ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN
KHOA CÔNG NGHỆ THÔNG TIN

MATH FOR AI

Attention is all you need

NHÓM 5

Giáo viên	Cấn Trần Thành Trung
	Nguyễn Ngọc Toàn
23122003	Nguyễn Văn Linh
23122022	Trần Hoàng Gia Bảo
23122026	Trần Chấn Hiệp
23122040	Nguyễn Thị Mỹ Kim

TP. HỒ CHÍ MINH, THÁNG 6/2025

Mục lục

1 Giới thiệu **2**

2 Embedding **2**
2.1 Embedding . . . 2
2.1.1 Khái niệm . . . 2
2.1.2 Chức năng . . . 3
2.1.3 Scaling Factor . . . 3
2.2 Positional Encoding . . . 4
2.2.1 Khái niệm . . . 4
2.2.2 Tại sao cần Positional Encoding . . . 4
2.2.3 Công thức toán học . . . 4

3 Cơ chế Attention **5**
3.1 Attention . . . 5
3.2 Scaled Dot-Product Attention . . . 5
3.3 Query, Key, Value . . . 7

4 Kiến trúc mô hình Transformer **8**
4.1 Tổng quan . . . 8
4.2 Kiến trúc . . . 8
4.2.1 Data Flow tổng quan của mô hình . . . 9
4.2.2 Multi-head Attention . . . 10
4.2.3 Position-wise Feed Forward Network . . . 13
4.2.4 Add and Norm . . . 14
4.2.5 Encoder Block . . . 16
4.2.6 Decoder Block . . . 17
4.3 Cách cập nhật của mô hình Transformer . . . 17
4.4 Giải thuật tham lam - Greedy Decode . . . 18

5 Ví dụ trực quan cho mô hình Transformer **19**
5.1 Học từ dữ liệu . . . 19
5.2 Đoán nghĩa của chuỗi . . . 20

6 Thực nghiệm **20**
6.1 Dữ liệu . . . 20
6.2 Tiền xử lí dữ liệu . . . 20
6.3 Setup mô hình Transformer . . . 21
6.4 BLEU score . . . 21
6.5 Phân tích, đánh giá . . . 23
6.6 Kết luận . . . 23

7 Phụ lục **24**
7.1 Paper . . . 24
7.2 Slide thuyết trình . . . 24
7.3 Video thuyết trình . . . 24
7.4 Dữ liệu . . . 24
7.5 Source Code . . . 24

Tóm tắt nội dung

Báo cáo này trình bày việc triển khai và đánh giá mô hình Transformer cho tác vụ dịch máy, đặc biệt tập trung vào bài báo "Attention is All You Need". Mô hình Transformer, với cơ chế self-attention, mang lại sự cải tiến đáng kể so với các mô hình truyền thống trong xử lý chuỗi, đặc biệt là trong dịch máy. Những khái niệm mới được nhóm nghiên cứu kĩ bao gồm việc sử dụng positional encoding, scaled dot product attention, multi-head attention,... giúp mô hình xử lý các chuỗi dài trong dữ liệu tuần tự một cách hiệu quả hơn. Nhóm cũng đã áp dụng mô hình Transformer cho tác vụ dịch từ tiếng Anh sang tiếng Việt, sử dụng một bộ dữ liệu khoảng 250.000 mẫu. Hiệu suất của mô hình được đánh giá thông qua BLEU score, đạt 0.6674 trên toàn bộ bộ dữ liệu và 0.6785 trên tập validation. Báo cáo cũng bao gồm phân tích về kiến trúc mô hình, tiền xử lý dữ liệu và thiết lập huấn luyện. Nghiên cứu bài báo "Attention is All You Need" đã tạo nền tảng kiến thức cho nhóm để nghiên cứu các bài báo và mô hình phức tạp có sử dụng mô hình Transformer sau này.

1 Giới thiệu

Mô hình Transformer là một kiến trúc dựa trên cơ chế Attention để xử lý chuỗi dữ liệu tuần tự, đồng thời khắc phục được vấn đề Vanishing Gradient mà các mô hình như RNN hay LSTM thường gặp phải khi lan truyền thông tin liên tục qua nhiều lớp. Mô hình Transformer bao gồm hai phần chính: Encoder và Decoder. Trong đó, Encoder có nhiệm vụ tiếp nhận và mã hóa thông tin từ chuỗi đầu vào, còn Decoder sử dụng thông tin từ Encoder để sinh ra chuỗi đầu ra, giúp mô hình thực hiện các nhiệm vụ như dịch máy, tóm tắt văn bản hay sinh văn bản.

Trong ứng dụng dịch máy, mô hình Transformer hoạt động theo cơ chế seq2seq (sequence-to-sequence), với đầu vào là một câu trong ngôn ngữ nguồn và đầu ra là bản dịch của câu đó sang ngôn ngữ đích. Quá trình này bao gồm hai bước chính:

Encoder: Đầu vào là một câu trong ngôn ngữ nguồn. Mỗi từ trong câu được chuyển đổi thành một vector đặc trưng thông qua cơ chế embedding, sau đó qua các lớp attention để học mối quan hệ giữa các từ trong câu, bất chấp vị trí của chúng trong chuỗi. Điều này cho phép mô hình có khả năng tập trung vào các phần quan trọng của câu khi xử lý từng từ.

Decoder: Sau khi thông tin từ Encoder được mã hóa, Decoder tiếp nhận và sử dụng thông tin này để sinh ra câu dịch. Quá trình sinh câu dịch diễn ra theo từng bước, mỗi bước mô hình sinh ra một token (từ) mới dựa trên các token đã sinh ra trước đó, kết hợp với thông tin từ Encoder.

Quá trình tạo ra output có thể sử dụng các phương pháp khác nhau, trong đó một phương pháp phổ biến là thuật toán tham lam (greedy algorithm). Thuật toán này chọn token tiếp theo có xác suất cao nhất dựa trên các token đã được sinh ra trước đó, và tiếp tục như vậy cho đến khi chuỗi đầu ra hoàn thành. Mặc dù phương pháp này đơn giản và hiệu quả, nó có thể không phải lúc nào cũng tạo ra kết quả tối ưu, đặc biệt trong những trường hợp cần sự đa dạng trong kết quả dịch.

Trong thực tế, mỗi token được sinh ra dựa trên xác suất (probability) mà mô hình học được từ các token trước đó trong chuỗi, thông qua cơ chế Self-Attention. Quá trình này lặp lại cho đến khi mô hình hoàn thành bản dịch của câu đầu vào.

2 Embedding

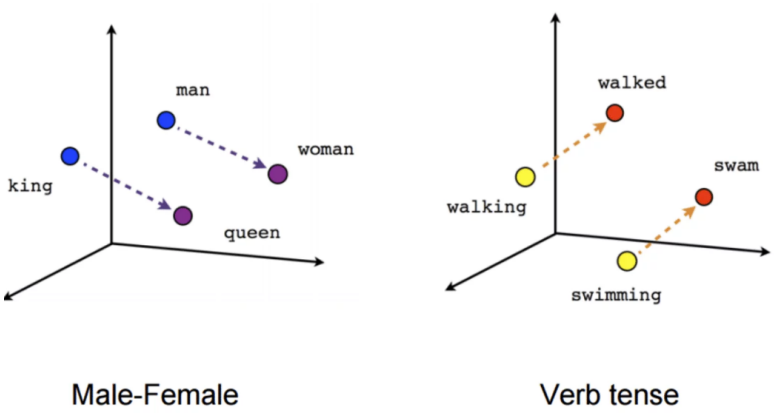
2.1 Embedding

2.1.1 Khái niệm

Embedding là quá trình biến đổi các từ/token trong ngôn ngữ tự nhiên thành các vector học liên tục trong không gian đa chiều. Trong bài báo này, mỗi từ được ánh xạ tới một vector cố định

có chiều là $d_{\text{model}} = 512$. Mỗi token được đại diện bởi một chỉ số nguyên (index) trong từ điển, và chỉ số đó được dùng để truy xuất ra một vector embedding từ một ma trận trọng số có kích thước vocab_size $\times\, d_{\text{model}}$.

2.1.2 Chức năng



Hình 1: Minh họa mối quan hệ đại số trong không gian embedding

- Các từ có nghĩa tương đồng sẽ có embedding gần nhau trong không gian vector theo khoảng cách cosine hoặc Euclidean
- Chuyển đổi dữ liệu rời rạc (discrete) thành dạng biểu diễn liên tục (continuous) để mạng neural có thể xử lý hiệu quả
- Embedding cho phép mô hình học được các mối quan hệ ngữ nghĩa phức tạp giữa các từ thông qua quá trình huấn luyện
- Tính chất đại số của embedding: Các phép toán vector trong không gian embedding có thể phản ánh các mối quan hệ ngữ nghĩa. Ví dụ nổi tiếng:

- King - Man + Woman = Queen
- Paris - France + Italy = Rome
- Swimming - Swim + Walk = Walking

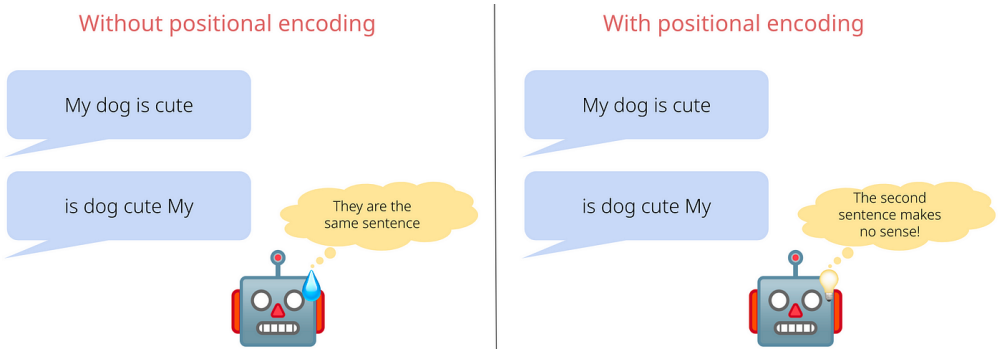
- Trong mô hình Transformer, embedding đầu ra chia sẻ trọng số với embedding đầu vào, điều này giúp giảm số lượng tham số và đảm bảo tính nhất quán giữa không gian biểu diễn đầu vào và đầu ra

2.1.3 Scaling Factor

Trong bài báo "Attention Is All You Need", các embedding được nhân với $\sqrt{d_{\text{model}}}$ trước khi cộng với positional encoding để cân bằng độ lớn giữa hai thành phần này:

$$\text{Input} = \sqrt{d_{\text{model}}} \cdot \text{Embedding} + \text{Positional Encoding}$$

2.2 Positional Encoding



Hình 2: Minh họa về chức năng của positional encoding

2.2.1 Khái niệm

Positional Encoding là kỹ thuật được sử dụng để cung cấp thông tin về vị trí của các token trong chuỗi đầu vào cho mô hình Transformer. Khác với các mô hình như RNN hoặc CNN vốn có khả năng xử lý tuần tự theo thời gian hoặc theo không gian, cơ chế attention trong Transformer là bất biến với hoán vị (permutation-invariant), tức là nó xử lý tất cả các vị trí song song mà không phân biệt thứ tự. Do đó, cần phải thêm thông tin vị trí một cách tường minh để mô hình hiểu được cấu trúc của chuỗi.

