A Arquitetura Transformer (Rascunho)

Caleb Martim

24 de abril de 2025

Resumo

Este documento tem o objetivo de explicar todos os conceitos e matemática da arquitetura Transformer, introduzida pela primeira vez em 2017 pelo Google no artigo, *Attention Is All You Need*. Aqui, iremos discutir os princípios de machine learning, deep learning e como isso se evoluiu para a arquitetura Transformer original. Explicaremos também a Álgebra Linear e o Cálculo usado que dão fundamento teórico para esses conceitos.

1 Definições

- **Inteligência Artificial:** Se refere a capacidade de sistemas computacionais de realizar tarefas que tipicamente requerem inteligência humana. Essas tarefas incluem raciocínio, aprendizagem, resolução de problemas, percepção, entendimento de linguagem e realização de decisões.
- **Machine Learning:** É um subconjunto da área de inteligência artificial que inclui os modelos que usam dados para melhorar a avaliação de um procedimento computacional para realizar uma tarefa específica. Isto é, o computador aprende melhor a realizar uma tarefa com dados, e "observação de padrões".
- **Deep Learning:** Um modelo de machine learning que usa *redes neurais artificiais* para seus fins. '*Deep*' vem da estrutura de redes neurais, que, tipicamente, são constituídas por múltiplas camadas.
- **Arquitetura:** Padrões de design e estruturas que irão construir os sistemas de inteligência artificial. Como "*Transformers*", "*Redes Neurais Recorrentes*" e "*Redes Neurais Convolucionais*".
- **Large Language Model:** Um sistema de inteligência artificial utilizado para entender, gerar e manipular linguagem humana. Estes são sistemas específicos, já treinados, como "GPT-3", "Claude", "LLaMA", etc.

2 Álgebra Linear

2.1 Produto escalar

Sejam v e w vetores de dimensão n, seu produto escalar, ou produto interno, ou *dot product*, é denotado e definido por:

$$v \cdot w = \sum_{i=1}^{n} v_i \cdot w_i$$

Seu significado pode ser interpretado geometricamente como a projeção do vetor w sobre o vetor v multiplicado pelo tamanho de v, com a consideração de que se w está apontando para um lado oposto ao de v, $v \cdot w$ é negativo. Matematicamente, seu significado equivalente pode ser visto como v sendo, na verdade, uma matriz que representa uma transformação linear da dimensão n para a dimensão 1, onde v é onde param os vetores bases da dimensão n e $v \cdot w$ é a aplicação dessa transformação linear em w.

3 O Transformer

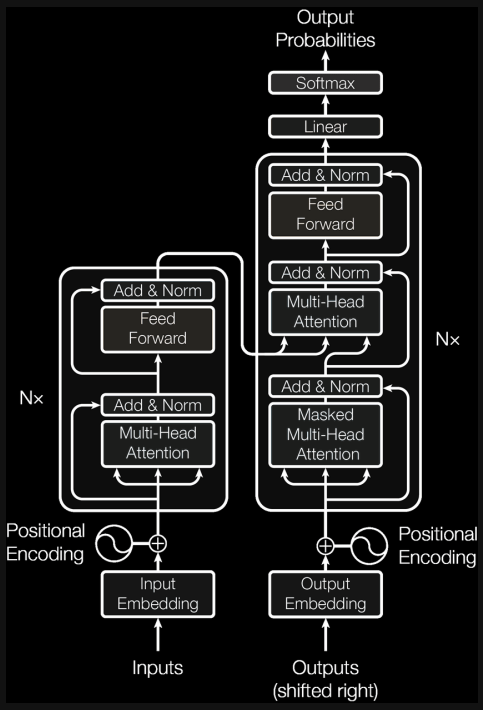

Figura 1: A arquitetura modelo do Transformer

Dado uma sequência de texto como entrada (*input*), a sequência é dividida em tokens e cada token é transformado em um vetor de números reais, isto é feito com o objetivo de codificar o significado da entrada. Esta sequência de vetores é passada por um mecanismo de atenção (*attention*). Neste processo, tokens atualizam os vetores de outros tokens, abstrativamente, esclarificando o significado da entrada. Após isso, os vetores passam por outra estrutura, às vezes referida como um *Multilayer Perceptron* e outras como um *Feed-Forward Network*, cada vetor é atualizado novamente, mas dessa vez individualmente, em paralelo. Após todas essas operações o último vetor terá valores que podem dar sentido a um novo possível token final, que estende a entrada, uma última operação é realizada nesse vetor e uma tabela de distribuição de probabilidades é dada para indicar o conjunto de tokens que estão previstos para possivelmente continuar a sequência. Quando o token for escolhido, ele pode ser anexado ao fim da entrada e a entrada pode ser realimentada no transformer para gerar o próximo token, continuamente, enquanto for desejado.

