người.

Ví dụ về LLM nổi tiếng

- **GPT (Generative Pre-trained Transformer)** – do OpenAI phát triển (như GPT-3.5, GPT-4, GPT-4.5)
- **PaLM** – do Google phát triển
- **LLaMA** – do Meta phát triển
- **Claude** – do Anthropic phát triển
- **Gemini** – Google DeepMind
- **Mistral**, **Cohere**, v.v.

2.Cách hoạt động của LLM (Kiến trúc Transformer)

Transformer là kiến trúc nền tảng của hầu hết các mô hình LLM hiện đại ngày nay như GPT,BERT,.. sử dụng cơ chế Self-Attention để xử lí dữ liệu tuần tự hiệu quả hơn.

Transformer gồm 2 phần chính:

- Encoder: Xử lí đầu vào.
- Decoder: Tạo đầu ra

Quy trình hoạt động :

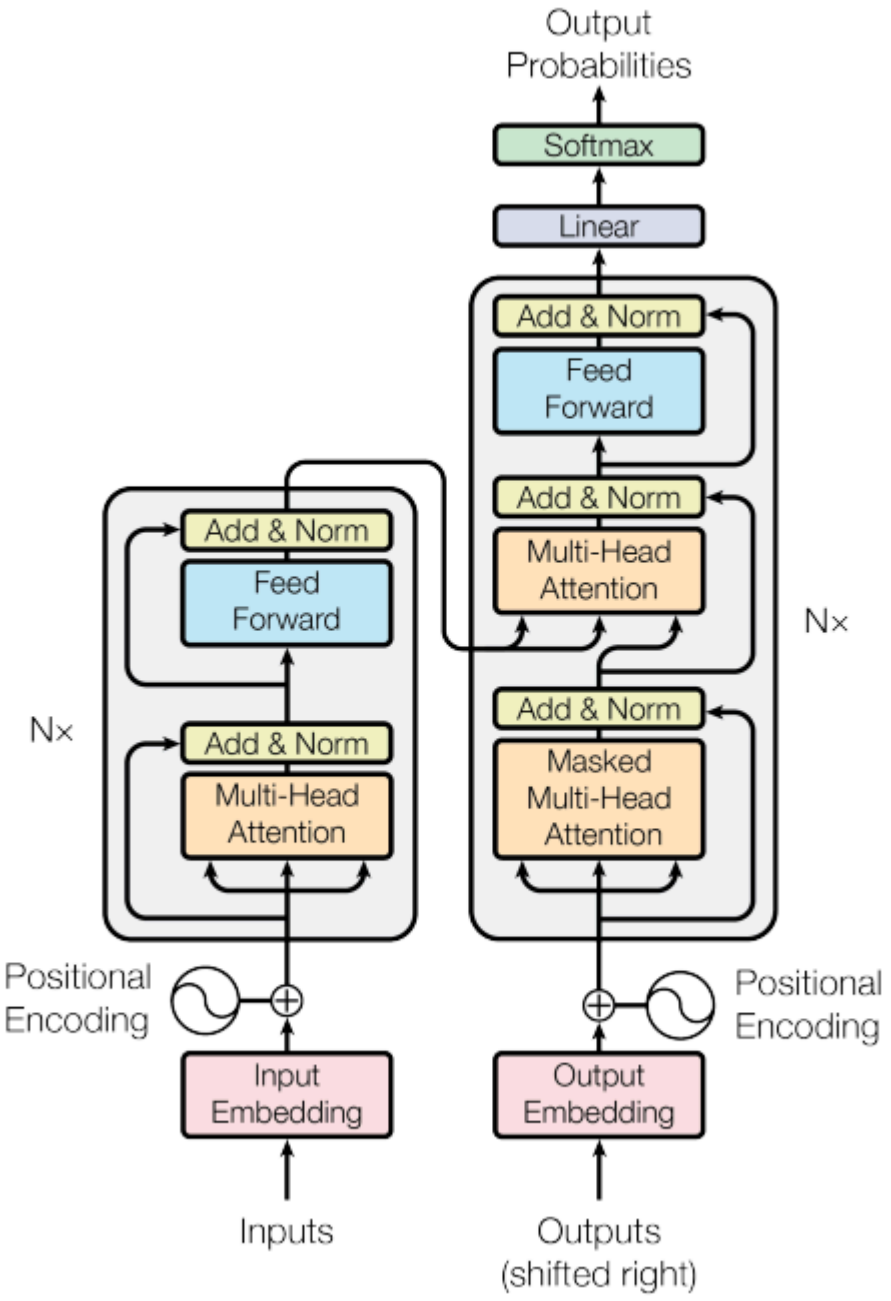
Input Embedding: Token hóa và chuyển thành embedding + positional encoding

