Lecture 5: Transformers

Resources

Giới thiệu

Trong bài viết này, tôi sẽ đề cập đến Transformers, một trong những thành phần cốt lõi của hầu hết các ứng dụng không chỉ trong xử lý ngôn ngữ tự nhiên mà còn trong nhiều lĩnh vực khác. Tôi sẽ trình bày về Transformers như chúng đã được triển khai lần đầu vào năm 2017, cũng như những cải tiến mà các nhà nghiên cứu hiện nay thực hiện để nâng cao hiệu suất của chúng trong các mô hình ngôn ngữ hiện đại.

Reminder: Attention

Cross Attention (Bahdanau et al. 2015)

Có hai loại attention chính mà bạn cần nhớ: đầu tiên là cross attention, trong đó bạn sẽ chú ý đến một câu hoặc một chuỗi khác. Cụ thể, bạn có một chuỗi đóng vai trò là "keys" mà bạn đang chú ý đến, và một chuỗi khác đóng vai trò là "queries", tức là những gì bạn sử dụng để chú ý đến chuỗi "keys". Quá trình này diễn ra cho từng phần tử trong vector truy vấn, nơi mỗi phần tử sẽ chú ý đến từng phần tử trong vector khóa.

Self Attention (Cheng et al. 2016, Vaswani et al. 2017)

Self attention cho phép mô hình chú ý đến cùng một chuỗi dữ liệu, đảm bảo rằng các truy vấn (queries) và khóa (keys) tương ứng với cùng một chuỗi. Đây là điểm khác biệt chính giữa self attention và cross attention. Các mô hình dựa trên Transformer thường sử dụng một trong hai loại chú ý này, hoặc kết hợp cả hai. Trong bài viết này, tôi sẽ đề cập đến hai loại mô hình Transformer sử dụng cả self attention và cross attention.

Calculating Attention

Trong quá trình tính toán độ chú ý (attention), chúng ta sử dụng các vector truy vấn và vector khóa. Đối với mỗi cặp truy vấn, chúng ta sẽ tính toán trọng số giữa chúng. Sau đó, chúng ta chuẩn hóa các trọng số này bằng cách sử dụng hàm softmax để đảm bảo rằng tổng trọng số bằng 1 và nằm trong khoảng từ 0 đến 1. Dựa trên các trọng số chú ý này, chúng ta nhân chúng với các vector giá trị để thu được một vector cuối cùng. Kết quả là mỗi vector truy vấn sẽ tạo ra một vector giá trị duy nhất.

Transformers

Attention is All You Need (Vaswani et al. 2017)

Bây giờ, chúng ta sẽ khám phá cách mà mô hình Transformer hoạt động, được giới thiệu trong bài báo "Attention is All You Need" của Vaswani và các cộng sự vào năm 2017. Ngay khi bài báo được công bố, tôi đã nhận thấy rằng đây sẽ là một bước đột phá lớn trong lĩnh vực xử lý

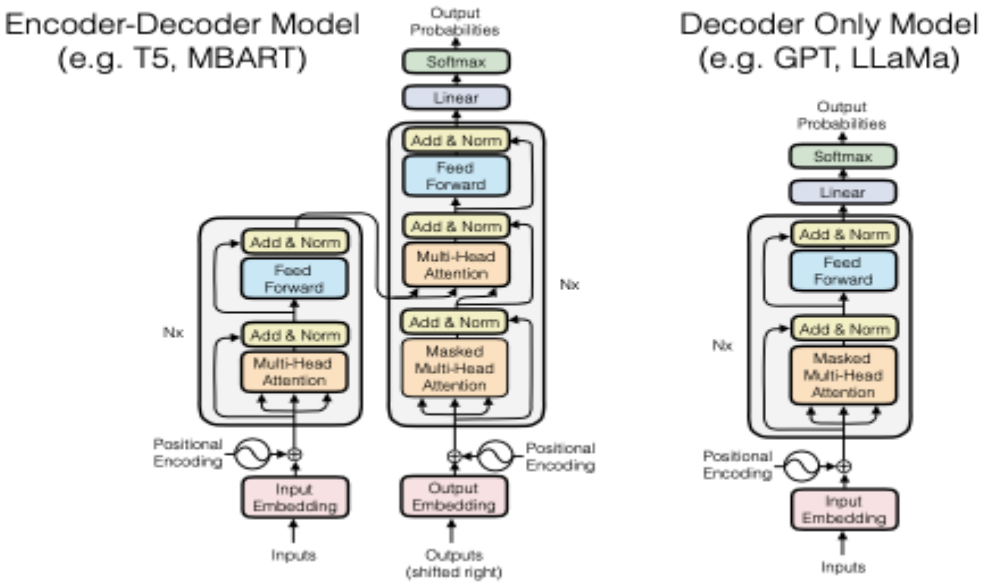
ngôn ngữ tự nhiên. Mô hình Transformer là một mô hình chuỗi, có khả năng tạo ra các chuỗi dựa hoàn toàn vào cơ chế attention, khác biệt hoàn toàn so với các mô hình trước đó, thường sử dụng RNN làm bộ mã hóa.

Trước đây, các mô hình thường sử dụng RNN cho cả phần mã hóa và giải mã, chỉ áp dụng attention cho phần cross attention. Tuy nhiên, Transformer đã loại bỏ hoàn toàn RNN trong việc mô hình hóa chuỗi và thay thế bằng self-attention. Điều này giải thích cho tên gọi "Attention is All You Need". Khi ra mắt, Transformer đã cho kết quả mạnh mẽ trong dịch máy và hiện nay, nó đã chứng minh hiệu quả trên nhiều tác vụ khác nhau.

Một điểm quan trọng khác là Transformer rất nhanh và chỉ bao gồm các phép nhân ma trận. Điều này rất quan trọng vì RNN thường bị hạn chế bởi việc phải chờ đợi kết quả từ trạng thái trước đó trước khi tính toán trạng thái tiếp theo. Với Transformer, điều này không còn là vấn đề, giúp tăng tốc độ xử lý. Tôi cho rằng lý do chính khiến Transformer trở nên phổ biến không chỉ vì nó là một phương pháp mô hình hóa tốt hơn, mà chủ yếu là do tốc độ xử lý nhanh chóng của nó.

Two Types of Transformers

Có hai loại kiến trúc Transformer: encoder-decoder (mã hoá-giải mã) và decoder-only (chỉ giải mã). Các mô hình như T5 và BART sử dụng kiến trúc encoder-decoder, trong khi các mô hình như GPT và LLaMA chủ yếu dựa vào kiến trúc decoder-only.



Kiến trúc của mô hình decoder-only chỉ bao gồm một khối duy nhất, trong khi mô hình encoder-decoder có cả khối mã hóa và giải mã. Cấu trúc chung của Transformer bao gồm các thành phần như embedding đầu vào, positional encodings, các khối multi-head attention và các khối feed-forward. Các khối multi-head attention thực hiện chức năng chú ý, trong khi các khối feed-forward kết hợp các đặc trưng đã được tính toán.