2.2.2 Tại sao cần Positional Encoding

- Cơ chế Self-Attention xử lý tất cả các token đồng thời, không có khái niệm thứ tự tự nhiên giữa các token.
- Nếu không có Positional Encoding, hai câu như "A loves B" và "B loves A" sẽ có biểu diễn embedding giống hệt nhau — điều này khiến mô hình không thể phân biệt được ý nghĩa của các câu dựa trên thứ tự.
- Thông tin vị trí là yếu tố thiết yếu để mô hình hiểu đúng ngữ nghĩa, cú pháp và quan hệ ngữ cảnh trong câu.

2.2.3 Công thức toán học

Trong bài báo *"Attention Is All You Need"*, các tác giả sử dụng Positional Encoding cố định (không học được), được tính toán dựa trên các hàm sin và cos với các tần số khác nhau:

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{\frac{2i}{d_{\text{model}}}}}\right) \tag{1}$$

$$PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{\frac{2i}{d_{\text{model}}}}}\right) \tag{2}$$

Trong đó:

- pos là vị trí của token trong chuỗi (từ 0 đến `max_length` - 1)

- i là chỉ số của chiều trong vector embedding (từ 0 đến $d_{\text{model}}/2 - 1$)
- $d_{\text{model}} = 512$ là chiều của vector embedding
- Các chiều chẵn ($2i$) sử dụng hàm sin, các chiều lẻ ($2i + 1$) sử dụng hàm cos

Ý nghĩa của công thức:

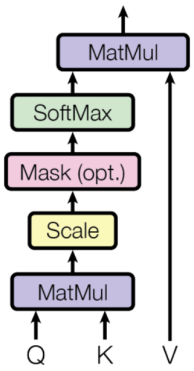
- Mỗi vị trí *pos* trong chuỗi được gán một vector encoding duy nhất.
- Việc sử dụng các bước sóng khác nhau giúp mô hình học được khoảng cách tương đối giữa các token.
- Vì hàm sin/cos có tính tuần hoàn, mô hình có thể tổng quát hóa cho các chuỗi có độ dài lớn hơn so với dữ liệu huấn luyện ban đầu.

3 Cơ chế Attention

3.1 Attention

Attention là một hàm ánh xạ từ các query và các (key-value) vào output. Output là tổ hợp tuyến tính của các giá trị v_i, với trọng số là mức độ tương thích giữa truy vấn và khóa $q_i \cdot k_i$, tính bằng dot-product rồi chuẩn hóa qua softmax.

3.2 Scaled Dot-Product Attention



Hình 3: Scaled Dot-Production Attention

Scaled Dot-Product Attention là một cải tiến dựa trên Attention (Attention chia $\sqrt{d_k}$). Công thức dạng ma trận:

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

với Q, V, K là các ma trận có các dòng là các queries, keys, values tương ứng. Ta cũng có công thức softmax là:

$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{n} e^{z_j}}$$

với z_i đại diện cho token thứ i trong một chuỗi và $z_j, j \in \overline{1,n}$ là các token trong chuỗi đó.

Việc chia $\sqrt{d_k}$ sẽ giúp cho softmax ổn định hơn (Giả sử các thành phần của q và k là ngẫu nhiên với trung bình 0 và phương sai 1 thì $q \cdot k = \sum q_i k_i$ có trung bình là 0 và phương sai d_k). Scaled Dot-Product Attention là dạng Attention được sử dụng trong mô hình Transformer.

Giải thích:

Ta có t' queries và có t keys và có t values lần lượt là t' dòng của Q, t dòng K và t dòng của V. Các queries này như các câu hỏi mà mô hình sẽ học dự trên đó, keys là các cách trả lời cho các câu hỏi đó. Việc tính QK^T sẽ thu được ma trận có kích thước $\mathbb{R}^{t' \times t}$ là ma trận mà mỗi hàng của nó là vector t keys ứng với một query $\{q_i \cdot k_1, ..., q_i \cdot k_t\}$, với q_i, k_i là các dòng của Q, K. Sau đó ta chia cho $\sqrt{d_k}$ và áp dụng sofmax cho từng dòng của ma trận thu được, cuối cùng là nhân ma trận với ma trận V để thu được ma trận Attention có kích thước $\mathbb{R}^{t' \times d_v}$ có dòng i là tổ hợp tuyến tính của t vector dòng của V ($\sum \alpha_j v_j$ trong đó α_j là kết quả của $q_i \cdot k_j$ và đưa qua softmax).

Ý nghĩa:

Việc tạo ma trận Q, K cũng giống như việc ta xây dựng các câu hỏi mà mô hình cần hỏi với chuỗi và các câu trả lời cho từng câu hỏi đó, ta xem việc tính dot-product là một cách đánh giá sự phù hợp của tất cả các câu trả lời cho từng câu hỏi. Sau cùng ma trận Value giống như một cách tổng hợp thông tin thu được sau khi hỏi và trả lời.

Ví dụ:

Giả sử ta có m queries với $Q = [q_1, \ldots, q_m]^T$ với mỗi một query là một vector dòng $q_i \in \mathbb{R}^{d_k}$. Tương tự ta có n keys $K = [k_1, \ldots, k_n]^T$ với mỗi key là một vector dòng $k_i \in \mathbb{R}^{d_k}$ và n values $V = [v_1, \ldots, v_n]^T$ với mỗi value là một vector dòng $v_i \in \mathbb{R}^{d_k}$. Lưu ý thì trong bài đa số ta cho $m = n$ để dễ tính toán. Thực chất thì $m \neq n$ vẫn tính toán được. Vì giới hạn số token cho input và output là không nhất thiết giống nhau. Ta cũng có thể thấy, số values và số keys trong mô hình Transformer luôn bằng nhau.

Ta tạo ma trận self attention scores như sau:

$$S = \frac{1}{\sqrt{d_k}} \begin{bmatrix} q_1 \cdot k_1 & q_1 \cdot k_2 & \ldots & q_1 \cdot k_n \\ q_2 \cdot k_1 & q_2 \cdot k_2 & \ldots & q_2 \cdot k_n \\ \vdots & \vdots & \ddots & \vdots \\ q_m \cdot k_1 & q_m \cdot k_2 & \ldots & q_m \cdot k_n \end{bmatrix}$$

Cụ thể hơn thì $S_{ij} = \text{scores}(q_i, k_j) = \frac{1}{\sqrt{d_k}}(q_i \cdot k_j)$.

Sử dụng softmax lên mỗi hàng của ma trận S để có được self attention weights:

$$A = \text{softmax}(S) \in \mathbb{R}^{m \times n}$$

Cụ thể tại hàng i cột j của ma trận attention weights thì:

$$A_{ij} = \text{softmax}(scores(q_i, k_j)) = \frac{\exp(\text{scores}(q_i, k_j))}{\sum_{j=1}^{n} \exp(\text{scores}(q_i, k_j))}$$

Cuối cùng, ma trận output sẽ có kích thước $m \times d_k$:

$$O = AV$$

Để có được hàng thứ i của ma trận O, ta lấy n trọng số nhờ lấy n giá trị tại hàng thứ i của ma trận A (trong n giá trị này, giá trị A_{ij} trong n giá trị có giá trị càng lớn thì chứng tỏ k_j tại đó ảnh hưởng nhiều hơn với q_i), ta nhân tương ứng n trọng số này của A_i với n các vector v_j trong

V ta có được $v'_j,\ j \in \overline{1,n}$, khi này ta cộng tất cả các vector v'_j này lại với nhau thì ta được hàng thứ i của ma trận O, tại hàng i thì có d_k giá trị.

$$O_i = \sum_{j=1}^{n} A_{ij} \cdot v_j = \sum_{j=1}^{n} \text{softmax}(\frac{1}{\sqrt{d_k}}(q_i \cdot k_j)) \cdot v_j$$

Ta minh họa bước cuối tính output để cho ra ma trận attention. Giả sử cho $n = 3$ và hàng thứ i của ma trận A là (không cần quan tâm có bao nhiêu queries hay m bằng bao nhiêu vì ta chỉ xét hàng thứ i):

$$A_i = \begin{bmatrix} 0.1 & 0.6 & 0.3 \end{bmatrix}$$

Lúc này giả sử $d_k = 2$ và có 3 vector values là $\vec{v}_1 = \begin{bmatrix} 1 & 2 \end{bmatrix}, \vec{v}_2 = \begin{bmatrix} 3 & 4 \end{bmatrix}, \vec{v}_3 = \begin{bmatrix} 5 & 6 \end{bmatrix}$ Lúc này ta thực hiện tính tổng các $\vec{v}'_j$ ($\vec{v}'_j$ được tính bằng cách nhân tương ứng n trọng số trong A_i với từng vector $\vec{v}_j$ tương ứng):

$$O_i = \vec{v}'_1 + \vec{v}'_2 + \vec{v}'_3 = A_{i1}\vec{v}_1 + A_{i2}\vec{v}_2 + A_{i3}\vec{v}_3 = 0.1\vec{v}_1 + 0.6\vec{v}_2 + 0.3\vec{v}_3 = \begin{bmatrix} 3.4 & 4.4 \end{bmatrix}$$

Như vậy ta đã hiểu rõ hơn về cách tính Attention, sau đây ta sẽ tìm hiểu cách sử dụng Query, Key, Value.