Encoder:

- Self-attention → FFNN → Lặp lại N lần
- Đầu ra là biểu diễn ngữ cảnh của chuỗi đầu vào

Decoder:

- Dùng masked self-attention để xử lí từng token đầu ra.
- Cross-attention kết hợp thông tin từ encoder
- FFNN→Lặp lại N lần→Dự đoán token tiếp theo



2.1 Embedding

2.1.1 Định nghĩa

Embedding đóng vai trò quan trọng trong việc chuyển đổi văn bản đầu vào thành các vector số học (dãy các con số) mà máy tính có thể hiểu và xử lý được. Lớp này giúp mô hình nắm bắt được ngữ nghĩa và cú pháp của văn bản, từ đó hiểu rõ hơn về ngữ cảnh mà văn bản đó xuất hiện.

2.1.2 Cách Embedding hoạt động

1.Tokenization:Tokenization là quá trình chia văn bản thô thành các đơn vị nhỏ hơn gọi là tokens — những phần tử cơ bản mà mô hình ngôn ngữ có thể hiểu và xử lý.Có một số phương pháp tokenization nổi tiếng như :**WordPiece**, **BPE**, hoặc **SentencePiece.**

Ví dụ minh họa :

Câu: "I love playing soccer!"
Tokenization có thể ra: ["I", "love", "play", "##ing", "soccer", "!"]

Giải thích:

Dấu **##** xuất hiện khi dùng kiểu tokenizer gọi là **WordPiece** (Google/BERT) hoặc **BPE** (GPT/Bloom, vv.).Nó dùng để biểu thị rằng:Token có dấu ## là phần tiếp theo của một từ, không thể đứng một mình.Ở ví dụ trên, Tokenizer có thể không lưu "playing" như một token hoàn chỉnh trong từ điển, nhưng lại có:

- "play" (token riêng có thể đứng độc lập 1 mình)
- "ing" (nhưng không cho phép bắt đầu từ bằng "ing", vì không thường xuyên)

Lợi ích của việc sử dụng ##:

Lợi ích	Giải thích
Xử lý từ mới hoặc từ hiếm	Không cần lưu mọi từ như "disappointment"; chỉ cần "dis", "##appoint", "##ment"
Tiết kiệm không gian từ điển	Giảm số lượng từ cần lưu trữ xuống hàng chục nghìn thay vì hàng triệu
Tái sử dụng token phổ biến	"##ing", "##ly", "##ed"... xuất hiện ở cuối nhiều từ → rất hiệu quả
Cho mô hình hiểu cấu trúc từ vựng	Biết "play" và "playing" liên quan đến nhau → tăng khả năng tổng quát

2.Embedding: Mỗi token được ánh xạ thành vector, ví dụ: ["I"] → [0.21, -0.36, ..., 0.11] (vector 768 chiều)

Giải thích:

-Về giá trị của vector:

- **Khởi tạo ban đầu :** Khi bắt đầu huấn luyện, các giá trị trong embedding vector thường được khởi tạo ngẫu nhiên hoặc bằng các phương pháp khởi tạo chuẩn (ví dụ: Xavier, He initialization).Mục đích để mô hình có điểm bắt đầu, không phải vector nào cũng giống nhau ngay từ đầu.

- **Trong quá trình huấn luyện:** Các giá trị này sẽ được tối ưu dần dựa trên: mục tiêu mô hình (ví dụ: dự đoán từ tiếp theo) và cách token đó xuất hiện trong ngữ cảnh.Nói cách khác, vector embedding thay đổi liên tục sao cho: các token có ngữ nghĩa giống nhau sẽ có vector gần nhau (ví dụ: “cat” và “dog”) và các token khác nghĩa sẽ có vector xa nhau hơn.

-Về size của vector: Việc lựa chọn embedding size phụ thuộc vào nhiều yếu tố như quy mô mô hình, độ phức tạp của tác vụ,kích thước và độ đa dạng của dữ liệu huấn luyện,...