Mô hình encoder-decoder thường được sử dụng trong các tác vụ có cấu trúc đầu vào - đầu ra rõ ràng, chẳng hạn như tóm tắt văn bản hoặc dịch ngôn ngữ. Tuy nhiên, trong các ứng dụng như chatbot, việc xác định đầu vào và đầu ra trở nên phức tạp hơn. Mô hình decoder-only cho phép chúng ta xem tất cả như một chuỗi dài, giúp đơn giản hóa quá trình xử lý.

Một lý do khác khiến mô hình decoder-only trở nên phổ biến là tính đơn giản của nó. Với ít thành phần hơn, chúng ta có thể dễ dàng điều chỉnh kích thước của các lớp hoặc số lượng lớp mà không làm tăng quá nhiều số lượng tham số.

Cuối cùng, mục tiêu của bài viết này là giúp bạn hiểu rõ hơn về các thành phần cơ bản của Transformer và những thay đổi mà LLaMA thực hiện so với kiến trúc gốc, cũng như tầm quan trọng của những thay đổi này.

Core Transformer Concepts

Trong bài này, chúng ta sẽ nói về những khái niệm cốt lõi của Transformer, bao gồm:

- Positional encodings
- Multi-headed attention
- Masked attention
- Residual + layer normalization
- Feed-forward layer

Review: Inputs and Embeddings

Đầu vào thường được chia thành các subwords, như đã thảo luận trước đó. Các embedding vector thường chỉ cần được tra cứu, giống như trong các mô hình trước đây. Tuy nhiên, một điểm nổi bật của các mô hình dựa trên Transformer là chúng đã chuẩn hóa việc phân đoạn subword, và hiện nay, hầu hết các mô hình Transformer lớn đều áp dụng phương pháp này.

Multi-head Attention

Intuition for Multi-heads

Tiếp theo, chúng ta sẽ khám phá khái niệm "multi-head attention", một trong những đổi mới quan trọng trong bài báo nghiên cứu về Transformer. Ý tưởng cơ bản của multi-head attention là thông tin từ các phần khác nhau của câu hoặc chuỗi mà bạn đang mô hình hóa có thể hữu ích theo nhiều cách khác nhau. Nếu chỉ sử dụng một đầu attention duy nhất, bạn sẽ phải đưa ra những quyết định khó khăn về phần nào của câu cần chú ý.

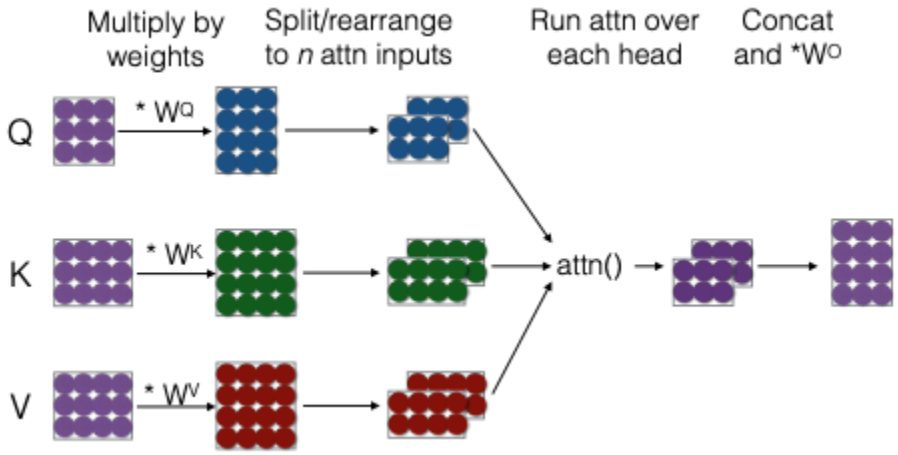
Ví dụ, với từ "run", chúng ta có bốn cách sử dụng khác nhau: "run a business" và "physically run" là hai động từ, trong khi "running a staffing" và "making it run" là hai danh từ. Sự khác biệt về nghĩa giữa các cách sử dụng này rất rõ ràng, và nếu dịch sang các ngôn ngữ khác, chúng sẽ có những bản dịch khác nhau do ý nghĩa khác nhau của chúng.

Để phân biệt các nghĩa khác nhau, thông tin ngữ pháp thường có thể được xác định từ ngữ cảnh gần kề. Chẳng hạn, nếu có một danh từ ở bên trái, điều đó thường có nghĩa là từ tiếp theo sẽ là một động từ. Ngược lại, nếu có một mạo từ ở bên trái, từ tiếp theo có khả năng cao sẽ là một danh từ hoặc tính từ. Tuy nhiên, để phân biệt nghĩa, bạn cần xem xét ngữ cảnh xa hơn.

Một điều thú vị là khi học các vector embedding từ, bạn có thể chỉ xem xét ngữ cảnh gần để học các vector ngữ pháp, hoặc chỉ xem ngữ cảnh xa để học các vector nghĩa. Điều này cho thấy rằng các phần khác nhau của ngữ cảnh có thể hữu ích cho các mục đích khác nhau, và đó chính là mục tiêu của multi-head attention: cho phép bạn xem xét nhiều phần của ngữ cảnh mà không phải lựa chọn giữa chúng.

Multi-head Attention Concept

Đầu tiên, chúng ta có một tập hợp các vector truy vấn và vector khóa. Trong ví dụ này, chúng ta chỉ có ba vector truy vấn và bốn vector khóa, điều này hoàn toàn hợp lệ. Số lượng vector truy vấn có thể khác với số lượng vector khóa và vector giá trị. Một trường hợp điển hình là khi chúng ta sử dụng masked attention, hoặc trong quá trình giải mã (decoding) khi có một chuỗi ngắn đang được chú ý đến một chuỗi dài hơn.



Khi thực hiện multi-headed attention, bước đầu tiên là nhân các vector đầu vào với các ma trận trọng số. Chúng ta có ba ma trận trọng số chính: ma trận truy vấn, ma trận khóa và ma trận giá trị. Sau khi nhân, chúng ta sẽ tách và sắp xếp các vector này thành nhiều đầu vào attention. Ví dụ, nếu chúng ta có hai đầu attention, mỗi đầu sẽ có vector kích thước hai.

Một điểm quan trọng là trong thực tế, chúng ta thường thực hiện phép nhân ma trận lớn một lần thay vì từng phần riêng lẻ để tối ưu hóa hiệu suất. Điều này giúp giảm thiểu thời gian tính toán và tăng tốc độ xử lý.