3.3 Query, Key, Value

Biểu diễn đầu vào và ma trận trọng số trong Self-Attention

Gọi X là ma trận đầu vào, là kết quả sau khi câu đã được qua lớp *embedding* và *positional encoding*. Khi đó:

$$E \in \mathbb{R}^{n \times d_{\text{model}}}$$

trong đó:

- n: số lượng token trong câu.
- d_{model}: số chiều của vector embedding.

Để tính toán cơ chế self-attention, đầu vào E sẽ được chiếu sang ba không gian: Query, Key và Value, thông qua ba ma trận trọng số học được:

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

trong đó d_k là số chiều của không gian Key và Query (thường bằng nhau, và bằng d_v – số chiều của Value).

Từ đó, ta có:

$$Q = EW^Q$$
$$K = EW^K$$
$$V = EW^V$$

Trong đó:

- Q: ma trận *Query*, gồm n vector truy vấn, mỗi vector tương ứng với một token trong câu.
- K: ma trận *Key*, chứa các vector dùng để so khớp với các truy vấn.
- V: ma trận *Value*, chứa thông tin được trích xuất dựa trên mức độ khớp giữa Query và Key.

Vai trò của Query, Key và Value

Sau khi được tạo ra, bộ ba ma trận Query, Key, và Value được sử dụng trong cơ chế Multi-Head Attention theo ba cách khác nhau. Vai trò và nguồn gốc của chúng thay đổi tùy thuộc vào vị trí trong kiến trúc mô hình.

Trong các lớp Self-Attention của Encoder

- **Nguồn gốc:**
$$Q, K, V \leftarrow \text{output của lớp Encoder trước}$$
- **Vai trò:** Cơ chế này cho phép "mỗi vị trí trong encoder có thể chú ý đến tất cả các vị trí trong lớp trước đó của encoder". Dựa trên mức độ liên quan đó, nó tạo ra một biểu diễn ngữ cảnh mới cho chính mình bằng cách tổng hợp có trọng số các thông tin (Value) từ toàn bộ câu

Trong các lớp Self-Attention của Decoder

- **Nguồn gốc:**.
$$Q, K, V \leftarrow \text{output của lớp Decoder trước}$$
- **Vai trò:** Ngăn chặn dòng thông tin từ tương lai, một cơ chế được áp dụng để loại bỏ các kết nối không hợp lệ, đảm bảo việc dự đoán một token chỉ phụ thuộc vào các token đã được sinh ra trước đó.

Trong các lớp Encoder-Decoder Attention

Đây là cơ chế kết nối giữa hai khối của mô hình.

- **Nguồn gốc:**
$$Q \leftarrow \text{output của lớp attention trước trong Decoder}$$
$$K, V \leftarrow \text{output cuối cùng của Encoder (memory)}$$
- **Vai trò:** So Query này với các Key từ Encoder và tổng hợp các Value tương ứng, Decoder có thể căn chỉnh và trích xuất ngữ cảnh cần thiết từ câu nguồn để tạo ra một bản dịch tốt.

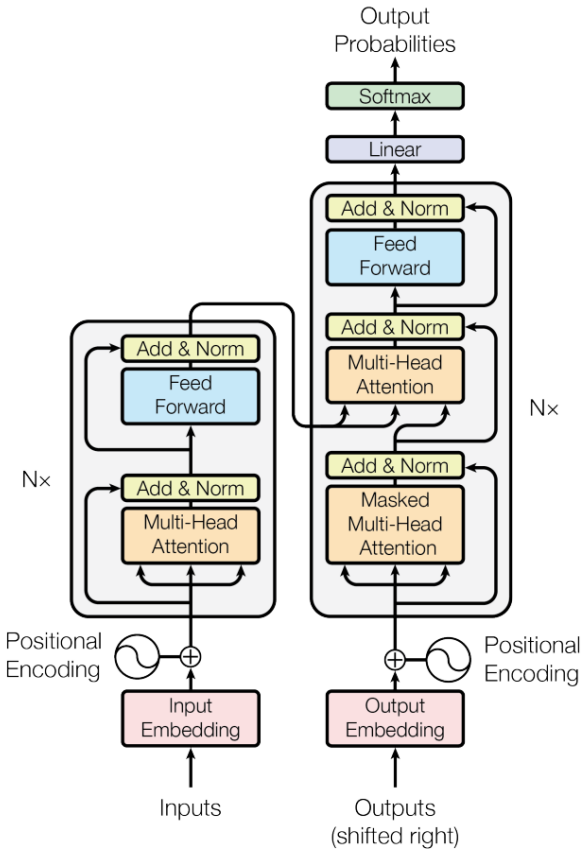
4 Kiến trúc mô hình Transformer

4.1 Tổng quan

Mô hình Transformer là mô hình deep learning không dùng RNN (Recurrent Neural Network) và CNN (Convolutional Neural Network), mà dùng Self-Attention (sẽ được đề cập sau) để xử lí chuỗi.

4.2 Kiến trúc

Mô hình Transformer gồm 2 thành phần chính là Encoder và Decoder. Encoder là phần đảm nhiệm xử lí và mã hóa chuỗi đầu vào, Decoder (không trực tiếp) sinh ra chuỗi đầu ra. Encoder và Decoder đều gồm N layers (lớp).



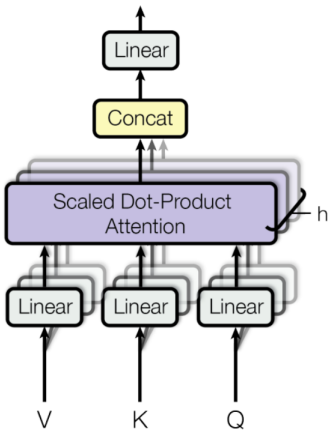
Hình 4: Kiến trúc mô hình Transformer

4.2.1 Data Flow tổng quan của mô hình

Ta sẽ giới thiệu qua là cách mô hình Transformer học từ dữ liệu và Dự đoán kết quả. Đầu tiên giả sử ta có **chuỗi input** (input sequence), ta sẽ chia chuỗi (sequence) thành **n tokens** (mỗi token được xem là một từ hay kí tự đặc biệt), sau đó từng token sẽ được đối chiếu vào từ điển (Từ điển cho Input) (Dictionary) (ta quy ước đánh số chỉ cho từng từ trong từ điển, số từ ta xét là **vocab_size**), ta dựng một ánh xạ từ các từ trong từ điển vào các vector có thể học có kích thước $\mathbb{R}^{d\_model}$ - thao tác mã hóa này cho chuỗi input được gọi là **Embedding**, sau đó các vector mã hóa này sẽ được thêm yếu tố vị trí của chúng trong chuỗi - đây là **Positional Encoding**, và rồi thì chúng sẽ được chạy qua N layers của Encoder để ra kết quả cuối tại block này - gọi là **Encoder Block Ouput.**
Bây giờ, giả sử chuỗi input có output mà mô hình cần học, chuỗi output cũng được phân thành các tokens. Sau đó các từ này sẽ được mã hóa thành các vector - Embedding (Dựa theo Từ điển cho Output) và lại cập nhật thêm thông tin vị trí - Positional Encoding, sau khi xong thì mới được chính thức truyền vào Decoder Block, và mỗi layer trong N layers của Decoder còn sử dụng thêm **Encoder Block Ouput**, cuối cùng mô hình trả ra một bộ các **vector decoder output** đã được trau dồi thông tin từ mô hình, từ các vector này mô hình sẽ học được việc dự đoán

chuỗi output từ chuỗi input. Chi tiết sẽ được đề cập trong từng phần sau.

4.2.2 Multi-head Attention



Hình 5: MultiHead Attention

Tổng quan

Trước tiên ta sẽ tìm hiểu head là gì, ta có công thức của head theo bài báo như sau:

$$\text{head} = \text{Attention}(QW^Q, KW^K, VW^V)$$

với $W^Q, W^K \in \mathbb{R}^{d_{model} \times d_k}, W^V \in \mathbb{R}^{d_{model} \times d_v}$ là các ma trận chứa hệ số có thể học từ data. $Q, K, V \in \mathbb{R}^{n \times d_{model}}$ ở đây là các ma trận input đi vào sub-layer Multi-head Attention.
Trong mô hình Transformer thì ta sẽ định nghĩa $d_k = d_v = \frac{d_{model}}{h}$, với h là một hằng số (h phải là ước của d_{model}), khi này ta thấy các ma trận đầu ra của head sẽ là các ma trận có kích thước $\mathbb{R}^{n \times d_k}$. Chính vì vậy ta sẽ chia thành h head, mỗi $head_i$ sẽ dùng cơ chế Attention để học ra các ma trận có kích thước $\mathbb{R}^{n \times d_k}$. Cụ thể:

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

và ta sẽ ghép h head này lại thành một ma trận, gọi là $\text{Concat}(head_1, ..., head_h)$ có kích thước $\mathbb{R}^{n \times hd_k} = \mathbb{R}^{n \times d_{model}}$. Ta kí hiệu:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(head_1, ..., head_h)W^O$$

với $W^O \in \mathbb{R}^{h.d_v \times d_{model}} = \mathbb{R}^{d_{model} \times d_{model}}$

Giải thích ý nghĩa:

Ý tưởng chính của Multi-Head Attention là ta sẽ chia nhỏ thành h Attention và để chúng cùng học từ một input. Như ta đã biết cách từng Attention học từ input (tương tự xây dựng câu trả lời cho các truy vấn với input), ở đây, ta chia nhỏ ra để ta có thể học được nhiều đặc trưng hơn từ một input. Sau đó ghép chúng lại tinh chỉnh (nhân cho W^O) để ra được ma trận học cuối cùng thay vì dùng thẳng một Attention duy nhất để học từ input. Việc này giúp cho mô hình có thể tập hợp được nhiều thông tin hơn ở nhiều khía cạnh, đặc trưng (như thứ tự, ngữ cảnh, ...). Trên thực tế, cơ chế này đã chứng minh hiệu quả vượt trội trong mô hình Transformer.

Data Flow của Multi-Head Attention SubLayer trong Encoder Layer:

Ta có một ma trận input embedding (input của Encoder Block) $X \in \mathbb{R}^{n \times d_{model}}$, khi này $Q = K = V = X$ và sẽ cho đi qua Multi-Head Attention Sub Layer tạo ra ΔX rồi qua lớp Add & Norm thu được X' cùng kích thước nhưng đã được học nhiều hơn. Ma trận X' này sẽ đi qua Feed Forward Network (đề cập bên dưới) và Add & Norm và đảm bảo nhận được ma trận $X'' \in \mathbb{R}^{n \times d_{model}}$.