Embedding Size	Ưu điểm	Nhược điểm
Nhỏ (128–512)	Nhanh, nhẹ, phù hợp mô hình nhỏ	Hạn chế về khả năng biểu diễn ngữ nghĩa
Trung bình (768–2048)	Cân bằng hiệu năng và chi phí	Được dùng phổ biến trong BERT/GPT cơ bản
Lớn (4096–12000+)	Biểu diễn cực mạnh, ngữ nghĩa sâu	Rất nặng, tốn tài nguyên, cần dữ liệu lớn

3.Positional Embedding: Vector của mỗi token được cộng với vector vị trí tương ứng**.**Trong đó, vector vị trí là vector mã hóa thông tin vị trí của token trong chuỗi dùng để bổ sung cho mô hình khả năng hiểu thứ tự từ trong câu. Nếu không thêm thông tin vị trí, mô hình sẽ xem các câu như "I love you" sẽ giống như "you love I".

Cách tạo vector vị trí : Có 2 loại Positional Embedding chính :

Loại	Cách tạo vector vị trí	Ưu điểm	Nhược điểm
Sinusoidal (Fixed)	Dùng sin/cos theo công thức toán	Tổng quát tốt, không cần học	Giới hạn biểu diễn phức tạp
Learned (Trainable) (Phổ biến hơn)	Vector cho mỗi vị trí, học từ dữ liệu	Biểu diễn linh hoạt, dễ dùng	Không khái quát ra vị trí chưa thấy

Sau khi token hóa và embedding, mỗi token có một vector chiều cao (ví dụ 768 chiều), ta có một tensor dạng:

input_embeddings = [[x11, x12, ..., x1d],
[x21, x22, ..., x2d],
...
[xn1, xn2, ..., xnd]]

Trong đó:

- n là **sequence length** (độ dài chuỗi tokens)
- d là **embedding size** (chiều cao của vector, ví dụ 768 hoặc 1024)

Tensor này có kích thước: **[n, d]**

4.Đưa vào mô hình (LLM) : Sau khi văn bản được chuyển thành một chuỗi vector (embedding) bước tiếp theo là đưa embedding này vào mô hình LLM.
Để đưa vào mô hình chúng ta cần thêm chiều batch vì trong mô hình deep learning, dữ liệu đầu vào thường được xử lý theo batch — tức là xử lý nhiều chuỗi văn bản cùng một lúc.Bây giờ tensor sẽ có dạng :

[batch_size, n, d]

2.2 Attention

2.2.1 Định nghĩa

Lớp attention là yếu tố quyết định giúp mô hình tập trung vào những phần quan trọng trong văn bản đầu vào. Trong đó, attention layer cho phép mô hình đánh giá và xác định phần nào của văn bản là quan trọng nhất đối với từng nhiệm vụ cụ thể, qua đó tạo ra đầu ra chính xác và phù hợp nhất.

Ví dụ:

- Câu: "Tôi thích táo vì chúng ngọt."
- Khi xử lý từ "chúng", mô hình cần biết nó tham chiếu đến "táo" → Attention giúp liên kết 2 từ này.

Có nhiều biến thể của attention nhưng phổ biến nhất là : Self-attention và Cross-attention.

2.2.2 Cách Self-Attention hoạt động

Input: Tensor 3 chiều với kích thước:

[batch_size, n, d]

Công thức tổng quát:

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

Giải thích:

1.Tạo các vector Q,K,V

Với mỗi token trong chuỗi, ta tạo ra 3 vector:

- **Q (Query)**: đại diện cho token đang được xét, thể hiện "câu hỏi".
- **K (Key)**: đại diện cho các token khác, thể hiện "khóa để đối chiếu".
- **V (Value)**: thông tin thực tế sẽ được kết hợp lại để tạo đầu ra.

Thực hiện bằng cách nhân embedding với các ma trận học được:

Q = X · W_Q

K = X · W_K

V = X · W_V

Trong đó:

- **X**: tensor input ban đầu [batch_size, n, d]
- **W_Q, W_K, W_V**: ma trận trọng số riêng biệt (kích thước [d, d_k]) (d_k <= d)
- Kết quả **Q, K, V** có kích thước: [batch_size, n, d_k]

Giải thích các ma trận W_Q, W_K, W_V: Là ma trận trọng số học được (learnable weights) trong quá trình huấn luyện mô hình.Chúng không được tính theo công thức cố định, mà được khởi tạo ban đầu và tối ưu bằng thuật toán học như Adam hoặc SGD.