Inicialmente, no modelo, temos o que chamamos de uma matriz de vetorização, ou matriz de *embedding*, que define um vetor de valores reais para cada possível token que possa ser reconhecido pelo modelo. Essa matriz é o que define os vetores iniciais da entrada quando ela é recebida. Para o modelo GPT-3, sua matriz de embedding, denotada por W_E (ou apenas W), tem $50,257$ linhas e $12,288$ colunas, por causa disso, dizemos que W tem forma $(50,257, 12,288)$. Além disso, um transformer só pode operar sobre um número máximo de tokens em cada uma de suas camadas, chamaremos esse número máximo de nosso *context window* e iremos denotá-lo por C, para o GPT-3, $C = 2,048$. Vamos denotar o número de dimensões para os vetores de embedding por d_{model}, por exemplo, no GPT-3, $d_{model} = 12,228$. Vamos denotar o número total de tokens reconhecíveis no modelo por V (tamanho do vocabulário), por exemplo, no GPT-3, $V = 50,257$.

Suponha que tenhamos uma sequência de input I, ao dividir I em tokens, tenhamos uma sequência de números que representa os códigos dos tokens da sequência I. Algo como, por exemplo:

$$Encoding(I) = [9, 15, 20]$$

Por simplicidade, suponha que o número de tokens obtidos seja exatamente o tamanho da janela de contexto para o nosso modelo, neste caso, $C = 3$. Agora, vamos usar a matriz W como uma espécie

de *lookup table* para construir o vetor de embedding de I:

$$Embedding(I) = \begin{bmatrix} W_9 \\ W_{15} \\ W_{20} \end{bmatrix}$$

Como cada W_x é um vetor de tamanho d_{model}, $Embedding(I)$ é na verdade uma matriz de forma (C, d_{model}).

Agora, quando quisermos gerar um próximo token para nossa sequência, vamos fazer uma amostragem de todos os tokens possíveis, a partir da tabela de probabilidades gerada após a sequência I for processada no bloco de atenção. O bloco de atenção irá gerar uma matriz de *hidden states* h com forma (C, d_{model}) a partir da nossa entrada I; quando obtermos h, vamos fazer a projeção:

$$L = hW^T$$

Onde W^T denota a transposição da matriz de embedding, note que esta matriz tem forma (d_{model}, V), o que permite que a projeção seja realizada. W^T pode ser chamada de a matriz de *unembedding* e é importante observar que em muitos modelos modernos, esta matriz é, na verdade, gerada por aprendizagem do modelo e pode ser denotada por W_U. Chamaremos os elementos da matriz L resultante da projeção de *logits*. Note que L tem forma (C, V). Isto representa que para cada token que temos na nossa sequência de input, estamos atribuindo valores para cada token que existe no nosso vocabulário. O próximo token da nossa sequência, o token gerado, vai depender apenas dos logits do último token da nossa sequência, isto é, vai depender apenas de L_C. Daí, vamos usar uma função para normalizar esses valores para uma distribuição de probabilidades, chamada $softmax$, que recebe um vetor de tamanho n e retorna outro vetor de tamanho n, onde todo valor está no intervalo $[0, 1]$ e a soma de todos os seus valores é igual a 1, intuitivamente, os maiores valores do vetor original se transformam nos valores mais pertos de 1, e os mais baixos, mais pertos de 0.