Lớp Encoder sau lấy output của lớp Encoder trước làm input. Quá trình đi từ Multi-Head Attention đến Feed Forward Network lặp lại tương tự sao cho đủ N lần (N là số lớp Encoder Layer trong Encoder Block).

Data Flow của Masked Multi-Head Attention SubLayer trong Decoder Layer:

Ta có một ma trận output embedding (input của lớp Decoder) $X \in \mathbb{R}^{n \times d_{model}}$, khi này $Q = K = V = X$ và sẽ đi qua Masked Multi-Head Attention tạo ra X' cùng kích thước.

Như đã biết trong cơ chế attention, mỗi từ sẽ tính toán mức độ chú ý đối với các từ khác thông qua một ma trận attention scores. Self-attention score giữa hai từ i và j trong một chuỗi tính như sau:

$$\text{score}_{\text{ij}} = \frac{\text{Q}_\text{i}\text{K}_\text{j}^\text{T}}{\sqrt{\text{d}_\text{k}}}$$

Ở phần Masked Attention, chúng ta muốn các từ ở vị trí hiện tại chỉ chú ý đến nó và các từ phía trước, nên ta sẽ mask (che) các từ sau nó. Chúng ta cần tạo ra ma trận mask, một ma trận vuông có cùng kích thước của ma trận attention (ma trận kích thước n * n với n là số token tối đa trong câu). Tại các giá trị trong ma trận attention được đánh dấu mask, ta làm giá trị $\text{score}_{\text{ij}} = -\infty$, điều này khiến cho giá trị tại vị trí đó sau khi softmax trở về 0. Giả sử ta có ma trận scores $S \in \mathbb{R}^{4 \times 4}$ với n = 4, nghĩa là có 4 token trong câu như sau:

$$\text{S} = \text{scores} = \frac{QK^T}{\sqrt{d_k}} = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix}$$

Sau đó, ta tạo một ma trận mask dạng lower triangular mask:

$$\text{mask} = \begin{bmatrix} 0 & -\infty & -\infty & -\infty \\ 0 & 0 & -\infty & -\infty \\ 0 & 0 & 0 & -\infty \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Bây giờ ta áp dụng mask cho ma trận scores, thì:

$$\text{masked\_scores} = \text{S} + \text{mask} = \begin{bmatrix} 1 & -\infty & -\infty & -\infty \\ 5 & 6 & -\infty & -\infty \\ 9 & 10 & 11 & -\infty \\ 13 & 14 & 15 & 16 \end{bmatrix}$$

Sau khi softmax, các giá trị tại các vị trí $-\infty$ sẽ có attention weights bằng 0, nghĩa là mô hình không chú ý đến các từ phía sau khi tính toán từ hiện tại. Attention weights lúc này:

$$A = \text{softmax}(\text{masked\_scores}) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ \frac{e^5}{e^5+e^6} & \frac{e^6}{e^5+e^6} & 0 & 0 \\ \frac{e^9}{e^9+e^{10}+e^{11}} & \frac{e^{10}}{e^9+e^{10}+e^{11}} & \frac{e^{11}}{e^9+e^{10}+e^{11}} & 0 \\ \frac{e^{13}}{e^{13}+e^{14}+e^{15}+e^{16}} & \frac{e^{14}}{e^{13}+e^{14}+e^{15}+e^{16}} & \frac{e^{15}}{e^{13}+e^{14}+e^{15}+e^{16}} & \frac{e^{16}}{e^{13}+e^{14}+e^{15}+e^{16}} \end{bmatrix}$$

Giả sử ta xét chuỗi "<sos> Anh yêu em <eos>" đưa vào Output Embedding. Khi đó, chuỗi được embed đưa vào decoder là "<sos> Anh yêu em" (shifted right).

Giả sử tại token 0 (hàng đầu tiên), $A_{00} \neq 0, others = 0$, nghĩa là tại token "<sos>", token này chỉ chú ý đến nó mà không nhìn đến từ khác. Tương tự tại hàng thứ hai $A_{10}, A_{11} \neq 0$, nghĩa là token "Anh" chỉ chú ý đến nó và "<sos>", các token sau không được chú ý. Tương tự như vậy cho các token khác trong câu.

Sau đó, ma trận attention_weights A này được nhân với ma trận $V \in \mathbb{R}^{n \times d_k}$ tạo nên ma trận attention.

Quay trở về với data flow của Mask MultiHead Attention, các head_i trong SubLayer này được tính dựa vào cách áp dụng mask như thế:

$$\text{head}_i = \text{MaskAttention}(QW_i^Q, KW_i^K, VW_i^V)$$

và ta sẽ ghép h head này lại thành một ma trận tương tự với cách làm của Multihead là:

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

với $\text{Concat}(\text{head}_1, \ldots, \text{head}_\text{h}) \in \mathbb{R}^{n \times hd_k}$ và $W^O \in \mathbb{R}^{hd_v \times d_{model}}$

Thực tế trong huấn luyện mô hình, Encoder/Decoder cũng có sử dụng ma trận mask khi gặp vị trí <pad>. Do kí tự "<pad>" được sử dụng để làm đầy ma trận embedding khi kích thước của ma trận embedding không đủ độ dài n (n là độ dài tối đa câu). Rõ ràng khi đưa vào mô hình, không phải câu nào cùng có độ dài giống nhau. Tuy nhiên, để đảm bảo kích thước các ma trận embedding đưa vào mô hình là $X \in \mathbb{R}^{n \times d_{model}}$ thì các kí tự "<pad>" cũng được embedding và điền vào các vị trí trống (nếu có) ở phía sau câu (1 kí tự "<pad>" được tính là 1 token, tương tự "<sos>" là Start of Sentence và "<eos>" là End of Sentence).

Đối với ví dụ Input: "I love you" và Output là "Anh yêu em", chọn $n = 5$ và d_{model} bất kì thì các ma trận embedding sẽ biểu diễn như sau:

$$\text{InputEmbedding} = \begin{bmatrix} - < sos > - \\ -I- \\ -love- \\ -you- \\ - < eos > - \end{bmatrix} \in \mathbb{R}^{n \times d_{model}}$$

$$\text{OutputEmbedding} = \begin{bmatrix} - < sos > - \\ -Anh- \\ -\text{yêu}- \\ -em- \\ - < pad > - \end{bmatrix} \in \mathbb{R}^{n \times d_{model}}$$

Tại token "<pad>" (hàng cuối) của ma trận OutputEmbedding thì Masking sẽ làm attention weights tại hàng cuối này cũng bằng 0 và khi đưa vào mô hình chúng ta đảm bảo kích thước ma trận embedding đưa vào là $\mathbb{R}^{n \times d_{model}}$ nhưng cũng không làm thay đổi ngữ nghĩa của câu.

Do có N Decoder Layer trong Decoder Block, do đó ngoại trừ ma trận Output Embedding được đưa vào đầu tiên cho Encoder Block thì các Masked MultiHead Attention của các Decoder Layer sau đều lấy từ output của Decoder Layer phía trước nó.

Data Flow của Cross Multi-Head Attention SubLayer trong Decoder Layer:

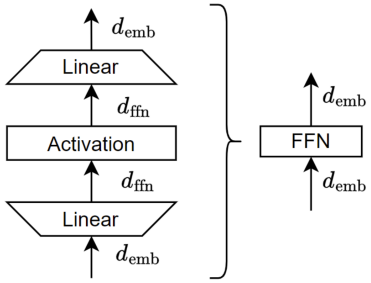
Cơ chế Cross MultiHead Attention khác với MultiHead Attention khi các query (truy vấn) lấy từ Encoder Output và keys (khóa) và values (giá trị) lấy từ Output của Mask MultiHead Attention ($Q \neq K = V$), trong khi query, key, value của self-attention đều đến từ cùng 1 nguồn đưa vào.

Trong mô hình Transformer của bài báo, tất cả N lớp Decoder Layer trong Decoder Block đều lấy keys và values từ output cuối cùng của Encoder Block. Còn đối với queries thì lấy output của Masked MultiHead Attention Sub Layer trước nó.

Cross Attention tìm mối quan giữa queries (từ Masked MultiHead Attention của Decoder) và keys từ output của Encoder Block để tìm xác định attention weights trong câu input và câu output, nhờ đó xác định được "mức chú ý" của các token trong câu "input" để tạo ra token tiếp theo ra cho câu "output".

Giả sử chúng ta có một câu tiếng anh input là "the cat sat on the mat" và câu tiếng việt output là "con mèo ngồi trên thảm" thì câu tiếng việt sau khi đi qua lớp decoder (giải mã) của mô hình nó sẽ tạo ra các queries (ví dụ như "con" trong câu tiếng việt). Cross attention sẽ so sánh query "con" này với các key từ câu input như ("the", "cat", "sat", "on", "the", "mat") để xác dịnh phần nào của câu input "đáng chú ý" nhất để tạo ra từ tiếp theo trong câu dịch.

4.2.3 Position-wise Feed Forward Network



Hình 6: Feed Forward Network

Tổng quan

Feed-Forward là một dạng của MLP (Multi-Layer Perceptron), gồm 2 lớp biến đổi tuyến tính, và một lớp biến đổi phi tuyến. Trong mô hình thì lớp biến đổi phi tuyến là ReLU, và các vector trong từng ma trận sẽ được xử lí độc lập (Position-wise). Ta có công thức của Feed-Forward như sau:

$$FFN(x) = max(0, xW_1 + b_1)W_2 + b_2$$

trong đó: $x \in \mathbb{R}^{1 \times d_{model}}$ là một vector trong ma trận đầu vào (vì các vector trong ma trận đầu vào $X = \{x_1, \ldots, x_n\}$ khi qua lớp FFN không tương tác với nhau nên ta sẽ trình bày tác động của lớp này lên một vector đại diện cho các vector còn lại trong ma trận đầu vào); $W_1 \in \mathbb{R}^{d_{model} \times d_{ff}}$, $W_2 \in \mathbb{R}^{d_{ff} \times d_{model}}$ là các ma trận có hệ số học được; b_1, b_2 là bias của lớp.