Khi chúng ta chạy attention trên mỗi đầu, chúng ta sẽ tính toán vector attention bằng cách sử dụng các vector truy vấn và khóa, sau đó nhân các vector giá trị với vector attention để có được kết quả cuối cùng. Kết quả này sẽ có số cột tương ứng với số cột trong ma trận truy vấn, và sau đó chúng ta sẽ nối các kết quả lại với nhau. Cuối cùng, chúng ta sẽ thực hiện một phép nhân ma trận cuối cùng để hoàn thiện quá trình.

$$MultiHead(Q,\ K,\ V) = Concat\left(head_1,\ \ldots,\ head_h\right)W^O$$

$$where\ head_i = Attention\left(QW_i^Q,\ KW_i^K,\ VW_i^V\right)$$

Code Example

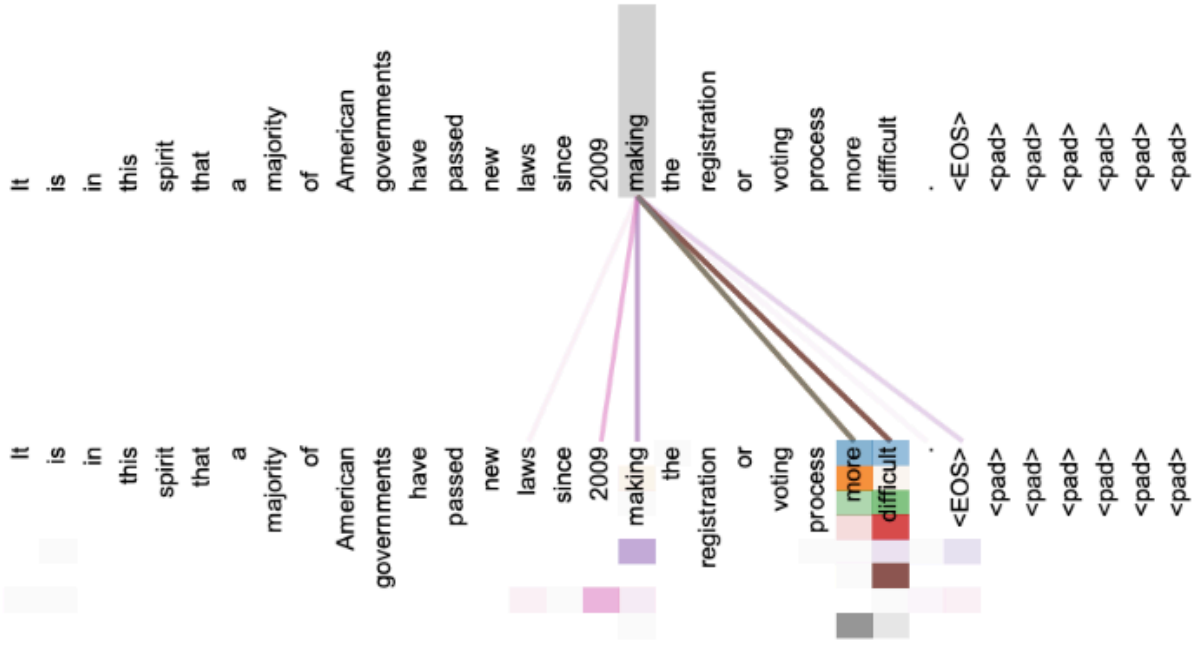
Mã nguồn cho quá trình này thường bao gồm các bước như thực hiện các phép chiếu tuyến tính cho tất cả các đầu, định hình lại để có H đầu, áp dụng attention cho tất cả các đầu, và sau đó nối lại trước khi áp dụng một lớp tuyến tính cuối cùng.

```
def forward(self, query, key, value, mask=None):
      nbatches = query.size(0)
      # 1) Do all the linear projections
      query = self.W_q(query)
      key = self.W_k(key)
      value = self.W_v(value)
      # 2) Reshape to get h heads
      query = query.view(nbatches, -1, self.heads, self.d_k).transpose(1, 2)
      key = key.view(nbatches, -1, self.heads, self.d_k).transpose(1, 2)
      value = value.view(nbatches, -1, self.heads, self.d_k).transpose(1, 2)
      # 3) Apply attention on all the projected vectors in batch.
      x, self.attn = attention(query, key, value)
      # 4) "Concat" using a view and apply a final linear.
      x=(
      x.transpose(1, 2)
      .contiguous()
      .view(nbatches, -1, self.h * self.d_k)
      )
      return self.W_o(x)
```

What Happens w/ Multi-heads?

Attention cho phép mô hình tập trung vào các phần khác nhau của câu để hiểu ngữ nghĩa một cách chính xác hơn. Ví dụ, khi tính toán self-attention cho từ "making", các giá trị attention chủ yếu tập trung vào những từ có liên quan, điều này phản ánh sự phức tạp trong ngữ nghĩa của từ này.

Example from Vaswani et al.



Từ "run" trong tiếng Anh là một động từ có nhiều nghĩa khác nhau, và cách dịch từ này sẽ phụ thuộc vào ngữ cảnh của các từ khác trong câu. Tương tự, "making" cũng có nhiều ý nghĩa khác nhau, như "making something more difficult" hay "making a cake". Để hiểu rõ cách dịch, mô hình cần thu thập thông tin từ các phần khác nhau trong câu. Một số attention heads có thể tập trung vào từ "making" chính nó, trong khi những heads khác có thể xem xét từ trước đó để thực hiện phân tích cú pháp.

Positional Encoding

Positional Encoding

Trong mô hình Transformer, một phần cốt lõi rất quan trọng là việc sử dụng mã hóa vị trí (positional encoding). Mã hóa vị trí được thêm vào cùng với embedding đầu vào. Do mô hình Transformer hoàn toàn dựa vào cơ chế attention, nếu chỉ sử dụng embedding mà không có thông tin về vị trí, sẽ không thể phân biệt được giữa các từ giống nhau. Ví dụ, nếu bạn có câu "a big dog" và "a big cat", giá trị attention cho từ "big" sẽ luôn giống nhau, vì các vector của chúng là giống hệt nhau. Điều này gây ra vấn đề, vì đôi khi thông tin cú pháp cần được lấy từ các ngữ cảnh có tính nhất quán cục bộ.

Một cách để khắc phục vấn đề này là sử dụng các mô hình RNN, vì RNN có khả năng xem xét thứ tự của các từ trước và sau. Tuy nhiên, mục tiêu của Transformer là không sử dụng RNN, vì vậy chúng ta cần một giải pháp khác. Giải pháp được đưa ra là sử dụng mã hóa vị trí. Mã hóa vị trí thêm vào một embedding dựa trên vị trí của từ trong câu. Như vậy, từ "big" xuất hiện ở vị trí thứ hai sẽ có embedding là embedding của "big" cộng với embedding của vị trí thứ hai. Ngược lại, từ "big" xuất hiện ở vị trí thứ tám sẽ có embedding là embedding của "big" cộng với

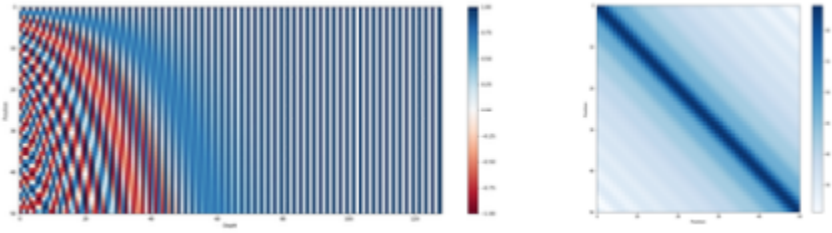
embedding của vị trí thứ tám. Cách tiếp cận này giúp giải quyết vấn đề phân biệt giữa các từ giống nhau trong ngữ cảnh khác nhau.