3.1 Softmax

Na função $softmax$, nós primeiramente transformamos todo valor v_i do nosso vetor e o transformamos em e^{v_i}, isto garante que cada elemento se torna positivo; depois, pegamos a soma de todos os elementos e vamos calcular a contribuição de cada elemento para esta soma, esta será a probabilidade que atribuímos a ele no nosso novo vetor. Em outras palavras, seja $v = (v_1, v_2, ..., v_n)$ o vetor de entrada, e seja $S = \sum_{i=1}^{n} e^{v_i}$, os elementos do vetor de saída $w = (w_1, w_2, ..., w_n)$ são definidos por

$$w_i = \frac{e^{v_i}}{S}$$

É simples verificar que, aqui, todo elemento está entre 0 e 1 e a soma de todos os elementos é 1. Além disso, podemos definir uma *temperatura* T, e redefinir $S := \sum_{i=1}^{n} e^{\frac{v_i}{T}}$. Assim como,

$$w_i := \frac{e^{\frac{v_i}{T}}}{S}$$

É possível notar que quando $T \to 0$, valores maiores crescem mais rápido, ou seja, a probabilidade associada a eles é maior. Quando $T \to \infty$, valores maiores crescem mais devagar, ou seja, a distribuição de probabilidade é mais igualitária e outros possíveis elementos têm mais chances de serem escolhidos como próximo token. Como última consideração, na terminologia, chamamos os elementos do vetor resultante de 'probabilidades', e chamamos os elementos do vetor obtido pela última multiplicação matriz-vetor de '*logits*'.

4 Attention

O mecanismo de attention funciona da seguinte forma: primeiro, rodamos alguns processos em paralelo que vão criar transformações lineares dos vetores de embedding. A posição de cada token também faz

parte do vetor de embedding, também dando possível significado semântico a ele. Um processo que faz essa operação é chamado de *attention head*.

Para cada token será gerado um vetor de 128 dimensões chamado *Query*, podemos chamá-lo de vetor de consulta. Para ele ser computado, iremos multiplicar uma matriz, que, no GPT-3, é referida como W_Q e a multiplicamos pelo nosso vetor, de embedding, chamaremos-o de E daqui em diante. Denotemos também nosso vetor de consulta por Q. Este vetor representa perguntas que queremos fazer sobre E, "Ele é um substantivo?", "Ele tem adjetivos que o qualificam?", etc. Os vetores de consulta são gerados por cada attention head em paralelo (dúvida: sabendo que attention heads criam padrões de atenção diferentes, quando eu escrevi isso, eu quis dizer, os vetores de consulta para cada attention head ou ambos isso e cada vetor de cada embedding isolado?). Um processo similar é feito para um vetor *Keys*, podemos chamá-lo de vetor chave, uma matriz W_K é multplicada para cada E_i da sua entrada gerando um vetor K_i. Conceitualmente, queremos pensar no vetor K como respondendo às perguntas do vetor Q, portanto, ele também é composto por 128 dimensões. Note que o vetor K_i não é o vetor que está respondendo às perguntas do vetor Q_i, serão todos os outros vetores ao mesmo tempo que estarão fazendo esse trabalho. Dizemos que o vetor K_j está respondendo o vetor Q_i se eles se encontram mais ou menos na mesma região do espaço de dimensão 128, logo, para determinar isso, iremos calcular o produto escalar de cada par (Q_i, K_j). Pela intuição de produto escalar, é fácil ver que, quanto maior for esse resultado, mais próximos os vetores estão em seu espaço de 128 dimensões, e, quanto menor, mais distantes eles se encontram. Quando o produto escalar entre Q_i e K_j é relativamente alto, dizemos que o embedding de E_j atende ao embedding de E_i, também dizemos que esta é uma forte relação de atenção entre os embeddings.

Vamos construir uma matriz 2D com os valores desses produtos escalares, chamaremos ela de *attention pattern*, ou, padrão de atenção, e definimos que a célula na posição (i, j) desta matriz é o produto escalar $Q_i \cdot K_j$, é importante notar que o número de elementos nessa matriz é o quadrado da janela de contexto, por isso, ela precisa ter um número pequeno, como $2,048$, este número, por exemplo, gera uma matriz com $4,194,304$ elementos. Note que os elementos da linha i estão justamente dando as respostas para a consulta i usando todos os vetores de embedding, podemos quantificar então o quanto cada K_j responde à Q_i aplicando a função $softmax$ na linha i do padrão de atenção. É este processo todo que motiva a fórmula:

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

Q, aqui, é a sequência de todos os vetores consulta da entrada, K é a sequência de todos os vetores chave, QK^T é a matriz 2D padrão de atenção (que computa os produtos escalares), a divisão por $\sqrt{d_k}$ representa a divisão de todo valor da matriz pela raiz quadrada da dimensão do modelo (128), o softmax é a aplicação dessa função em toda linha. A sequência V será explicada mais tarde e o que o produto com ela faz.