ReLU là một một hàm phi tuyến được định nghĩa như sau:

$$\text{ReLU(x)} = \max(0, \text{x})$$

Nhờ đó các giá trị x âm sẽ trả về 0, các giá trị dương trả về chính nó. Rõ ràng ReLU không là một hàm tuyến tính vì nó không có dạng $mx + b$. Thay vào đó, ReLU có "bước nhảy" tại x = 0. Việc làm các giá trị x thay đổi đột về giá trị 0 khiến hàm ReLU không còn sự liên kết tuyến tính giữa đầu vào và đầu ra.

Thực tế, input $X \in \mathbb{R}^{n \times d_{model}}$ của FFN là output của của lớp trước (MultiHead Attention), Position-wise FFN thực hiện biến đổi mỗi token trong n token độc lập của chuỗi qua việc chia sẻ cùng một trọng số W_1, W_2 cho toàn bộ chuỗi.

Mục đích chính của FFN trong Transformer là biến đổi đầu ra từ cơ chế self-attention theo một cách phi tuyến tính (riêng biệt, không tương tác nhau). Như ta thấy, mặc dù cơ chế attention rất tốt trong việc xử lí thông tin ở mức độ tương quan giữa các token, nhưng các biểu diễn output của lớp MultiHead Attention vẫn chủ yếu mang tính tuyến tính và không đủ để mô hình học được các mối quan hệ phức tạp và phi tuyến tính trong các tác vụ như dịch máy, generate văn bản,... Do đó, ta sử dụng FFN sau lớp MultiHead Attention giúp mô hình học được các đặc trưng phức tạp hơn nhờ vào việc biến đổi thông tin biểu diễn của các token này. Như vậy, việc kết hợp giữa self-attention và FFN giúp mô hình có thể vừa nắm bắt được sự phụ thuộc giữa các token trong chuỗi (qua cơ chế attention) và vừa thực hiện các biến đổi biểu diễn của token (qua FFN).

Giải thích Data Flow

Giả sử ta có ma trận đầu vào $X \in \mathbb{R}^{n \times d_{model}}$ gồm các vector dòng của từng token sau khi đi qua lớp MultiHead Attention: $X = \{x_1, \ldots, x_n\}$ với $x_i \in \mathbb{R}^{d_{model}}$. Khi này từng các x_i song song đi qua lớp fully connected layer thứ nhất (Linear1) của FFN. Các vector này sau đi qua lớp Linear1 sẽ trở thành một vector biểu diễn với $x'_i \in \mathbb{R}^{d_{ff}}$.

$$x'_i = f(W_1, b_1) = x_i \cdot W_1 + b_1$$

với $W_1 \in \mathbb{R}^{d_{model} \times d_{ff}}$ và $b_1 \in \mathbb{R}^{d_{ff}}$.

Sau đó x'_i được đưa qua hàm ReLU biến các giá trị âm thành dương với $x''_i = \text{ReLU}(x'_i) = \max(0, x'_i)$.

Cuối cùng đầu ra x''_i được chuyển qua lớp fully connect layer thứ hai (linear2):

$$x'''_i = f(W_2, b_2) = x''_i \cdot W_2 + b_2$$

với $W_2 \in \mathbb{R}^{d_{ff} \times d_{model}}$ và $b_2 \in \mathbb{R}^{d_{model}}$.

4.2.4 Add and Norm

Tổng quan

Đây chính là Residual Connection và LayerNormalization. Công thức của lớp này:

$$\text{X} = \text{LayerNorm}(\text{X} + \text{SubLayer}(\text{X}))$$

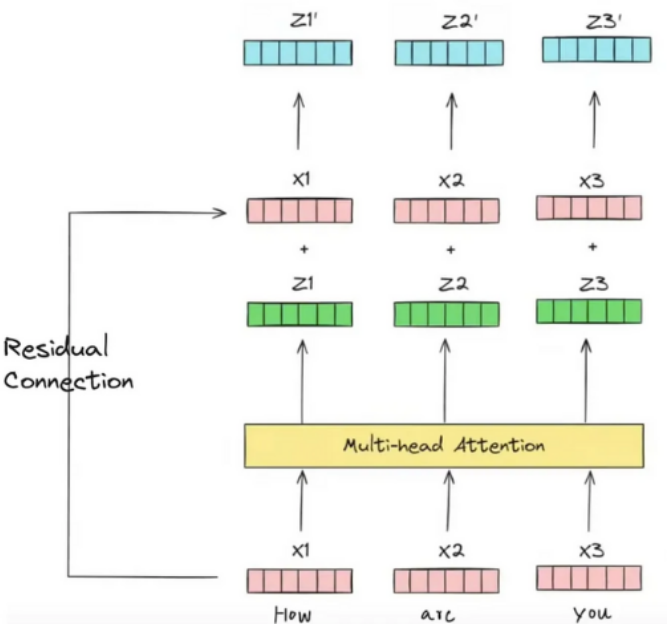
với SubLayer(X) có thể MultiHead(X) hoặc FFN(X).

Trong một residual connection, thay vì sử dụng trực tiếp output y qua một SubLayer nào đó cho SubLayer tiếp theo, thì một phần đầu vào x sẽ được cộng trực tiếp vào output của SubLayer đó. Cụ thể, output y của residual connection được tính bằng:

$$y = F(x) + x$$

Khi đó, x được định nghĩa là input của một SubLayer, $F(x)$ là hàm Sublayer được sử dụng ví dụ như MultiHead Attention SubLayer, Feed Forward Network SubLayer. Phép cộng $F(x) + x$ giữ lại thông tin ban đầu và thông tin đã bị biến đổi qua F.

Trong deep learning, khi đầu ra của lớp trước được sử dụng làm đầu vào cho lớp sau và quá trình này lặp lại qua nhiều lớp, các giá trị gradient trong quá trình lan truyền ngược



Hình 7: Residual connection

(backpropagation) có thể trở nên rất nhỏ. Điều này dẫn đến hiện tượng gradient biến mất (vanishing gradient). Tuy nhiên, nhờ có kết nối dư (residual connection), một phần của đầu vào x được giữ lại, giúp giảm thiểu việc gradient bị thu nhỏ, từ đó làm cho quá trình học nhanh hơn và hiệu quả hơn. (gradient càng nhỏ học càng chậm vì máy tính phải tính toán các số cực kì nhỏ, hơn nữa gradient càng nhỏ thì việc thay đổi trong trọng số mô hình không đủ lớn để mô hình cải thiện hiệu suất - có thể hiểu qua khi đặt learning rate khởi tạo quá nhỏ khiến mô hình rơi về điểm cực trị cũng cực kì lâu).

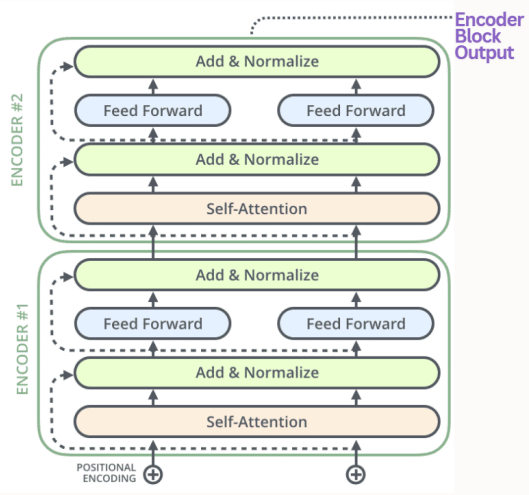
Layer Normalization chuẩn hóa giá trị đầu vào của mỗi sample theo trung bình và độ lệch chuẩn của các giá trị trong lớp đó. Công thức của Layer Normalization là:

$$\hat{x_i} = \frac{x_i - \mu}{\sigma} \cdot \gamma + \beta$$

Trong đó, x_i là giá trị đầu vào tại vị trí thứ i, μ, σ là trung bình và độ lệch chuẩn của các giá trị trong lớp, γ, β là các learnable parameters dùng để điều chỉnh các giá trị sau khi chuẩn hóa.

Do chỉ chuẩn hóa các giá trị trong một lớp nên Layer Normalization không ảnh hưởng bởi kích thước batch size mà chỉ quan tâm đến từng mẫu khi training. Việc chuẩn hóa cũng giảm vanishing gradient hay exploding gradient trong quá trình huấn luyện.

Giải thích Data Flow: Giả sử ta có ma trận đầu vào $X \in \mathbb{R}^{n \times d_{model}}$ gồm các vector là dòng là các vector đã được học trước đó, ta tính SubLayer(X) sẽ cho ra ΔX và ta sẽ cập nhật $X := X + \Delta X$. Khi này, rõ ràng $\Delta X \in \mathbb{R}^{n \times d_{model}}$, sau khi cập nhật thì kích thước của $X \in \mathbb{R}^{n \times d_{model}}$. Cuối cùng là chuẩn hóa X qua LayerNorm(X) vừa được cập nhật, ma trận sau khi tính toán không đổi kích thước.



Hình 8: Encoder Block

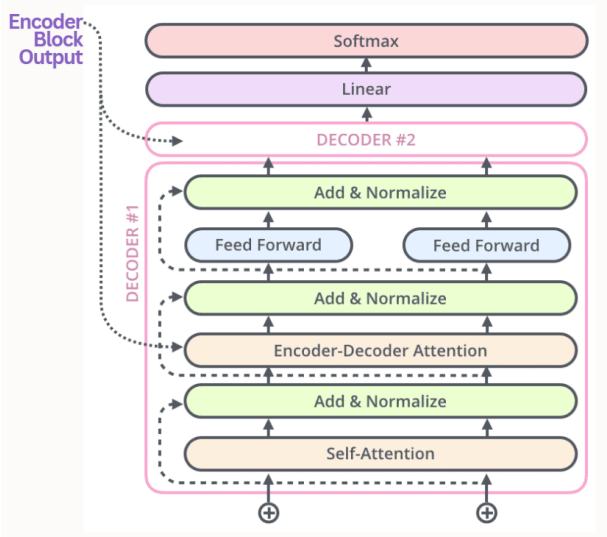
4.2.5 Encoder Block

Tổng quan

Block Encoder bao gồm N Encoder layers (một ngăn xếp Encoder với N encoder xếp chồng lên nhau, mỗi Encoder layer có một trọng số riêng biệt). Ngoại trừ InputEmbedding được đưa vào Encoder Block ban đầu, output của một lớp Encoder phía trước sẽ làm input cho lớp tiếp theo liền sau nó, và khi qua hết N layer của block, ta thu được ma trận đầu ra cuối cùng. Ma trận output của Encoder Block này sẽ được đưa vào Decoder Block tại Cross MultiHead Attention SubLayer này để làm keys, values cho Sublayer này.
Bên trong mỗi Encoder layer của block Encoder bao gồm 2 lớp con (sublayer) chính, lớp con thứ nhất sẽ là Multi-head Attention - Add & Norm, lớp con thứ hai là Feed Forward - Add & Norm, cụ thể Multi-Head → Add & Norm → Feed Forward → Add & Norm.