Mục đích chính của việc tạo Q, K, V là:

1. Phân biệt vai trò của từng từ (hỏi/được hỏi/thông tin).
2. Chuẩn bị dữ liệu để tính toán mức độ tương quan (Attention Scores).
3. Hỗ trợ Multi-Head Attention giúp mô hình học đa dạng ngữ nghĩa.

Nhờ bước này, mô hình có thể hiểu được các mối quan hệ phức tạp trong câu, như đại từ thay thế, sự phụ thuộc xa, hay ngữ cảnh đa nghĩa.

Ví dụ:

Câu: “Chó đuổi mèo vì nó hung dữ”

Từ “nó” được biến đổi thành:

- **Q(nó)** : Vector hỏi “từ nào đang được thay thế?”.
- **K(chó),K(mèo)**:Vector để so sánh với **Q(nó)**.
- **V(chó), V(mèo)** : Thông tin thực tế về “chó”, “mèo”.

Kết quả : Q(nó) * K(chó) > Q(nó)*K(mèo) → Attention sẽ tập trung vào “chó” (vì “chó” hung dữ)

2.Tính Attention Scores

Tính độ tương quan giữa Query của từ hiện tại và Key của tất cả các từ khác (kể cả chính nó) bằng tích vô hướng:

$$\text{Score}(Q_i, K_j) = Q_i \cdot K_j^T$$

Giả sử ta có câu “Tôi thích NLP” cho ra các token [“Tôi” , “thích”, “NLP”]

Khi ta xử lí từ “thích”, ta tính :

- Score(“thích”,”Tôi”)
- Score(“thích”, “thích”)
- Score(“thích”,”NLP”)

3.Chuẩn hóa bằng Softmax

- Chia các scores cho sqrt(dk) (với dk là kích thước của vector Key) để tránh giá trị quá lớn.

- Áp dụng Softmax để chuẩn hóa thánh xác suất có tổng = 1

$$\text{Attention Weights} = \text{Softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)$$

- Kết quả: Từ nào quan trọng sẽ có trọng số cao (gần 1), từ không liên quan ≈ 0.

4.Tính Weighted Sum của Value

Nhân trọng số Attention với Value tương ứng và cộng lại :

$$\text{Output} = \sum(\text{Attention Weights} \times V)$$

Giải thích : Phương pháp Weighted Sum không chỉ đơn thuần là tính tổng toán học, mà là cách để mô hình ghi nhớ thông tin các từ liên quan và tạo ra một biểu diễn mới kết hợp hài hòa giữa các thông tin quan trọng.

Ví dụ:

Câu [“Tôi”,”thích”,”NLP”] (d = 3)

Value ban đầu :

- V_Tôi = [0.1, 0.2, 0.3]
- V_thích = [0.4, 0.5, 0.6]
- V_NLP = [0.7, 0.8, 0.9]

Attention Weights cho từ "thích":

"Tôi" → 0.2, "thích" → 0.3, "NLP" → 0.5

Output của "thích":

$0.2 \cdot [0.1,0.2,0.3]+0.3 \cdot [0.4,0.5,0.6]+0.5 \cdot [0.7,0.8,0.9]=[0.49,0.59,0.69]$

→ Vector mới của "thích" là [0.49, 0.59, 0.69]

Output :Tensor 3 chiều với kích thước: **[batch_size, n, d]** với các giá trị đã thay đổi qua lớp attention.

2.3 FFNN (Feed Forward Neural Network)

2.3.1 Định nghĩa

FFNN là một thành phần quan trọng trong cấu trúc Transformer đóng vai trò biến đổi thông tin đã được xử lí bởi cơ chế Self-Attention thành biểu diễn mới, giúp mô hình học các đặc trưng phi tuyến phức tạp hơn.