Uma importante observação: Para o processo de treinamento, é eficiente fazer com que o modelo tente prever não só o próximo token da entrada, mas fazer também com que, na própria entrada, para cada token, o modelo tente prever qual é o próximo token, não só isso, é também útil para o modelo aprender a prever o próximo token não só em contextos de tamanho $2,048$, mas, também, contextos de tamanho no intervalo $[1, 2,048]$. Assim, ele aprende um pouco com cada token que é dado na entrada. Para fazer isso, não podemos deixar com que vetores chave K_j respondam consultas Q_i com $j > i$, para isso, vamos artificialmente trocar toda célula no padrão de atenção com coordenadas (i, j) que atendem $j > i$ para $-\infty$, isso vai fazer com que, na função softmax, esses valores serão todos transformados em 0 e os outros elementos na linha de interesse estejam no intervalo $[0, 1]$ e somem para 1.

Agora, vamos introduzir mais um vetor gerado por cada embedding, dessa vez um vetor de valores, *(values)* criado a partir de uma multiplicação de E por uma matriz W_V criando um vetor V, este, de $12,288$ dimensões. O que vamos fazer, para atender a última parte da nossa fórmula é fazer uma multiplicação por escalar de cada valor no nosso padrão de atenção atual que está na coluna j com o vetor V_j. Isto finaliza a fórmula de atenção. Por fim, vamos somar cada um desses vetores numa linha i, seja esta soma ΔE_i e no final somar esse vetor com o embedding E_i, este será o novo E_i

$$E_i := E_i + \Delta E_i$$

.

Novamente, todas essas operações são realizadas em um só head de attention, mas um bloco inteiro de attention consiste de *Multi-headed attention.*. Cada um desses têm, na verdade, diferentes mapeamentos W_Q, W_K, W_V. Por exemplo, GPT-3 usa 96 attention heads em cada bloco de attention. Em cada processo, cada head produz um ΔE_i para cada E_i da janela de contexto. Na verdade, no final

$$E_i := E_i + \sum_{j=1}^{96} \Delta E_i^{(j)}$$

Onde $\Delta E_i^{(j)}$ denota a mudança do embedding E_i no head attention j.

4.1 Output Matrix

Agora, uma pequena correção ao texto acima é que a matriz W_V não existe de verdade do jeito que eu descrevi. Ele é na verdade, como W_Q e W_K, uma matriz de 128 linhas e $12,228$ colunas. Porém, claro, não podemos somar um embedding E_i com um vetor V_i que possui apenas 128 dimensões. O que é feito é o seguinte, vamos denotar, aqui, por $\Delta V_i^{(k)}$ a soma dos vetores valor, já multiplicados pelos seus pesos no padrão de atenção no attention head k:

$$\Delta V_i^{(k)} = \sum_{j=1}^{12,288} w_{i,j}^{(k)} V_j^{(k)}$$

E vamos denotar apenas por ΔV_i a concatenação desse vetor entre todos os attention heads

$$\Delta V_i = \bigoplus_{k=1}^{96} \Delta V_i^{(k)}$$

Onde $\oplus$ denota concatenação de vetores, por falta de um símbolo melhor. Teremos, independente dos attention heads uma matriz de output, W_o, de tamanho $12,288 \times 12,288$ que será multiplicada por ΔV_i, assim estamos reinserindo os pesos que não inserimos ao fazer W_V ter 128 dimensões e finalmente:

$$\Delta E_i := W_o \Delta V_i$$

Podendo redefinir E_i de volta para:

$$E_i := E_i + \Delta E_i$$

Para o passo seguinte.

5 Multilayer Perceptron

Após passar pelo mecanismo de atenção. A sequência de vetores de embedding será alimentada por um *Multilayer Perceptron*. Cada vetor E_i passará por uma quantidade de operações, gerando um vetor E_i', e, no final, fazemos $E_i := E_i + E_i'$. Todas essas operações são realizadas em paralelo.