Data Flow

Như đã giới thiệu trước đó trong Data Flow của Kiến trúc Transformer, block Encoder là block đảm nhiệm xử lý dữ liệu input (câu muốn dịch), giúp cho mô hình học từ input. Chính vì vậy, ở layer đầu tiên của block Encoder, gọi $X \in \mathbb{R}^{n \times d_{model}}$ là ma trận input đã được Embedding và Positional Encoding cho một dữ liệu (sequence) nào đó, X sẽ là input đầu tiên cho layer này, hay cụ thể X sẽ được cập nhật lại theo $X := \text{LayerNorm}(X + \text{MultiHead}(X, X, X))$ và sau đó lại được cập nhật theo $X := \text{LayerNorm}(X + FFN(X))$. Đây chính là ouptut của layer thứ nhất, output của layer này sẽ tiếp tục làm input cho layer thứ hai, tương tự như thế cho lần lượt tất cả các layer còn lại của block Encoder để cho ra ma trận cuối cùng, gọi là Encoder Output.
Encoder Output được xem là sự trao đổi thông tin giữa các vector embedding đầu vào, đây là tổng hợp thông tin của dữ liệu đầu vào mà mô hình thu được, Encoder Output là input cho tất cả các Decoder Layer trong Decoder Block như là các queries và keys cho Cross MultiHead Attention Sublayer của từng Decoder Layer.



Hình 9: Decoder Block

4.2.6 Decoder Block

Tổng quan

Cũng tương tự như block Encoder, block Decoder cũng có N Decoder Layers, mỗi lớp gồm có 3 lớp sublayer, lớp con thứ nhất là Masked Multi-head Attention - Add & Norm, lớp con thứ hai là Cross Multi-head Attention - Add & Norm, lớp con thứ ba là Feed Forward - Add & Norm.

Data Flow

Giả sử input decoder s là sequence đầu vào cho lớp Decoder, đầu tiên, chuỗi sẽ được Embedding, sau đó là Positional Encoding, cho ra ma trận X. X sẽ bắt đầu đi qua sublayer đầu tiên của layer đầu tiên và được cập nhật thành $X := \text{LayerNorm}(X + \text{MaskedMultiHead}(X, X, X))$, ta sẽ biến ma trận Attention thành dạng tam giác dưới (masking), (các vị trí phía trên đường chéo chính được gán thành $-\infty$), từ đó dữ liệu dự đoán tại token thứ i chỉ được dựa nó và các token trước nó mà không phụ thuộc vào các token phía sau, tại đây X là ma trận được học từ input của Decoder, ta sẽ lấy $Q = X$ và $K = V = E_{out} = \text{EncoderOutput}$ để làm đầu vào cho lớp Cross MultiHead Attention Sublayer tiếp theo, cụ thể thay vì dù X làm input cho cả Q, V, K (như $\text{MultiHead}(X, X, X)$ thông thường) thì tại đây ta sẽ lấy $\text{MultiHead}(X, E_{out}, E_{out})$ và cập nhật X, $X := \text{LayerNorm}(X + \text{MultiHead}(\{X, E_{out}, E_{out}\})$ với E_{out} là ma trận đầu ra của block Encoder. Sau cùng ta sẽ cập nhật $X := \text{LayerNorm}(X + \text{FFN}(X))$, đây là output của 1 layer trong block Decoder, output này sẽ làm input cho lớp tiếp theo, và tương tự khi đi qua lớp con thứ 2 của layer tiếp theo, dữ liệu vẫn sẽ lấy Keys và Values từ E_{out}, chỉ có queries là output của lớp con liền trước nó.

4.3 Cách cập nhật của mô hình Transformer

Khi data được học qua 2 lớp Encoder và Decoder, output của lớp Decoder ($D_{out} \in \mathbb{R}^{n \times d_{model}}$) sẽ được biến đổi tuyến tính và cập nhật thành $logits := D_{out}W + b$ với $W \in \mathbb{R}^{d_{model} \times vocab\_size}$

(Ta gọi đầu ra sau khi biến đổi tuyến tính cho D_{out} là **logits**). Sau đó, ta sẽ dùng hàm Softmax cho mỗi dòng của *logits* để tạo hành các vector phân phối.

Hàm loss mà mô hình Transformer sử dụng là CrossEntropy. Công thức của hàm loss CrossEntorpy cho một phân phối xác suất bất kì là: $CE(\{p_1, ..., p_V\}) = -\log(p_i)$ với i là vị trí đúng của từ theo thứ tự các từ trong Từ điển mà mô hình quy ước, V là vocab_size hay kích thước của Từ điển.
Đối với một sequence thì hàm loss sẽ được tính như sau:

$$Loss(seq) = \frac{1}{n}\sum_{i=1}^{n} -\log(z_{i,t_i})$$

với n là số token có trong ground truth và z_{i,t_i} là xác suất mà mô hình tính được để token dự đoán tại vị trí thứ i là token đúng với ground truth tại vị trí thứ i. Từ đó loss trên một batch là:

$$Loss(B_i) = \frac{\sum_{i=1}^{B} n_i Loss(z_i)}{\sum_{i=1}^{B} n_i}$$

với n_i là số token có trong ground truth của sequence thứ i; B là số sequence có trong một batch; z_i là các phân phối cho các từ trong Từ điển.
Và cuối cùng ta sẽ tính hàm Loss cho tất cả các batch:

$$Loss = \frac{\sum_{i=1}^{T}\sum_{j=1}^{B_i} n_{ij} Loss(seq_{ij})}{\sum_{i=1}^{T}\sum_{j=1}^{B_i} n_{ij}}$$

với T là tổng số batch.
Ta sẽ dùng các thuật toán để tìm cực trị cho hàm Loss, mô hình trong paper dùng Adam, và dùng lan truyền ngược (Back Propagation) để cập nhật hệ số.

4.4 Giải thuật tham lam - Greedy Decode

Tổng quan:

Đây là một hướng cơ bản để mô hình có thể đưa ra dự đoán cho chuỗi đầu ra.
Trước tiên, mô hình khởi tạo chuỗi đầu vào cho Decoder Block là "<sos>", từ đây mô hình sẽ bắt đầu dự đoán từ token $< sos >$. Bước đầu tiên, mô hình dự đoán xác suất cho tất cả các từ trong Từ điển (Từ điển của Output Tokenizer) và chọn từ có xác suất cao nhất. Sau đó, mô hình tiếp tục dự đoán cho các từ tiếp theo dựa trên từ vừa mới được dự đoán và dựa trên các từ trước đó, ta tiếp tục chọn từ có xác suất cao nhất. Quá trình này lặp lại cho đến khi gặp kí tự kết thúc $< eos >$ hoặc độ dài chuỗi đạt giá trị n (n là số token tối đa trong chuỗi đầu ra)

Giải thích:

Từ một chuỗi input (input sequence) s, ta tạo ma trận InputEmbedding $X \in \mathbb{R}^{n \times d_{model}}$ và đưa vào block Encoder, như vậy $E_{out} = Encoder(X) \in \mathbb{R}^{n \times d_{model}}$. Còn đối với chuỗi dự đoán khởi tạo sẽ là $predict\_seq = \{< sos >\}$. Gọi $Z \in \mathbb{R}^{1 \times d_{model}}$ là ma trận OutputEmbedding của $predict\_seq$. Ta sẽ đi tính $D_{out} = \text{Decoder}(Z, E\_out)$ và thông qua lớp biến đổi tuyến tính linear tạo thành $logits = \text{linear}(D_{out}) \in \mathbb{R}^{1 \times d_{vocab\_size}}$, sau đó dùng softmax cho logits ta được $Prob = \text{softmax}(logits) \in \mathbb{R}^{1 \times vocab\_size}$ với vocab_size là số lượng từ trong Từ điển của Tokenizer Output. Khi đó, ta chọn từ có xác suất cao nhất trong Từ điển nhờ vào tìm giá trị max của $Prob$, token đó sẽ được thêm vào chuỗi dự đoán. Giả sử từ được chọn là t_1 thì

$predict\_seq = \{< sos >, t_1\}$. Lúc này, ta lại tiếp tục sử dụng $predict\_seq$ vừa được cập nhật để tiếp tục tìm token tiếp theo, khi này $Z' \in \mathbb{R}^{2 \times d_{model}}$ và $D'_{out} = \text{Decoder}(Z', E_{out})$. Tiếp theo ta đưa D_{out} vào lớp linear cuối cùng để có được $logits' \in \mathbb{R}^{2 \times vocab\_size}$. Để có được dự đoán của từ tiếp theo thực tế ta chỉ cần tính softmax cho hàng cuối của $logits'$. Như vậy ta lại có được token dự đoán cho từ tiếp theo. Quá trình này lặp lại cho đến khi token dự đoán mới là $< eos >$ hoặc khi $predict\_seq$ đạt kích thước tối đa n.

Nhược điểm

Greedy decoding thực chất chỉ dựa trên lựa chọn tốt nhất tại mỗi bước, nhưng điều này có thể dẫn đến kết quả không tối ưu. Ví dụ, mô hình có thể chọn từ có xác suất cao nhất tại mỗi bước, nhưng chuỗi kết quả cuối cùng có thể không phải là chuỗi tối ưu nếu xét toàn bộ chuỗi.

5 Ví dụ trực quan cho mô hình Transformer

5.1 Học từ dữ liệu

Ta đang xét cách mô hình Transformer học từ dữ liệu. Giả sử trong Data ta dùng để train cho mô hình.