Sinusoidal Encoding (Vaswani+ 2017, Kazemnejad 2019)

Bài báo gốc về Transformer đã giới thiệu mã hoá vị trí thông qua việc sử dụng các mã hóa sinusoidal, nhưng vào thời điểm đó, nhiều người vẫn chưa hoàn toàn hiểu rõ lý do và cách thức hoạt động của nó.

Cách thức hoạt động của mã hóa vị trí rất đơn giản: mỗi mã hóa vị trí được tính toán dựa trên hàm sin và cos. Cụ thể, trong mỗi mã hóa, các vị trí chẵn sẽ sử dụng hàm sin, trong khi các vị trí lẻ sẽ sử dụng hàm cos. Công thức cho các mã hóa này là $w_k = 1/10000^{2k/d}$, trong đó k là chỉ số của chiều và D là kích thước của chiều.

$$p_t^{(i)} = f(t)^{(i)} := sin\left(w_k \cdot t\right) \quad \text{if } i = 2k \text{ else} := cos\left(w_k \cdot t\right) \text{ if } i = 2k + 1$$



Mục đích của việc sử dụng mã hóa vị trí là để tạo ra một sự phân biệt giữa các từ trong câu. Khi bạn nhân các mã hóa vị trí với nhau, bạn sẽ nhận được một giá trị cao hơn nếu các từ ở gần nhau trong không gian vị trí. Điều này giúp tăng cường giá trị attention cho những từ gần nhau, đặc biệt là trong các lớp đầu tiên của mô hình, nơi mà thông tin về ngữ cảnh địa phương là rất quan trọng.

Các mã hóa vị trí này được thêm vào ngay từ lớp đầu tiên và sau đó được truyền qua tất cả các lớp tiếp theo. Nhờ vào việc sử dụng mã hóa vị trí, mô hình có thể phân biệt được các trường hợp khác nhau ngay từ đầu, và sau khi thông tin đã được kết hợp qua các lớp, nó sẽ tự động xử lý thông tin về ngữ cảnh địa phương. Hơn nữa, với các kết nối phần dư, thông tin này sẽ được truyền tiếp vào các lớp sau, giúp cải thiện khả năng học của mô hình.

Learned Encoding (Shaw+ 2018)

Cách thứ hai dùng để mã hoá vị trí là học các mã hóa (learned encodings). Các mã hóa này cơ bản là tạo ra một embedding có thể học được mà bạn chỉ cần thêm vào.

Cách này đơn giản hơn vì bạn không cần phải suy nghĩ về các hàm số như sin hay cos. Nó cũng linh hoạt hơn vì mô hình có thể học bất cứ điều gì cần thiết để thực hiện tốt việc giảm thiểu tổn thất (loss). Tuy nhiên, nhược điểm lớn nhất là nó khiến cho việc suy diễn (extrapolate) ra các chuỗi dài hơn so với những gì bạn đã thấy trong quá trình huấn luyện trở nên không thể.

Vì bạn không có các embedding đã học cho các chuỗi dài hơn, nên về nguyên tắc, việc suy diễn ra các chuỗi dài hơn là không thể, trừ khi bạn sử dụng một số phương pháp heuristics. Nếu bạn sử dụng heuristics, bạn sẽ không thực sự biết điều gì sẽ xảy ra. Đó là nhược điểm của cách làm này.

Ngược lại, bạn chỉ cần làm cho K lớn hơn và tính toán một hàm xác định, và bạn có thể sử dụng lý thuyết suy diễn. Tuy nhiên, thực tế cho thấy các mô hình, ngay cả những mô hình sử dụng embedding có thể suy diễn, cũng không hoạt động tốt khi suy diễn ra các chuỗi dài hơn.

Absolute vs. Relative Encodings (Shaw+ 2018)

Khi nói về mã hóa vị trí, có một sự khác biệt giữa mã hóa vị trí tuyệt đối và mã hóa vị trí tương đối. Mã hóa vị trí tuyệt đối là như tôi đã đề cập trước đó, chúng là những mã hóa mà bạn thêm vào tại mỗi vị trí cụ thể mà không xem xét liệu một vector truy vấn có gần hay xa so với một vector khóa hay không. Nói cách khác, mã hóa vị trí tuyệt đối không trực tiếp xem xét khoảng cách giữa các vector.

Ngược lại, mã hóa vị trí tương đối lại mã hóa rõ ràng vị trí tương đối. Điều này có nghĩa là khi bạn thực hiện attention, nó sẽ xem xét liệu một embedding cụ thể không phải ở vị trí thứ tám, mà là liệu embedding khóa có cách embedding truy vấn một khoảng là -5 hay -8. Bài báo đầu tiên thực hiện điều này đã học được cách mã hóa vị trí tương đối bằng cách học một scalar, nơi mà bạn lấy trung tâm là 0 và có các giá trị âm và dương ở một khoảng cách nhất định. Họ cũng giới hạn khoảng cách này, ví dụ như từ -128 đến +128, tạo thành một vector có độ dài cố định. Bất kỳ vị trí nào xa hơn khoảng cách này sẽ nhận được cùng một embedding.

Vấn đề với phương pháp này là, thứ nhất, nó thêm vào các tham số có thể học được, và thứ hai, nó tốn kém hơn về mặt tính toán khi áp dụng lên ma trận attention mỗi lần. Vì bạn cần áp dụng điều này ở mỗi lớp, mỗi khi thực hiện attention ở mỗi lớp.

Rotary Positional Encoding (RoPE - Su+ 2021)

Một ý tưởng thông minh được gọi là mã hóa vị trí xoay vòng (rotary positional encodings) đã được phát triển. Mã hóa vị trí xoay vòng về cơ bản giống như mã hóa vị trí tuyệt đối nhưng có nhiều đặc điểm mong muốn của mã hóa vị trí tương đối. Ý tưởng cơ bản của họ là tạo ra một vector mã hóa embedding mà kết hợp cả vector thực tế và vị trí, và một vector mã hóa khác kết hợp vector tuyệt đối và vị trí. Tích của hai vector này trở thành một hàm chỉ phụ thuộc vào hai vector và vị trí tương đối, từ đó loại bỏ mọi thông tin về vị trí tuyệt đối. Điều này có nghĩa là bạn chỉ còn lại thông tin về vị trí tương đối.

$$f_q\left(x_m,\, m\right) \cdot f_k\left(x_n,\, n\right) = g\left(x_m,\, x_n,\, m - n\right)$$

Điều này phức tạp hơn so với tưởng tượng vì bạn cần đảm bảo rằng không thể khôi phục vị trí tuyệt đối, do đó chỉ dựa vào vị trí tương đối. Điều này giúp mô hình có khả năng tổng quát tốt hơn khi gặp các đầu ra mới. Để đạt được điều này, họ đã sử dụng nhiều toán học, bao gồm số phức và lượng giác. Bằng cách sử dụng các vector truy vấn và vector khóa, họ đã tạo ra một hàm có các thuộc tính mong muốn.