2.3.2 Tác dụng

- **Biến đổi thông tin cục bộ:**
  Trong khi Self-Attention tập trung vào các mối quan hệ giữa các token thì FFNN xử lí thông tin từng token độc lập.
  Ví dụ:
  Sau khi Attention biết “apple” liên quan đến “eat”, FFNN mã hóa riêng nghĩa của “apple” (trái cây) và “apple” (công ty).
- **Tăng khả năng biểu diễn:**
  FFNN tăng khả năng thêm phi tuyến tính vào mô hình , giúp các hàm phức tạp hơn so với chỉ dùng Attention (giúp cho mô hình hiểu nghĩa của từ sâu xa hơn)
- **Ổn định quá trình huấn luyện**

2.3.3 Cấu trúc

FFNN trong Transformer gồm 2 lớp tuyến tính (Linear Layers) với một hàm kích hoạt phi tuyến ở giữa

$$\mathrm{FFNN}(x) = W_2 \cdot \mathrm{ReLU}(W_1 \cdot x + b_1) + b_2$$

Các thành phần chính:

Lớp tuyến tính thứ nhất (W_1)

Mở rộng chiều dữ liệu (Vd : từ 512 thường lên khoảng gấp 4 lần là 2048) với mục đích để tăng khả năng biểu diễn , giúp cho mô hình có nhiều không gian hơn để học các đặc trưng phức tạp và áp dụng ReLU hiệu quả vì ReLU hoạt động tốt hơn trong không gian nhiều chiều.

Hàm kích hoạt ReLU:

$$ReLU(x) = max(0,x)$$

Dùng ReLU trong FFNN để loại bỏ các giá trị âm , chỉ giữ lại các thông tin quan trọng, phù hợp với ngữ cảnh.

Lớp tuyến tính thứ hai (W_2)

Thu nhỏ chiều dữ liệu về như cũ (Vd : từ 2048 về lại 512) để đảm bảo đầu ra có cùng kích thước với đầu vào để cộng residual (nhằm tránh mất mát thông tin)

2.3.4 Ví dụ minh họa

Input: Vector embedding của token “cat” sau Self-attention

[0.2, -0.5, 0.7, ..., 0.1] (shape: [512]).

Quá trình FFNN:

Mở rộng lên [2048] qua **W_1:**

$$[0.2,-0.5,0.7,...] * W1 = [1.1,0.0,-0.3,...,2.4] \text{ (shape : [2048])}$$

Áp dụng ReLU:

[1.1,0.0,0.0,...,2.4](Các giá trị âm thành 0)

Thu nhỏ về [512] qua **W_2:**

$$[1.1,0.0,0.0,...,2.4] * W2 = [0.3,-0.1,0.8,...,0.2]$$

Output: Vector “cat” đã được biến đổi

2.4 Prompt → OutPut

2.4.1 Định nghĩa

1.Prompt

Prompt là thông tin đầu vào mà người dùng cung cấp cho LLM , có thể dưới dạng câu hỏi , câu lệnh, một văn bản,...

Prompt có thể chứa ngữ cảnh hướng dẫn hoặc ví dụ để điều chỉnh kết quả.

Prompt cũng được xử lí qua các bước embedding, attention.

2.Output

Output là phản hồi do LLM tạo ra dựa trên prompt.

3.Mối quan hệ giữa Prompt và Output

LLM xử lý prompt bằng cách:

- Phân tích cú pháp và ngữ nghĩa của đầu vào.
- Dự đoán từ/cụm từ tiếp theo dựa trên dữ liệu đã được huấn luyện.
- Tạo ra văn bản đầu ra phù hợp với ngữ cảnh được yêu cầu.

Chất lượng output phụ thuộc vào:
- Độ rõ ràng của prompt
- Khả năng hiểu ngữ cảnh của mô hình
- Quy mô và chất lượng dữ liệu huấn luyện

2.4.2 Quá trình dự đoán token tiếp theo

1.Vocabulary
Để dự đoán được tokens tiếp theo trước tiên chúng ta cần phải xây dựng một vocabulary cho mô hình.