ENCODER

Ta có chuỗi input seq_{input}, ta chia chuỗi input này thành m tokens và ánh xạ mỗi token cho các vector embedding tương ứng để thu được $E_{encoder} \in \mathbb{R}^{m \times d\_model}$. Sau đó, ta dựng ma trận input $X_{encoder} = \sqrt{d\_model} \cdot E_{encoder} + PE$ trong đó PE là ma trận Positional Encoding.

Tiếp đến, ma trận X này sẽ được làm input cho lớp Encoder. Tại mỗi lớp của block Encoder, đầu tiên chúng sẽ đi qua sublayer đầu tiên là Multi-Head Attention - Add & Norm để thu được $X_1 = LayerNorm(X + MultiHead(X, X, X))$ và sau đó chúng sẽ qua sub-layer thứ hai là Feed Forward Network - Add & Norm và thu được $X_2 = LayerNorm(X_1 + FFN(X_1))$ trong đó thì LayerNorm() của một ma trận là ta sẽ chuẩn hóa lại từng vector dòng của ma trận đó.

Sau khi qua một lớp của khối Encoder như trên, ta sẽ thu được ma trận $X_2 \in \mathbb{R}^{m \times d\_model}$. Output của lớp này sẽ làm Input cho lớp kế tiếp và cứ tiếp tục như vậy cho đến khi chúng đi qua hết tất cả layers của Encoder, thu được ma trận cuối cùng là $D_{encoder} \in \mathbb{R}^{m \times d\_model}$.

DECODER

Ta xét đến chuỗi output seq_{output} tương ứng với chuỗi input trên, ta chia chuỗi output thành n tokens, ánh xạ mỗi token cho các vector embedding tương ứng để thu được ma trận $E_{decoder} \in \mathbb{R}^{n \times d\_model}$. Sau đó, ta cũng dựng ma trận input $X_{decoder} = \sqrt{d\_model} \cdot E_{decoder} + PE$ trong đó PE là ma trận Positional Encoding.

Tiếp đến, ma trận này sẽ được làm input cho block Decoder. Tại mỗi lớp của block Decoder, đầu tiên thì ma trận $X_{decoder}$ sẽ đi qua sub-layer thứ nhất là Masked Multi-Head Attention để thu được $X_1 = LayerNorm(X + MaskedMultiHead(X, X, X))$. Sau đó, chúng sẽ được đi qua sub-layer thứ hai, thu được $X_2 = LayerNorm(X_1 + MultiHead(X_1, D_{encoder}, D_{encoder}))$, và cuối cùng thì chúng sẽ qua sub-layer Feed Forward Network - Add & Norm thu được $X_3 = LayerNorm(X_2 + FFN(X_2))$.

Sau khi qua một lớp của Decoder như trên, ta sẽ lấy Output của lớp trước làm input cho lớp sau và tiếp tục cho đến khi qua hết tất cả các lớp của block Decoder, thu được $D_{decoder} \in \mathbb{R}^{n \times d\_model}$. Sau khi thu được $D_{decoder} \in \mathbb{R}^{n \times d\_model}$, ta sẽ biến đổi tuyến tính bằng cách nhân cho một ma trận $W \in \mathbb{R}^{d\_model \times d\_vocab\_size}$ kèm theo bias, thu được ma trận logits $logits \in \mathbb{R}^{n \times d\_vocab\_size}$ và ta áp dụng hàm softmax cho từng dòng của ma trận logits để thu được mỗi dòng của logits là một phân phối xác suất cho từng token của chuỗi seq_{out}. Tới đây, ta sẽ dùng ma trận logits (đã qua softmax) để tính hàm Loss và cập nhật tham số cho mô hình thông qua các kĩ thuật

Gradient Descent và Back Propagation.

5.2 Đoán nghĩa của chuỗi

Sau khi mô hình được học từ dữ liệu, mô hình sẽ có thể dự đoán output sequence mà không cần có output mẫu. Nhờ có cơ chế Masked MultiHead của Decoder, các token chưa được sinh ra sẽ có thông tin suy luận được từ các token đã xuất hiện trước đó và dựa trên xác suất xuất hiện. Chính vì vậy mô hình có thể đoán từng kí tự, có nhiều thuật toán cho việc này như (beam search, greedy decode, ...).
Thực tế, ban đầu chuỗi output $seq_{out} = \{< sos >\}$, chỉ có kí tự $< sos >$ đại diện cho token khởi đầu, sau đó chuỗi input seq_{input} ta nhập sẽ được đi qua block Encoder. Còn chuỗi output seq_{output} sẽ được chạy qua block Decoder và cứ mỗi lần chạy qua như vậy, ta sẽ luôn lấy vector dòng cuối cùng của $D_{decoder}$ làm phân phối để tìm token mới cho chuỗi seq_{out} (token được chọn là token có xác suất lớn nhất), và cứ tiếp tục như vậy cho đến khi token $< eos >$ được thêm vào seq_{out} hay chuỗi seq_{out} đạt được số kí tự tối đa cho phép (hằng số).

6 Thực nghiệm

6.1 Dữ liệu

Trong bài báo "Attention all you need", Google sử dụng data WMT 2014 English-German có khoảng 4.5 triệu mẫu (mỗi mẫu gồm 2 câu tiếng Anh và tiếng Đức) và WMT 2014 English-French gồm 36 triệu câu để thực nghiệm mô hình Transformer. Nhóm cũng nghiên cứu các dataset tương tự nhưng cho tác vụ dịch tiếng Anh sang tiếng Việt. Tuy nhiên, các dataset lớn được đề cập trong bài báo cần rất nhiều tài nguyên GPU, các dữ liệu như ncduy/mt-en-vi được host trên HuggingFace cũng chứa 2.88 triệu mẫu, còn PhoMT của VinAI chứa khoảng 3 triệu mẫu. Nhóm hướng đến tìm một dataset nhỏ hơn (khoảng vài trăm nghìn mẫu) để phù hợp với mục đích nghiên cứu của nhóm - hiểu về mô hình Transformer và cơ chế Attention của nó. Vì thế, nhóm đã sử dụng "hungnm/englishvietnamese-translation" trên kaggle (khoảng 250 ngàn mẫu) với các câu tiếng anh cơ bản, không quá dài để dịch sang tiếng việt. Nhóm đạt hiệu suất trên toàn bộ dataset với BLEU score là 0.6672, trên tập validation là 0.6782.

6.2 Tiền xử lí dữ liệu

Nhóm chia dữ liệu với tỉ lệ 9:1 gồm 90% cho training và sử dụng 10% còn lại cho validation.

Dữ liệu đưa vào mô hình Transformer ngoài việc phải qua bước tiến xử lí cơ bản như loại bỏ các khoảng trắng thừa, loại bỏ dấu câu, loại bỏ các mẫu dữ liệu có chứa NaN,... thì bước quan trong nhất trong việc chuẩn bị dữ liệu là đảm bảo dữ liệu được embedding đúng cách.

Một mẫu data khi dịch cần có câu input và output mong muốn. Giả sử input là "how are you" và output mong muốn là "bạn khỏe không", chọn n token tối đa là 6 và d_{model} tùy ý thì mẫu data đó trước khi đưa vào mô hình cần:

- Input Embedding (Encoder Input) cho câu "<sos> how are you <eos> <pad>"
- Output Embedding, Shifted Right (Decoder Output) cho câu "<sos> bạn khỏe không <pad> <pad>". Do shifted right nên output embedding không chứa kí tự "<eos>".
- Label, Shifted Left (được sử dụng cho tính loss của mô hình), câu được encode là "bạn khỏe không <eos> <pad> <pad>". Shifted left nên label không chứa kí tự "<sos>".

Rõ ràng ta thấy các kí tự <pad> được thêm vào sau khi đã xếp vị trí <sos>/<eos> vào cho các câu và chỉ được sử dụng để làm đầy ma trận đầu vào để phù hợp kích thước $X \in \mathbb{R}^{n \times d_{model}}$. Ngoài ra, một mẫu data cũng cần:

- Input Mask: ma trận mask $\in \mathbb{R}^{n \times n}$ che các kí tự "<pad>" khi thực hiện tính attention weights trong Encoder Block.
- Output Mask: ma trận mask $\in \mathbb{R}^{n \times n}$ che các từ phía sau nó và các kí tự "<pad>" khi thực hiện tính attention weights trong Mask MultiHead Attention của Decoder Layer.

Ngoại trừ lưu các biểu diễn số của input và output đầu vào thì một mẫu data cũng cần lưu các text gốc để dễ dàng hơn cho việc đánh giá (evaluate) mô hình bằng điểm BLEU score.

6.3 Setup mô hình Transformer

Trong bài báo gốc, Google sử dụng $P_{drop} = 0.1, d_{model} = 512, d_{ff} = 2048, h = 8, N = 6$. Do mô hình nhóm train trên dữ liệu nhỏ hơn, nên nhóm đã giảm các parameter trong mô hình để giảm thời gian tính toán và tránh overfit (dữ liệu nhỏ mà mô hình quá phức tạp) như sau:

- Batch size: Số lượng mẫu trong mỗi batch được sử dụng trong quá trình huấn luyện. Với batchsize = 64, mỗi lần huấn luyện sẽ sử dụng 64 mẫu trong một batch để tính toán gradient và cập nhật trọng số.
- Epochs: 50, mô hình sẽ trải qua 50 lần huấn luyện trên toàn bộ dữ liệu huấn luyện. Tuy nhiên kết quả thực tế chỉ lấy 10 epoch đầu.
- Learning Rate: 3×10^{-4}, trong quá trình huấn luyện nhóm sử dụng learning rate động với giá trị khởi tạo là 3×10^{-4}, sau đó learning rate sẽ tự điều chỉnh cho phù hợp với quá trình training qua các epoch.
- P_{drop} (dropout): Xác suất loại bỏ các neuron một cách ngẫu nhiên trong mỗi một Sublayer có qui định dropout và trong mỗi lần training nhằm hạn chế overfitting. Với dropout = 0.2, một neuron có xác suất bị tắt là 20%.
- N (số layers): Số lớp của Encoder và Decoder trong mô hình Transformer. Với N = 4, mô hình sẽ sử dụng 4 lớp Encoder và 4 lớp Decoder.
- h (số heads): Số heads trong cơ chế MultiHead Attention. Với h = 4, mô hình sẽ sử dụng 4 head attention để xử lý các phần khác nhau của chuỗi đầu vào song song.
- d_{model}: 256. Đây là kích thước không gian embedding trong mô hình.
- d_{ff}: 1024, mô hình sử dụng một mạng gồm 1024 neuron trong Feed Forward SubLayer.
- n: 70. Mô hình xử lí chuỗi đầu vào có tối đa 70 token trong mỗi lần huấn luyện.