Cụ thể, họ thêm vào các vector một hàm cosine với tham số Theta, tương tự như tham số Omega trước đó, và m là bước thời gian, vị trí trong chuỗi. Bằng cách hoán đổi thứ tự và đảo ngược embedding, họ chứng minh được rằng hàm này có các thuộc tính mong muốn. Điều này cho phép mã hóa vị trí xoay vòng có thể mở rộng vô hạn, hoặc ít nhất là mở rộng tốt, vì nó loại bỏ mọi thông tin về vị trí tuyệt đối.

$$R^d_{\Theta,m}\mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ \vdots \\ x_{d-1} \\ x_d \end{pmatrix} \otimes \begin{pmatrix} \cos m\theta_1 \\ \cos m\theta_1 \\ \cos m\theta_2 \\ \cos m\theta_2 \\ \vdots \\ \cos m\theta_{\frac{d}{2}} \\ \cos m\theta_{\frac{d}{2}} \end{pmatrix} + \begin{pmatrix} -x_2 \\ x_1 \\ -x_4 \\ x_3 \\ \vdots \\ -x_d \\ x_{d-1} \end{pmatrix} \otimes \begin{pmatrix} \sin m\theta_1 \\ \sin m\theta_1 \\ \sin m\theta_2 \\ \sin m\theta_2 \\ \vdots \\ \sin m\theta_{\frac{d}{2}} \\ \sin m\theta_{\frac{d}{2}} \end{pmatrix}$$

Mã hóa vị trí xoay vòng được sử dụng trong mô hình LLaMA và có tác động tích cực đến việc tinh chỉnh các mô hình. Nó không có giới hạn về độ dài ngữ cảnh tối đa, điều này giúp nó mở rộng tốt hơn so với mã hóa vị trí tuyệt đối, vốn vẫn giữ thông tin về vị trí trong chuỗi và có thể dẫn đến việc mô hình bị khớp quá mức.

Cuối cùng, khi muốn áp dụng mã hóa này cho một chuỗi dài hơn, bạn không cần phải thay đổi kích thước của embedding. Bạn có thể mở rộng bằng cách tăng kích thước embedding, ngay cả khi chưa từng thấy điều gì đó về T, bạn vẫn có thể tính toán điều này về mặt lý thuyết, điều mà không thể thực hiện được với mã hóa học được.

Layer Normalization and Residual Connections

Reminder: Gradients and Training Instability

Trong phần này, chúng ta sẽ khám phá vai trò của chuẩn hóa lớp (layer normalization) và các kết nối dư (residual connections) trong việc ổn định quá trình huấn luyện của mô hình Transformer. Như đã đề cập trước đó, các mạng nơ-ron hồi tiếp RNNs thường gặp phải vấn đề về độ ổn định trong quá trình huấn luyện, đặc biệt là triệt tiêu gradient.

Cụ thể, trong RNNs, chúng ta đã thấy rằng quá trình lan truyền ngược có thể dẫn đến việc giảm độ lớn của gradient, và điều này cũng xảy ra tương tự trong các mô hình Transformer. Trên thực tế, trong phiên bản ban đầu của Transformer, đã tồn tại một vấn đề gây ra hiện tượng triệt tiêu gradient. Tuy nhiên, vấn đề này đã được khắc phục trong các phiên bản mới hơn của Transformer.

Layer Normalization (Ba et al. 2016)

Khi chúng ta chạy các mô hình Transformer với số lượng lớp lớn, chẳng hạn như 8, 12 hoặc 16 lớp, một vấn đề thường gặp là hiện tượng triệt tiêu gradient. Để khắc phục điều này, Layer Normalization được sử dụng. Mặc dù Layer Normalization không hoàn toàn giải quyết vấn đề

triệt tiêu gradient, nhưng nó giúp ngăn chặn gradient bùng nổ hoặc trở nên không ổn định. Cách hoạt động của Layer Normalization là chuẩn hóa các đầu ra để chúng nằm trong một khoảng giá trị nhất định, từ đó giảm thiểu sự biến thiên trong quy mô.

Cụ thể, Layer Normalization thực hiện các bước sau: đầu tiên, nó tính toán trung bình của các vector bằng cách cộng tất cả các giá trị lại và chia cho số lượng phần tử trong vector. Tiếp theo, nó tính độ lệch chuẩn của vector bằng cách lấy tổng bình phương của các giá trị trừ đi trung bình, sau đó lấy căn bậc hai của tổng đó. Kết quả là các giá trị trong vector sẽ được chuẩn hóa để có trung bình bằng 0 và độ lệch chuẩn bằng 1.

$$\mu(x) = \frac{1}{n}\sum_{i=1}^{n} x_i \qquad \sigma(x) = \sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(x_i - \mu\right)^2}$$

$$LayerNorm(x;\ g,\ b) = \frac{g}{\sigma(x)} \odot (x - \mu(x)) + b \qquad \text{with } g\text{: } gain,\ b\text{: } bias$$

Tuy nhiên, Layer Normalization còn thực hiện thêm hai bước quan trọng khác: thêm một độ lệch (bias) và nhân với một hệ số khuếch đại (gain). Điều này có nghĩa là sau khi chuẩn hóa, các giá trị sẽ được dịch chuyển ra khỏi khoảng giá trị chuẩn. Ví dụ, nếu chúng ta có một vector mới X1 và một vector khác X2, Layer Normalization sẽ chuẩn hóa X2 và sau đó dịch chuyển nó lên một vị trí mới trong không gian. Nhờ vậy, tất cả các vector sẽ nằm trong một phần không gian nhất quán, với vị trí và độ phân tán được xác định bởi bias và gain.

Điều này mang lại lợi ích cho sự ổn định trong quá trình huấn luyện, vì mỗi lần chúng ta tiêu thụ đầu ra từ một lớp đã được chuẩn hóa, chúng ta sẽ nhận được một kết quả có thể dự đoán được.

Một câu hỏi thường gặp là sự khác biệt giữa Layer Normalization và Batch Normalization. Batch Normalization chuẩn hóa không phải trên toàn bộ lớp mà là trên toàn bộ batch, tức là tất cả các phần tử trong batch. Điều này có thể gây ra vấn đề vì Batch Normalization thay đổi các thống kê dựa trên các phần tử khác trong batch. Trong khi đó, Layer Normalization chỉ phụ thuộc vào từng trường hợp cụ thể, do đó không cần lo lắng về các batch khác nhau. Mỗi đầu vào và đầu ra đều nhất quán, bất kể các phần tử khác trong batch.

RMSNorm (Zhang and Sennrich 2019)

Một cải tiến đáng chú ý của Layer Norm là RMS Norm, hay còn gọi là Square Normalization. RMS Norm thực chất là một phiên bản đơn giản hóa của Layer Norm, trong đó bước chuẩn hóa trung bình đã được loại bỏ. Thay vì đưa mọi thứ về trung tâm của không gian, RMS Norm giữ nguyên vị trí của các giá trị trong không gian nhưng thực hiện việc chuẩn hóa lại độ khuếch tán giữa chúng.