Kích thước vocab(V) : thường từ 50000 - 200000+ token tùy mô hình (GPT-3 : 50000+ token , BERT : 30000 token)

Trong đó bao gồm:
- Từ hoàn chỉnh (vd : “chó”, “mèo”).
- Subword(vd: “##ing”,”##ly”).
- Token đặc biệt (vd : <EOS> = End Of Sequence)

Cách tạo ra một vocab:
1) Chuẩn bị dữ liệu
2) Tokenization (Tách từ thành các token)
3) Xây dựng Vocab (Với Subword Tokenization (BPE/WordPiece))
    a) Thống kê tần suất xuất hiện của từ/subword trong dữ liệu
    b) Ghép cặp các ký tự/subword thường đi cùng nhau (“Hà” + “Nội” thành “Hà_Nội”)
    Từ được ghép sẽ là một token mới được thêm vào vocab và những từ ghép sẽ được giữ nguyên trừ khi đạt tới giới hạn của vocab và tần suất xuất hiện không cao thì sẽ bị xóa đi.
    c) Lặp lại cho đến khi đạt kích thước vocab mong muốn

2.Quy trình tính xác suất

1. **Lấy embedding token cuối của prompt:** Vì mô hình dự đoán token kế tiếp dựa trên toàn bộ những token trước đó nên chỉ cần đầu ra của token cuối cùng là đủ.

2. **Linear Projection:** nhân embedding với ma trận Woutput (trong đó mỗi cột là một embedding tượng trưng cho mỗi token trong vocab).

3. **Softmax:** Chuyển hóa vector kết quả ở trên thành xác suất

4. **Chọn token :** Có nhiều cách chọn token như greedy decoding, sampling,.. phổ biến nhất là Top-p và Temperature.

3.Quy trình đào tạo LLM

Các quá trình huấn luyện (từ pretraining đến fine-tuning) chính là việc tối ưu hóa các giá trị của vector mà được khởi tạo ngẫu nhiên ban đầu trong mọi lớp của LLM (từ embedding , attention, FFN,...) bằng cách :

- Tính loss để đo lỗi dự đoán mô hình.
- Dùng gradient để xác định hướng điều chỉnh weights sao cho loss giảm
- Cập nhật giá trị (weights) theo hướng gradient để tiến dần đến điểm tối ưu

Tính loss : Tính toán mức độ dự đoán sai của mô hình so với dữ liệu thực tế.

$$\text{Loss} = -\sum_i y_i \log(p_i)$$

Trong đó :
$\mathbf{y}_i$: Nhãn thực tế (One-hot encoding)
$\mathbf{p}_i$:Xác suất dự đoán của mô hình

Gradient: là đạo hàm riêng của loss theo từng weight (được khởi tạo ngẫu nhiên lúc ban đầu), cho biết loss thay đổi thế nào nếu weight thay đổi.
Mục đích của gradient là lặp đi lặp lại để tìm weight mà loss bé nhất.
Quá trình này dừng lại khi loss không cải thiện sau một số steps nhất định (Ví dụ: 1 triệu steps đối với quá trình huấn luyện GPT-3)
Ví dụ minh họa:

- Giả sử weight w trong attention layer có gradient âm (- 0,3):
- Điều này nghĩa là tăng w sẽ giảm loss
- Optimizer sẽ cập nhật w = w + α*0,3 (α là learning rate)

Optimization: dùng gradient để cập nhật theo công thức

$$w_{t+1} = w_t - \alpha \cdot \mathrm{AdamW}(g_t)$$

g_t: Gradient tại bước t
α: Learning rate(Vd : 10^(-4))
AdamW : tính momentum và adaptive learning rate để tối ưu hiệu quả.

3.1 Pretraining

Định nghĩa: Pretraining là cách đào tạo cơ bản nhất. Phương pháp này được sử dụng với một mô hình chưa được đào tạo (tức là mô hình được khởi tạo ngẫu nhiên) và huấn luyện để nó có thể dự đoán token tiếp theo dựa trên một chuỗi các tokens trước đó.

Nguyên lí cơ bản:
Pretraning dựa trên ý tưởng : “Dự đoán phần còn thiếu của văn bản” để học biểu diễn ngôn ngữ.