6.4 BLEU score

BLEU (Bilingual Evaluation Understudy) score là chỉ số dùng để đánh giá tác vụ machine translation nhờ tính độ chính xác (precision) và áp dụng hình phạt độ dài (brevity penalty) giữa bản dịch của mô hình (hypothesis) và bản dịch tham chiếu (reference), bản dịch tham chiếu có thể coi là groundtruth của mô hình.

Giả sử chúng ta có 1 bản hypothesis và 1 bản reference như sau:

- Reference: "there is a cat on the mat"
- Hypothesis: "the cat on the mat"

Đầu tiên, chúng ta cần đánh giá precision của hypothesis với reference. Precision trong BLEU score được tính theo n-gram (n-gram có thể hiểu là n số chuỗi con được xét trong 1 lần).

Đầu tiên ta tính cho n-gram unigram (n=1), khi xét n-gram cho 1 câu thì các từ được xét độc lập, cho dù chúng xuất hiện nhiều lần trong hypothesis và reference. Như vậy rõ ràng ta thấy, các unigram (từ đơn) là:

- Reference: ["there", "is", "a", "cat", "on", "the", "mat"]
- Hypothesis: ["the", "cat", "on", "the", "mat"]

Các matches của reference và hypothesis là "the", "cat", "on", "mat" với số lượng là 4. Vậy precision của unigram là:

$$\text{Precision(unigram)} = \frac{\text{matches}}{\text{number of unigram in hypothesis}} = 4/5 = 0.8$$

Tương tự tính n-gram bigram (n=2), các bigram là:

- Reference: ["there is", "is a", "a cat", "cat on", "on the", "the mat"]
- Hypothesis: ["the cat", "cat on", "on the", "the mat"]

Các bigram match là: "cat on", "on the", "the mat" với số lượng là 3.

$$\text{Precision(bigram)} = \frac{\text{matches}}{\text{number of bigram in hypothesis}} = 3/4 = 0.75$$

Sau đó, ta thực hiện Brevity Penalty (BP) để điều chỉnh cho các bản dịch hypothesis quá ngắn so với reference. Công thức tính BP như sau:

$$\text{BP} = \begin{cases} 1, c > r \\ e^{1-\frac{r}{c}}, c \leq r \end{cases}$$

Trong đó, c là số token của hypothesis và r là số token của reference.

Như vậy độ dài của hypothesis trong "the cat on the mat" là $c = 5$ và độ dài của referece trong "there is a cat on the mat" là $r = 7$.

Vì $c < r$ nên $\text{BP} = e^{1-\frac{7}{5}} = e^{-0.4}$

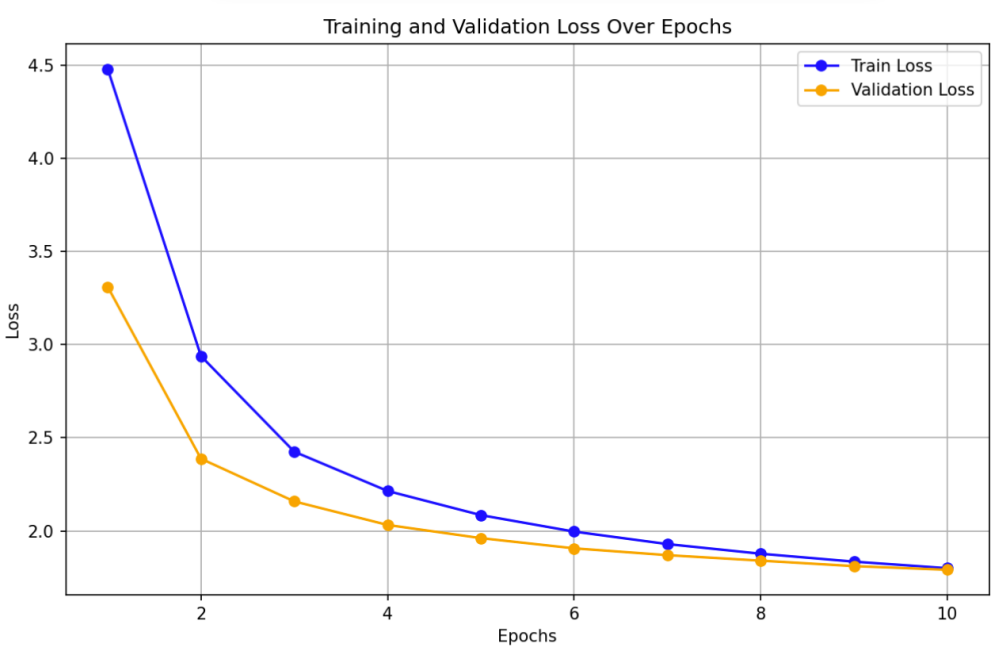
Cuối cùng, ta thực hiện tính BLEU score là trung bình của các precision cho n-gram và điều chỉnh bởi brevity penalty.

$$\text{BLEUscore} = \text{BP} \times \exp(\frac{1}{N}\sum_{n=1}^{N}\log p_n)$$

Trong đó p_n là precision cho n-gram n và N là số lượng n-gram.

Do ở đây ta tính BLEU score cho unigram và bigram nên $N = 2$. Như vậy:

$$\text{BLEUscore} = \text{BP} \times \exp(\frac{1}{2}(\log p_1 + \log p_2)) = e^{-0.4} \times \exp(\frac{1}{2}(\log 0.8 + \log 0.75)) \approx 0.5192$$



Hình 10: Train Loss và Validation Loss trên tập dữ liệu

6.5 Phân tích, đánh giá

Training Loss: Màu xanh dương đại diện cho training loss, giảm dần theo số epoch. Điều này cho thấy mô hình đang học và cải thiện dần dần trong quá trình huấn luyện. Loss huấn luyện giảm nhanh trong những epoch đầu tiên và tiếp tục giảm với tốc độ chậm hơn sau đó.

Validation Loss: Màu vàng đại diện cho validation loss. Trong giai đoạn đầu, validation loss giảm dần, tuy nhiên, nó có vẻ không giảm nhanh như training loss và có xu hướng ổn định sau vài epoch.

Sau khi train 10 epoch trong 12 tiếng, mô hình đạt hiệu suất BLEUscore với toàn bộ dataset là 0.6674 và trên riêng tập val là 0.6785.

Tuy nhiên, dataset nhỏ nên mô hình chưa thể nhận diện các tên quá unique như "Kim", "Linh", "Hiep", "Bao", ... khi nhóm test bằng cách inference model đã train. Các từ phức tạp như "menace" mô hình cũng chưa học được do không có trong từ điển Input Tokenizer hay dataset mà nhóm đã train (Beam Search, ...)

6.6 Kết luận

Từ kết quả huấn luyện và đánh giá trên tập validation, mô hình Transformer đã cho thấy sự cải thiện rõ rệt trong việc giảm mất mát (loss) trong suốt quá trình huấn luyện. Mặc dù có sự khác biệt giữa training loss và validation loss, mô hình vẫn duy trì khả năng tổng quát khá tốt khi validation loss chỉ giảm chậm lại mà chưa có dấu hiệu của overfitting.

Hiệu suất BLEU score đạt được trên toàn bộ dataset là 0.6674, trong khi trên tập validation là 0.6785, cho thấy mô hình có khả năng dịch tốt trên cả train set và validation set. Mặc dù mô hình có thể tiếp tục cải thiện trong những epoch sau, nhưng kết quả hiện tại cho thấy một hiệu suất chấp nhận được trong bài toán này.

Tuy nhiên, để có thể đưa vào thực tế và ứng dụng dịch máy, cần nhiều tài nguyên GPU hơn và cần sử dụng một dataset lớn hơn với các câu dài và phức tạp để tăng khả năng suy luận và tổng quát hóa của mô hình. Chúng ta cũng có thể tăng hiệu suất mô hình nhờ thay đổi thuật toán dự đoán câu dịch.

7 Phụ lục

7.1 Paper

Đây là đường dẫn tới paper trong báo cáo: Paper "Attention is All you need".

7.2 Slide thuyết trình

Đây là đường dẫn tới slide thuyết trình của nhóm: Slide thuyết trình .

7.3 Video thuyết trình

Đây là đường dẫn tới video thuyết trình của nhóm: Video thuyết trình .

7.4 Dữ liệu

Đây là đường dẫn tới dataset: English-Vietnamese Translation Dataset.

7.5 Source Code

Đây là đường dẫn tới source code của nhóm: Source Code cài đặt mô hình Transformer.

Tài liệu

[1] Arun Mohan, "Transformer from Scratch using PyTorch," [Online]. Available: `https://www.kaggle.com/code/arunmohan003/transformer-from-scratch-using-pytorch`

[2] Trí Tuệ Nhân Tạo, "Minh họa trực quan Transformer," [Online]. Available: `https://trituenhantao.io/tin-tuc/minh-hoa-transformer/`

[3] Huỳnh Hữu Ngôn, "English to Vietnamese Translation with Transformer," [Online]. Available: `https://www.kaggle.com/code/huhuyngun/english-to-vietnamese-with-transformer`

[4] HuggingFace Datasets, "MT EN-VI Dataset by ncduy," [Online]. Available: `https://huggingface.co/datasets/ncduy/mt-en-vi`

[5] AI Coffee, "Hiểu Transformer đơn giản qua ví dụ dịch máy," YouTube video, [Online]. Available: `https://www.youtube.com/watch?v=eMlx5fFNoYc`

[6] A. Vaswani et al., "Attention is All You Need," [Online]. Available: `https://arxiv.org/abs/1706.03762`

[7] Harvard NLP, "The Annotated Transformer," [Online]. Available: `http://nlp.seas.harvard.edu/2018/04/03/attention.html`

[8] J. Alammar, "The Illustrated Transformer," [Online]. Available: https://jalammar.github.io/illustrated-transformer/

[9] PyTorch, "Transformer Tutorial with TorchText," [Online]. Available: https://pytorch.org/tutorials/beginner/transformer_tutorial.html

[10] Hugging Face, "Transformers Library Documentation," [Online]. Available: https://huggingface.co/docs/transformers/index