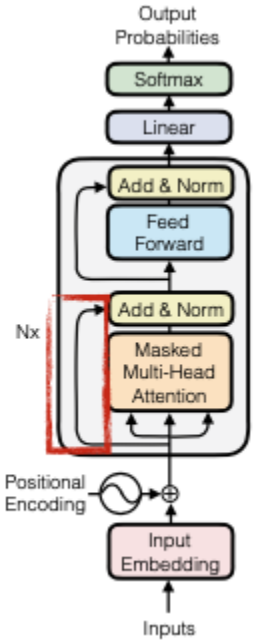
Khi xem xét Layer Normalization, chúng ta thấy rằng nó tính toán giá trị trung bình, trong khi RMS Norm không thực hiện bước này. Cụ thể, trong Layer Norm, chúng ta có bước trừ đi giá trị trung bình, nhưng trong RMS Norm, không có bước trừ nào cả. Đây chính là điểm khác biệt cơ bản giữa hai phương pháp này.

$$RMS(x) = \sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2} \qquad RMSNorm(x) = \frac{x}{RMS(x)} \cdot g$$

Mặc dù RMS Norm không thực sự tốt hơn Layer Norm, nhưng nó mang lại kết quả tương tự và nhanh hơn, đồng thời không kém hơn nhiều. Chính vì lý do này, RMS Norm được sử dụng để cải thiện hiệu suất trong các mô hình học sâu, bao gồm cả trong mô hình Lama. Thêm vào đó, RMS Norm cũng loại bỏ tham số bias và chỉ giữ lại tham số gain, giúp giảm số lượng tham số cần thiết.

Residual Connections

Trong bài viết gần đây, chúng ta đã thảo luận về các kết nối phần dư (residual connections), một khái niệm quan trọng trong mạng nơ-ron. Kết nối phần dư là một kết nối cộng giữa đầu vào và đầu ra, cho phép chúng ta thêm đầu vào vào đầu ra của một hàm. Điều này rất đơn giản nhưng lại có tác động lớn đến quá trình học của mô hình. Thay vì học cách ánh xạ đầu vào thành đầu ra, mô hình sẽ học cách điều chỉnh đầu ra bằng cách áp dụng sự khác biệt từ đầu vào.



$$Residual(x,\ f) = f(x) + x$$

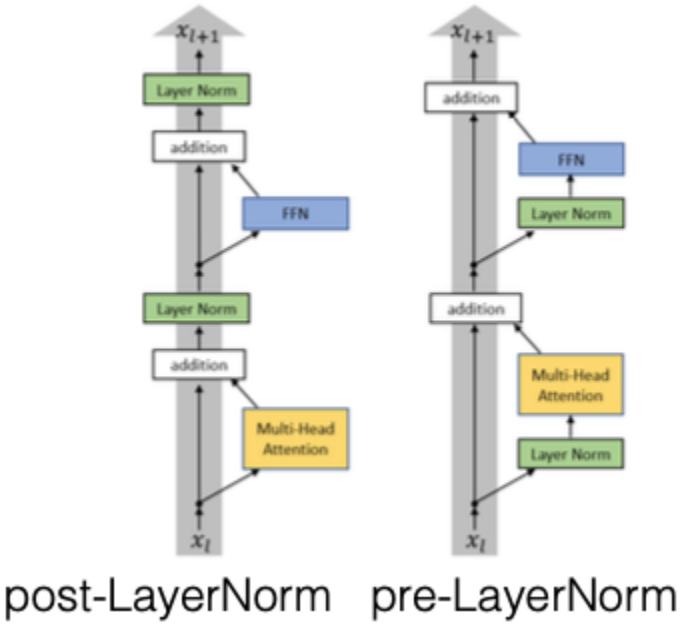
Một điểm thú vị là kết nối phần dư có ảnh hưởng lớn đến cơ chế attention, đặc biệt là trong multi-head attention. Khi có kết nối phần dư, mô hình sẽ giảm thiểu việc chú ý đến chính nó, mà thay vào đó sẽ tập trung vào các thông tin xung quanh để hiểu rõ hơn về những gì cần thêm vào để tạo ngữ cảnh. Điều này có nghĩa là thông tin từ chính nó đã được cung cấp miễn phí thông qua kết nối phần dư, do đó nó chỉ cần thu thập thông tin bổ sung từ các nguồn khác.

Khi xem xét các đầu vào trong visualization, chúng ta có thể thấy rằng chỉ có một đầu attention đang chú ý đến chính nó, trong khi tất cả các đầu attention khác không làm như vậy. Điều này

xảy ra chính vì sự hiện diện của các kết nối residual. Nếu không có chúng, mô hình sẽ phải chú ý nhiều hơn đến chính nó và sau đó mới thu thập thông tin từ các nguồn khác.

Post- vs. Pre-Layer Norm (e.g. Xiong et al. 2020)

Một cải tiến quan trọng khác của Transformer là việc áp dụng Layer Norm trước và sau các lớp. Trong thiết kế ban đầu của Transformer, Layer Norm được áp dụng sau các bước như multi-head attention và feed-forward network. Tuy nhiên, việc này đã làm gián đoạn kết nối phần dư, điều này gây khó khăn cho việc truyền gradient. Khi có một hàm khác ngoài hàm đồng nhất (identity) ở giữa các lớp, việc truyền gradient qua nhiều lớp sẽ trở nên khó khăn hơn.



Giải pháp cho vấn đề này là sử dụng pre-layer Norm, trong đó Layer Norm được áp dụng trước các lớp như multi-head attention và feed-forward. Điều này tạo ra một kết nối phần dư trực tiếp từ đầu đến cuối, giúp cải thiện việc truyền gradient và làm cho quá trình huấn luyện của Transformer trở nên ổn định hơn.

Một câu hỏi thường gặp là tại sao việc áp dụng Layer Norm giữa các lớp lại gây hại cho việc truyền gradient. Câu trả lời là bất kỳ hàm nào khác ngoài hàm đồng nhất đều có thể làm thay đổi gradient. Cụ thể, Layer Norm có thể làm giảm gradient nếu độ lệch chuẩn (standard deviation) trong lớp đó lớn, đặc biệt khi gradient ban đầu nhỏ. Khi đó, việc chia cho độ lệch chuẩn sẽ làm cho gradient trở nên nhỏ hơn.

Feed Forward Layers

Feed Forward Layers

Các lớp Feed Forward có nhiệm vụ trích xuất các đặc trưng kết hợp từ đầu ra đã được chú ý. Cụ thể, mạng feed forward được áp dụng độc lập cho từng vector trong chuỗi. Ví dụ, nếu chúng ta có một vector, quá trình áp dụng sẽ diễn ra như sau:

$$FFN\left(x;\ W_1,\ b_1,\ W_2,\ b_2\right) = f\left(xW_1 + b_1\right)W_2 + b_2$$

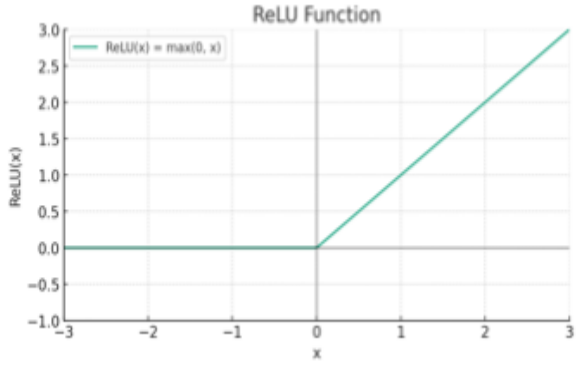
Hiện nay, việc loại bỏ bias trong các lớp này khá phổ biến, chủ yếu vì bias là các tham số thừa không thực sự cần thiết và có thể gây ra một số bất ổn trong quá trình huấn luyện. Do đó, bạn thường thấy các lớp tuyến tính không có bias, vì điều này không ảnh hưởng đến khả năng học của mạng.