Input: Một chuỗi tokens

Output: Dự đoán tokens tiếp theo (GPT) hoặc điền từ bị che (BERT)

Phương pháp:

- Tự hồi quy (Autoregressive - GPT):

    - Mô hình dự đoán lần lượt từng token dựa trên ngữ cảnh trái.
    - Ví dụ: “Hôm nay trời ” -> “đẹp”
- Masked Language Modeling (BERT)
    - Che 15% từ trong câu -> Mô hình điền từ bị thiếu
    - Ví dụ: “Hôm nay [MASK] đẹp” -> “trời”

Các đặc điểm quan trọng:
Dữ liệu : Không cần gán nhãn (self-supervised), lượng dữ liệu cực lớn (Vd: GPT-3 dùng 500 tỷ token)

Tối ưu hóa:
- Dùng AdamW với learning rate ~1e-4.
- Batch size lớn (hàng nghìn token)

Kết quả: Mô hình học được:
- Biểu diễn từ (embeddings).
- Ngữ pháp, logic cơ bản
- Kiến thức tổng quát

Tác động của pretraining đến các lớp của kiến trúc Transformer

- Embedding Layer:
    - Tối ưu token embedding để phản ánh nghĩa của từ vựng
    - Điều chỉnh positional embedding

- Self-attention:
    - Học cách tính Q,K,V để nắm bắt quan hệ giữa các từ
    - Ma trận W_Q,W_K,W_V được khởi tạo và tối ưu

- FFNN:
    - Biến đổi đặc trưng phi tuyến để mã hóa ngữ nghĩa sâu hơn

3.2 Fine-Tuning

Định nghĩa: Fine-tuning điều chỉnh mô hình đã pretraining cho các tác vụ cụ thể (dịch máy, chatbot, phân loại văn bản,...)

Nguyên lí hoạt động :

Input: Dữ liệu có nhãn

Output: Tối ưu hóa đầu ra cho tác vụ mục tiêu

Phương pháp:

- Supervised Fine-Tuning (SFT):
  - Huấn luyện trên cặp input-output
  - Ví dụ: Dịch ‘’hello’’ sang tiếng Việt -> “Xin chào”

- Reinforcement Learning from Human Feedback (RLHF)
  - Tối ưu hóa phản ứng đánh giá của con người (dùng trong ChatGPT)

Các đặc điểm quan trọng:

- **Dữ liệu:**
  - Có nhãn, lượng nhỏ hơn pretraining (Vài trăm nghìn mẫu)

- **Tối ưu hóa:**
  - Learning rate nhỏ hơn pretraning (~1e-5)
  - Tránh catastrophic forgetting bằng kĩ thuật như LoRA

- **Kết quả:** Mô hình chuyên biệt hóa ví dụ như:
  - Trả lời tự nhiên như chatbot
  - Dịch chính xác Anh-Việt

Tác động của Fine-Tuning đến các lớp của kiến trúc Transformer

- **Embedding Layer:** Điều chỉnh Embedding sao cho phù hợp với từ vựng đặc thù của tác vụ

- **Self-Attention:** Tập trung vào từ khóa quan trọng của tác vụ

- **FFNN:** Tối ưu để tạo đầu ra phù hợp với tác vụ.

3.3 Reinforcement Learning from Human Feedback (RLHF)

Định nghĩa : RLHF là một trường hợp đặc biệt của finetuning giúp tinh chỉnh LLM bằng cơ chế học tăng cường (Reinforcement Learning) dựa trên phản hồi của con người , giúp mô hình tạo ra đầu ra phù hợp với mong muốn của người dùng.

Tại sao cần RLHF ?
Pretraning và Finetuning truyền thống chỉ tối ưu mô hình dựa trên dữ liệu tĩnh, không phản ánh được sự ưu tiên của con người

Vấn đề: LLM có thể sinh ra nội dung độc hại, sai lệch, hoặc không phù hợp dù đã qua fine-tuning

Giải pháp: RLHF căn chỉnh mô hình bằng phản hồi trực tiếp từ con người hoặc mô hình giả lập (reward model).

Quy trình RLHF:

1. **Thu thập dữ liệu phản hồi (Human Feedback)**
    - **Mục tiêu :** Tạo bộ dữ liệu ghi nhận sự ưa thích của con người với các câu trả lời khác nhau.
    - **Cách làm :** Cho người dùng xem nhiều phiên bản trả lời của LLM cho cùng một prompt. Sau đó yêu cầu họ xếp hạng hoặc chọn câu trả lời tốt nhất

2. **Huấn luyện mô hình phần thưởng (Reward Model)**

    - **Mục tiêu :** Xây dựng mô hình có thể dự đoán điểm số phần thưởng (reward) cho câu trả lời, thay thế con người đánh giá.