Primeiramente, estaremos pegando nosso E_i e multiplicando ele por uma matriz de $49,152 (= 4 \cdot 12,288)$ dimensões. Isso, na prática, tenta fazer algumas perguntas específicas para o embedding, ilustrando esse multiplicação como vários produtos escalares com E_i, a resposta à pergunta é afirmativa se o produto escalar for positivo, e negativo, caso contrário. Depois, normalizamos essas respostas passando o vetor resultante por uma função de normalização $ReLU$, que simplesmente transfomar cada valor $E_{i,j}$ em $max(0, E_{i,j})$. Por fim, fazemos outra multiplicação de matrizes, para E_i voltar para o número de dimensões original, e essa multiplicação pode ser vista como informação adicionada a cada parâmetro do embedding quando ele não foi nulificado pelo $ReLU$. É importante informar também que após cada uma das duas multiplicações por matriz feita, o vetor resultante é somado por um vetor *Bias*, que além de fazer parte de como redes neurais clássicas funcionam, também simplifica a informação adicionada ao vetor.

5.1 Lema de Johnson Lindenstrauss

Pesquisar Sparse Autoencoder

5.2 Positional Encoding

Pela natureza das operações que realizamos nos embeddings dos tokens, nós não guardamos a informação de posicionamento de cada token na sequência e, com isso, podemos, obviamente, perder informações de interesse na nossa sequência. Para corrigir isso, somamos a cada vetor de embedding um vetor que irá dar informação sobre o posicionamento desse token na sequência de input. Os pesos deste vetor podem ser parâmetros aprendidos, mas, na arquitetura original do Transformer, o embedding de posições (PE) é determinado por uma função definida:

$$PE_{pos,i} = \begin{cases} \sin\left(pos/10000^{\left(\frac{i}{d_{model}}\right)}\right), & i \equiv 0 \mod 2 \\ \cos\left(pos/10000^{\left(\frac{i-1}{d_{model}}\right)}\right), & i \equiv 1 \mod 2 \end{cases}$$

onde pos é a posição absoluta do token na sequência de input; i é a dimensão atual do vetor (a posição do vetor que aquele valor é inserido); d_{model} é o tamanho da dimensão dos vetores de embedding (no caso do GPT-3, este é aquele número $12,288$).

A motivação para essa escolha é a seguinte, é importante para um modelo de linguagem estabelecer relações de posicionamento relativo entre tokens, por exemplo, um adjetivo provavelmente vai dar significado aos tokens mais próximos a ele; o conteúdo cuja posição está entre duas aspas é uma referência a algo ou um nome, etc.

A escolha dessa função é feita porque possibilita que tokens possam facilmente atender à posições relativas já que a codificação de um offset de posições pode ser representado como uma combinação linear da codificação de posições dadas.

$$\sin(\alpha+\beta) = \sin(\alpha)\cos(\beta) + \cos(\alpha)\sin(\beta)$$

$$\cos(\alpha+\beta) = \cos(\alpha)\cos(\beta) - \sin(\alpha)\sin(\beta)$$

mas, temos que:

$$\begin{bmatrix} \sin(\beta) & \cos(\beta) \\ -\sin(\beta) & \cos(\beta) \end{bmatrix} \begin{bmatrix} \sin(\alpha) \\ \cos(\alpha) \end{bmatrix} = \begin{bmatrix} \sin(\alpha)\cos(\beta) + \cos(\alpha)\sin(\beta) \\ \cos(\alpha)\cos(\beta) - \sin(\alpha)\sin(\beta) \end{bmatrix}$$

Por causa deste resultado, estamos dizendo que o modelo pode aprender a atender à relações relacionadas à posição relativa entre tokens. Isto é, caso for do interesse de um Q_{pos} obter informações sobre um $K_{pos+\phi}$, então, o produto interno entre eles irá trazer informação sobre como esses tokens se relacionam por conteúdo e por posição. O modelo irá adaptar as matrizes de peso W_K e W_Q para fazer com que o vetor resultante Q_{pos} aponte no espaço para uma região em que o token pos está interessado em $pos + \phi$ e o token $pos + \phi$ vai dizer a respeito da posição em que ele se encontra em $K_{pos+\phi}$, portanto, os vetores estarão mais próximos no espaço e, por fim, seu produto escalar será maior.

Além disso, como cada valor é obtido por uma função seno ou cosseno, os valores estão apenas em $[-1, 1]$

6 Tokenização

O processo de tokenização da sequência em tokens de interesse pode ser feita usando SentencePiece, do Google; ou tiktoken, da OpenAI.

7 Fontes

1. Attention Is All You need, 2017
2. 3blue1brown

3. https://medium.com/thedeephub/positional-encoding-explained-a-deep-dive-into-transformer-pe-65cfe8cfe10b