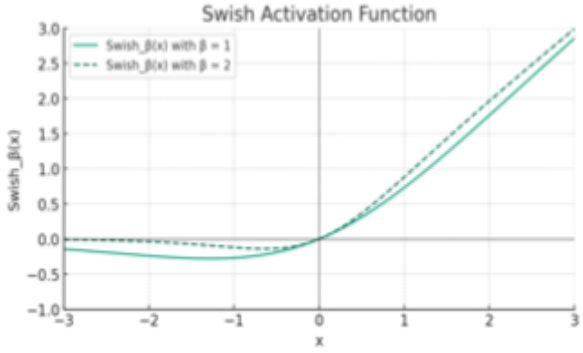
Hàm f ở đây đại diện cho một loại phi tuyến nào đó. Thông thường, mạng feed forward trong các mô hình Transformer sẽ mở rộng đến một vector lớn hơn để trích xuất nhiều đặc trưng hơn. Mỗi phần tử trong vector này có thể được coi là một đặc trưng.

Nhiều nhà nghiên cứu khi giải thích các mô hình Transformer thường xem xét những đặc trưng này, vì chúng thường tương ứng trực tiếp với thông tin mà chúng ta mong đợi. Ví dụ, khi người ta muốn ghi nhớ các sự kiện cụ thể trong Transformer, như "who is the president of the United States", họ thường xem xét các vector trong lớp feed forward này.

Some Activation Functions in Transformers

Hàm kích hoạt phổ biến nhất được sử dụng trong Transformer gốc là ReLU (Rectified Linear Unit). Hàm ReLU được định nghĩa là Max(0, x), có nghĩa là nó trả về giá trị 0 cho bất kỳ đầu vào nào nhỏ hơn 0, và giữ nguyên giá trị đầu vào nếu nó lớn hơn 0. Tuy nhiên, một vấn đề lớn với ReLU là khi đầu vào nhỏ hơn 0, gradient sẽ bằng 0, điều này có thể gây ra những khó khăn trong quá trình tối ưu hóa.





Để khắc phục vấn đề này, một hàm kích hoạt thay thế gần đây được sử dụng là SiLU (Sigmoid Linear Unit), hay còn gọi là hàm sigmoid tuyến tính. Hàm này được định nghĩa là $x \odot Sigmoid(\beta x)$, trong đó β thường được đặt bằng 1. SiLU có hình dạng tương tự như ReLU nhưng không có gradient bằng 0 ở bất kỳ điểm nào, cho phép có một đòn bẩy nhẹ khi đầu vào trở nên âm, giúp các giá trị có cơ hội phục hồi và tiến gần hơn đến giá trị trung tâm. Theo kinh nghiệm, SiLU cho thấy hiệu quả tốt trong thực tế và cũng được áp dụng trong một số mô hình hiện đại.

Optimization Tricks for Transformers

Transformers are Powerful but Fickle

Mặc dù Transformers rất mạnh mẽ, nhưng chúng cũng có thể rất khó khăn trong việc tối ưu hóa. Khi mới ra mắt, chúng ta chưa có những công thức huấn luyện ổn định cho các mô hình này, dẫn đến việc tối ưu hóa trở nên phức tạp hơn.

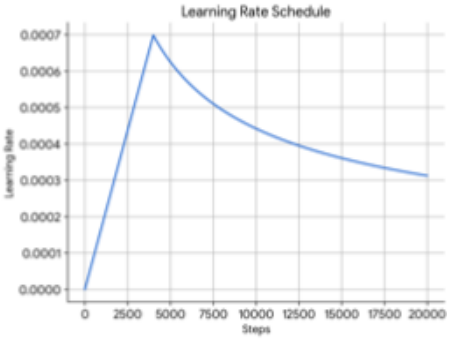
Một ví dụ điển hình cho điều này là cuốn nhật ký huấn luyện của Meta về cách họ đã huấn luyện mô hình 175 tỷ tham số. Trong đó, bạn sẽ thấy tất cả những vấn đề mà họ gặp phải trong quá trình huấn luyện, mặc dù họ là những chuyên gia trong lĩnh vực này. Những vấn đề này bao gồm việc máy móc gặp sự cố, các kỹ sư phần cứng phải khôi phục lại máy móc, hay việc mất mát (loss) không ổn định và phải quay lại các bước trước đó một cách thủ công.
https://github.com/facebookresearch/metaseq/blob/main/projects/OPT/chronicles/OPT175B_Logbook.pdf

Optimizers for Transformers

Trong quá trình ổn định việc huấn luyện các mô hình Transformer, có một số phương pháp mà các nhà nghiên cứu thường áp dụng. Đầu tiên, về các bộ tối ưu (optimizers), chúng ta đã thảo luận về SGD (Stochastic Gradient Descent), một phương pháp cập nhật theo hướng giảm thiểu hàm mất mát. Một bộ tối ưu phổ biến khác là Adam, với đặc điểm là thêm một thành phần động lượng và chuẩn hóa theo độ lệch chuẩn của các đầu ra, giúp cập nhật các tham số ít được cập nhật thường xuyên hơn.

Một cải tiến mới được giới thiệu bởi Vaswani và các cộng sự khi họ phát triển Transformers là việc điều chỉnh tốc độ học thông qua việc tăng và giảm dần. Cụ thể, họ đã thực hiện 4,000 bước khởi động (warm-up steps) để tăng tốc độ học trước khi giảm dần. Tuy nhiên, gần đây có ý kiến cho rằng có thể đạt được kết quả tốt hơn mà không cần khởi động, miễn là sử dụng pre-layer Norm. Mặc dù khởi động vẫn được sử dụng rộng rãi, nhưng không còn là điều bắt buộc với các công thức huấn luyện mới.

$$lrate = d_{\text{model}}^{-0.5} \cdot \min(step^{-0.5}, step * warmup\_steps^{-1.5})$$



Ngoài ra, một số người cũng áp dụng phương pháp giảm tốc độ học tuyến tính thay vì giảm dần theo dạng đường dốc. Một bộ tối ưu khác đang được ưa chuộng là AdamW. AdamW thực hiện việc giảm trọng số, tức là giảm dần các trọng số về phía không. Mục đích của việc này là để ngăn ngừa hiện tượng quá khớp (overfitting) của mô hình, tương tự như một phương pháp

chuẩn hóa (regularization). AdamW là một sự điều chỉnh của weight decay, đặc biệt xem xét đến việc Adam sử dụng động lượng trong quá trình chuẩn hóa, để đảm bảo rằng nó thực sự tương ứng với các điều kiện chuẩn hóa đúng cách.