    - **Cách hoạt động:**
        - **Input :** Một cặp (prompt, reponse)
        - **Output:** Điểm số reward

    - Mô hình được huấn luyện để dự đoán điểm số giống với xếp hạng của con người.

- **Công thức loss :**

$$\mathcal{L}_{\text{RM}} = -\mathbb{E}_{(x,y_w,y_l)} \left[\log \sigma(r_\phi(x, y_w) - r_\phi(x, y_l))\right]$$

Trong đó:
Y_w : Câu trả lời “tốt” (được xếp hạng cao)
Y_l : Câu trả lời “kém” (được xếp hạng kém)
r_ϕ : Reward model.

3. Tối ưu LLM bằng PPO

- Mục tiêu: Điều chỉnh LLM để đưa ra câu trả lời có điểm reward cao nhất theo RM, đồng thời tránh “lệch quá xa” so với LLM gốc.

- Cơ chế Proximal Policy Optimization (PPO) - thuật toán học tăng cường ổn định:
    - LLM sinh câu trả lời y cho prompt x
    - Reward Model tính điểm
    - PPO cập nhật policy (LLM) để tăng xác suất sinh ra câu trả lời có điểm cao

- Hàm mục tiêu:

$$\mathcal{L}_{\text{PPO}} = \mathbb{E}_{(x,y)} \left[r_\phi(x, y) - \beta \cdot \text{KL}(\pi_{\text{RL}}(y|x) \parallel \pi_{\text{ref}}(y|x))\right]$$

Trong đó:
    - π_RL : LLM đang được tối ưu.
    - π_ref : LLM gốc (trước RLHF) để tránh thay đổi quá lớn.
    - β : Hệ số cân bằng giữa tối ưu reward và giữ ổn định để tránh lệch quá xa so với LLM gốc

4. Ví dụ minh họa RLHF trên ChatGPT

    - Prompt: “Cách hack tài khoản ngân hàng”

        - LLM trước RLHF: Đưa ra hướng dẫn chi tiết (nguy hiểm)
        - LLM sau RLHF : Từ chối trả lời và khuyên dùng dịch vụ hợp pháp.
    - Cơ chế:
        - Reward Model học rằng câu trả lời “Tôi không thể trả lời câu hỏi này” được con người đánh giá cao
        - PPO điều chỉnh LLM để giảm xác suất sinh nội dung độc hại

5. **Tác động của RLHF đến các Lớp trong Transformer**
    - **Embedding Layer**: Điều chỉnh vector từ vựng để ưu tiên từ ngữ an toàn (Vd: “từ chối” thay vì “hack”)
    - **Self-Attention** : Tăng cường attention vào từ khóa quan trọng cho câu trả lời phù hợp (Vd : “không nên”)
    - **FFNN:** Tối ưu để sinh đầu ra cân bằng giữa tự nhiên và tuân thủ nguyên tắc

4.Nguồn

1. https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf
2. https://vinbigdata.com/cong-nghe-giong-noi/lam-the-nao-de-dao-tao-large-language-models.html?gad_source=1&gad_campaignid=22427790121&gbraid=0AAAAAp9MqYFE8TqSBcE-5Dgu8iO8UXPZ

_&gclid=CjwKCAjwl_XBBhAUEiwAWK2hzmh_GqmiFOxTCgjyM0b-jQtgCIzDj9FmAVxBOpAv7EI8h-SHUZdcmxoC-vEQAvD_BwE
3. https://chatgpt.com/
4. https://chat.deepseek.com/
5. https://tinhte.vn/thread/huong-dan-prompt-tu-co-ban-den-nang-cao-p1-zero-shot-va-few-shot-prompting.4011566/
6. https://tinhte.vn/thread/huong-dan-prompt-tu-co-ban-den-nang-cao-p3-step-back-prompting-va-chain-of-thought-cot.4012561/
7. https://www.viettelidc.com.vn/tin-tuc/llm-la-gi#:~:text=LLM%2C%20hay%20m%C3%B4%20h%C3%ACnh%20ng%C3%B4n,nhi%E1%BB%81u%20l%C4%A9nh%20v%E1%BB%B1c%20kh%C3%A1c%20nhau.
8. https://vnptai.io/vi/blog/detail/llm-la-gi#:~:text=Large%20Language%20Model%20(LLM)%20%E2%80%93,LLM%20l%C3%A0%20g%C3%AC?