Low-Precision Training

Trong quá trình huấn luyện các mô hình lớn, việc sử dụng độ chính xác thấp (low Precision) là rất cần thiết, đặc biệt khi bạn chỉ có ít GPU. Huấn luyện với độ chính xác 32 bit có thể tốn kém, vì vậy việc sử dụng 16 bit là phổ biến. Có hai lựa chọn chính cho độ chính xác 16 bit: fp16 và bfloat 16.

fp16 là dạng số thực 16 bit tiêu chuẩn, được sử dụng rộng rãi trên các máy tính và CPU. Nó phân bổ 1 bit cho dấu, 5 bit cho số mũ và 10 bit cho phần thập phân, cho phép biểu diễn các phân số và số mũ với độ chính xác tương đối nhưng có giới hạn về phạm vi giá trị.

Ngược lại, bfloat16, được phát triển bởi Google Brain, tăng số bit cho số mũ lên 8 và giảm số bit cho phần thập phân xuống 7. Điều này cho phép biểu diễn một phạm vi giá trị rộng hơn với độ chính xác thấp hơn, giúp xử lý tốt hơn các số rất nhỏ và rất lớn. Mặc dù mất một chút độ chính xác ở phần thập phân, bfloat16 lại ổn định hơn trong quá trình huấn luyện.

Hiện nay, phần cứng hỗ trợ cho bfloat16 khá tốt, với các GPU của Nvidia và nhiều thiết bị khác đều hỗ trợ công nghệ này.

Checkpointing/ Restarts

Khi huấn luyện các mô hình lớn, một yếu tố quan trọng cần lưu ý là việc kiểm tra và khôi phục (checkpointing and resets). Dù đã cố gắng hết sức, quá trình huấn luyện vẫn có thể gặp vấn đề. Một trong những cách để theo dõi các vấn đề tiềm ẩn là giám sát độ lớn của gradient (gradient norm). Ví dụ, trong quá trình huấn luyện của Meta, khi theo dõi độ lớn của gradient, có thể thấy rằng nó đột ngột tăng cao, điều này thường chỉ ra một vấn đề. Sau khi tăng đột biến, perplexity của mô hình có thể giảm xuống một thời gian, nhưng sau đó lại bắt đầu tăng lên, cho thấy mô hình đang ở trong một trạng thái không tốt trong không gian tham số.

Nếu quá trình huấn luyện gặp sự cố, một giải pháp phổ biến là quay lại một checkpoint trước đó. Thay vì quay lại ngay trước khi gradient tăng đột biến, bạn có thể quay lại khoảng 100 bước trước đó, sau đó xáo trộn tập dữ liệu huấn luyện hoặc chuyển sang một phần khác của tập dữ liệu để tiếp tục. Việc này giúp tạo ra sự ngẫu nhiên trong quá trình huấn luyện, có thể giúp ổn định quá trình.

Ngoài ra, việc kiểm tra mã nguồn cũng rất quan trọng. Nếu mã của bạn không có các hàm nguy hiểm và tốc độ học được thiết lập hợp lý, khả năng gặp sự cố sẽ giảm đi đáng kể. Nếu bạn thấy các tăng đột biến liên tục, đó có thể là dấu hiệu cho thấy có vấn đề trong mã. Ví dụ, nếu bạn đang tính log của một giá trị mà bạn nghĩ là dương, nhưng giá trị đó lại gần bằng 0, bạn sẽ gặp phải gradient rất lớn, vì log của giá trị gần 0 có gradient vô hạn.

Cuối cùng, việc chẩn đoán và làm cho quá trình huấn luyện ổn định hơn là một kỹ năng cần có, và thường cần nhiều kinh nghiệm.

Comparing Transformer Architectures

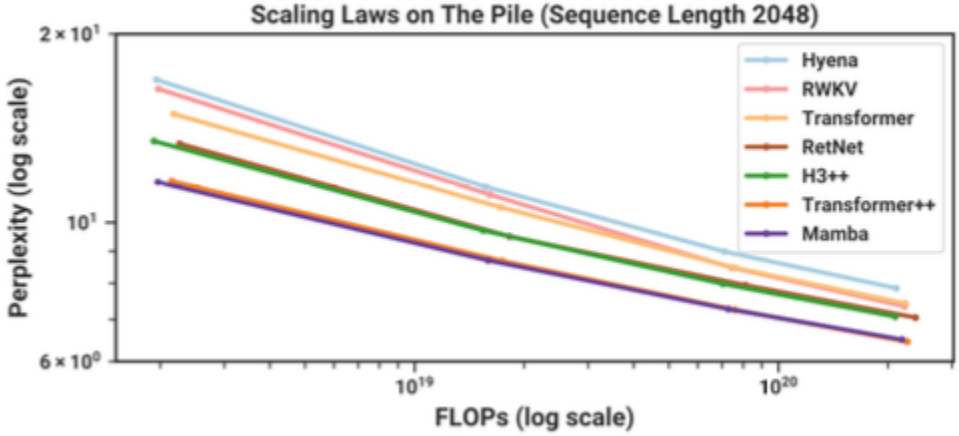
Original Transformers vs. LlaMa

Phần cuối của bài viết, chúng ta sẽ so sánh các kiến trúc Transformer, đặc biệt là hai mô hình nổi bật là mô hình Transformer gốc của Vaswani et al. và LLaMA. Cả hai mô hình đều sử dụng Transformer và có nhiều điểm tương đồng, nhưng cũng có những khác biệt đáng chú ý.

Đầu tiên, về vị trí chuẩn hóa, Vaswani et al. sử dụng post-norm trong khi LLaMA áp dụng pre-norm. Tiếp theo, loại chuẩn hóa cũng khác nhau: Vaswani et al. sử dụng Layer Norm, trong khi LLaMA chọn RMS Norm. Về hàm phi tuyến, Vaswani et al. sử dụng ReLU, còn LLaMA sử dụng SiLU. Cuối cùng, về mã hóa vị trí, Vaswani et al. áp dụng sinusoidal, trong khi LLaMA sử dụng RoPE.

How Important is It?

Một nghiên cứu thú vị được thực hiện bởi Albert Goo, đã đề xuất một kiến trúc mới cho mô hình Transformer. Một điểm đáng chú ý trong bài báo này là sự so sánh giữa kiến trúc Transformer gốc và kiến trúc Transformer Plus+ (LLaMa). Nghiên cứu đã so sánh perplexity (độ khó) của hai kiến trúc này, cụ thể là perplexity trên thang log.



Kết quả cho thấy, để đạt được kết quả tương tự, kiến trúc cũ cần đến gấp 10 lần số lượng phép toán (flops) so với kiến trúc mới. Điều này cho thấy tầm quan trọng của việc tối ưu hóa kiến trúc trong quá trình huấn luyện mô hình. Mặc dù nhiều người cho rằng "scale is all you need" (quy mô là tất cả những gì bạn cần) và không coi trọng việc tối ưu hóa kiến trúc, nhưng thực tế cho thấy rằng việc cải tiến kiến trúc là rất quan trọng. Những tiến bộ trong lĩnh vực này trong 5 đến 7 năm qua đã tạo ra sự khác biệt lớn trong hiệu suất của các mô hình ngôn ngữ lớn.

Resources

https://phontron.com/class/anlp2024/lectures/#transformers-jan-30